O'REILLY®

MLOps with Databricks

Machine Learning and GenAI Applications End to End



Maria Vechtomova

Forewords by Matei Zaharia & Alejandro Saucedo

Praise for *MLOps with Databricks*

This book goes beyond theories. If you want to learn the battle-tested best practices of MLOps and LLMOps, it will walk you through how to implement them with popular Python libraries and deploy them to Databricks. It's an essential guide in the agentic era.

—Jason Yip, Databricks MVP and director of data and AI, Tredence

This is much more than a book on how to implement MLOps with Databricks; more specifically, it is a real and practical guide to not just doing it, but doing it well—and in a more complete manner than most are willing to undertake. By covering topics like appropriate CI/CD processes for machine learning and AI applications, as well as how Databricks and MLflow can help designers and organizations adhere to the guidelines for AI set out by the EU, this book can be a useful reference for aspiring and practicing data scientists.

—Tristen Wentling, coauthor of Delta Lake: The Definitive Guide

Maria has spent her career closing the gap between building models and running them reliably. This book captures that hard-won experience and pairs it with a platform that can actually deliver on it. Essential reading for anyone serious about production AI.

—Hamza Tahir, cofounder and CTO, ZenML

Maria has been teaching MLOps to engineering teams for a while now and leading the way in sharing best practices from her own experiments at work. Everything she teaches in this book is informed by her own learnings and first-hand view into Ops practices within several engineering companies whom she has trained and shaped along the way. That alone makes it one of the best practitioner guides in this space.

—Abi Aryan, inference infrastructure engineer, author LLMOps

Maria's material has been polished by the process of having to explain it over and over again, and also having done it—a vital combination that you don't always find. This text is one of the easiest ways you can find, on one of the leading platforms, to get started with all this bewildering ML stuff.

—Niall Richard Murphy, distinguished engineer; coauthor of *Site Reliability Engineering* and *Reliable Machine Learning*

To me, Maria's work here captures modern MLOps—the principles that have stuck around—concretely implemented on a widely-used tech stack. This book would have shortcutted me years of working in this space. Amazing job, Maria!

—Eric Riddoch, former ML platform lead at BENlabs, Pattern, and Subquadratic

I've known Maria since the middle of the MLOps hype cycle back in 2022. Even then, she was one of the most knowledgeable and grounded people I knew, focused on the fundamentals that actually help you ship ML products. She made a bet on Databricks winning the MLOps world, building a course, a book, and a consultancy business around it—a bet that paid off. With this rare combination of being a true expert and knowing exactly what and how to teach, there's probably no better person from whom to learn MLOps with Databricks.

—Paul Iusztin, founder of Decoding AI and author of LLM Engineer's Handbook

Maria Vechtomova has written the MLOps book I wish I had earlier in my career. It is principles-first and code-heavy, built around one real application that moves from a prediction model to an AI agent. The fundamentals are here, and they still matter: traceability, reproducibility, observability, reliability. Maria explains them with the kind of practical detail that comes from doing the work. If you are building on Databricks and want real advice for production, keep this one close.

—Denny Lee, PM director, startups & ecosystem, Databricks

This book represents a milestone in codifying MLOps as a discipline for modern ML and AI systems. Only systems that run reliably in production create value. Our job does not end when a feature is built and integrated; it ends when users can rely on the product in production. The lessons presented here go far beyond tooling. They should be known by ML engineers and data scientists, and they are highly valuable for SREs and platform engineers as ML models become an increasingly important production workload.

—Dr. Heinrich Hartmann, lead SRE at Zalando, founder at SignalsConf

Maria derives and draws from her vast industrial experience as a platform engineer, hands-ons practitioner, tech leader, founder, and an educator. She charts your journey into the imperative and fast evolving field of ML/LLM/AIOps, regardless of where you are in your career. At all levels of this field, the book has something precious to offer and apply.

—Jules S. Damji, technical staff, developer relations, Databricks

MLOps with Databricks

Machine Learning and GenAI Applications End to End

Maria Vechtomova



MLOps with Databricks

by Maria Vechtomova

Copyright © 2026 Mariia Vechtomova. All rights reserved.

Published by O’Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black

Development Editor: Angela Rufino

Production Editor: Christopher Faucher

Copyeditor: Doug McNair

Proofreader: Carol McGillivray

Indexer: WordCo Indexing Services, Inc.

Cover Designer: Susan Brown

Cover Illustrator: Monica Kamsvaag

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

August 2026: First Edition

Revision History for the First Edition

- 2026-08-07: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798341608252 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *MLOps with Databricks*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-60825-2

[LSI]

Foreword by Matei Zaharia

Bringing machine learning and AI into production is not just about the model. Most of the work, and most of the risk, lies in the operational process surrounding it: tracking what you built and how, reproducing a result months later, deploying it safely, and knowing when it has quietly stopped working. As organizations race to bring models, and now agents, into production, that work has only grown more important.

I have spent much of my career on this problem. When we built MLflow, we kept hearing the same thing from teams everywhere: machine learning development is complex in ways that traditional software is not. There are too many tools to stitch together, it is hard to track what went into a result, hard to reproduce it later, and hard to move a model out of a data science notebook and into production. None of that is about the algorithm. We built MLflow to be open, working with any library or tool rather than locking teams into one stack, and later extended that approach to govern data and AI together in Unity Catalog. The goal throughout has been to make ML and AI development as robust and predictable as the best traditional software.

What has been missing is a resource that connects these pieces. There is no shortage of tutorials for any single tool in ML engineering, but practitioners demanded a guide showing how they fit into one coherent practice, from a first notebook to a monitored, governed AI product that a team can trust. That is what this book provides, based on Maria’s deep experience building successful ML products at multiple companies.

This book is principles first and code heavy. It does not stop at traditional ML, but covers the whole engineering and operations process. The same lifecycle that carries a model from notebook to production is extended to LLM-based and agentic systems, with a real application built end to end. Few books cover both worlds, and fewer still show how much they share. The tools will keep changing. The principles of traceability, reproducibility, observability, and reliability will not.

The distance between a model in a notebook and a system in production has never been more worth closing. This book shows you how, in a clear and easy-to-follow fashion based on deep experience.

Matei Zaharia
CTO and cofounder, Databricks

Foreword by Alejandro Saucedo

Today, many teams can build models and many can build platforms. However, far fewer can connect those two disciplines with the clarity and depth required to build platforms and engines that let organizations continuously deliver value with high speed, reliability, and trust.

Maria has spent years doing exactly that. Her work across machine learning operations (MLOps), platform thinking, and now large language model operations (LLMOps) reflects not only her technical fluency but a grounded understanding of what it takes to move from experimentation to reliable, repeatable delivery. As I have had the pleasure to know Maria for many years, I can confidently say that she belongs to a rare group of practitioners who can bridge the gap between two things that often remain divided in the field of machine learning: the rigor of software/platform engineering for production ML and the flexibility to empower scientists with the ability to deliver business value. Today, bridging that gap matters more than ever.

When Maria mentioned that she was going to consolidate all the knowledge she had gathered in a book, I was very excited. As I have been able to read the multiple iterations of this book, I can say that it does what it initially set out to do. It does not treat MLOps as a checklist of processes or the Databricks platform as a collection of isolated features. Instead, it approaches both from first principles. Maria breaks down how real production excellence comes from how architecture, governance, reproducibility, deployment, and monitoring fit together as one operating model. That is why this book will be useful not only to practitioners who want to deepen their Databricks expertise but also to leaders and engineers who are trying to build AI systems that can stand up to the demands of scale, compliance, and real-world change. Maria writes from hard-earned experience, and this book captures lessons that many teams only learn the expensive way.

Alejandro Saucedo
AI and tech executive, Zalando SE
AI expert for AI Advisory Group, United Nations

Board member, ACM

Preface

I have been working in Data and AI for over a decade, and I've devoted most of my career to machine learning operations (MLOps). I started doing it before MLOps became a formalized term in the late 2010s. Even though MLOps is an established term today, too few companies and teams follow proper MLOps principles.

In many teams I encounter today, there tends to be an imbalance in expertise. Some team members have deep experience in data science and know how to build and monitor machine learning (ML) models, but they are less familiar with development and operations (DevOps) practices. Others come from a strong software engineering background and follow DevOps principles rigorously, yet they often treat the ML or AI component as a black box.

Neither situation is inherently problematic, but both present challenges when the team's goal is to build reliable ML/AI systems that enable other teams or the organization to continuously deliver value at an accelerated pace. Success requires bridging the gap: acknowledging the nondeterministic nature of many ML/AI components while also applying the discipline of DevOps practices.

This book focuses on traditional ML. About a third of it is devoted to large language model operations (LLMOps), which is a specialized subset of MLOps that focuses on applications built around large language models and, more broadly, foundational and multimodal generative models.

In that sense, MLOps provides the *foundation*: the principles and practices for building, deploying, and maintaining ML systems. These principles and practices extend naturally to newer classes of models, even as the emphasis often shifts from training to inference.

While foundational models have unlocked powerful new capabilities and are rapidly being adopted across a wide range of use cases, traditional ML

is not going away. Many real-world systems continue to rely on it, and the core principles behind it still underpin modern AI applications.

My goal is to help to professionalize the field of ML (and AI, as a broader concept) and make the AI systems we build more reliable and robust, so science teams can deliver value faster and with higher levels of trust.

Why Databricks?

I believe MLOps is not about tools but about principles. However, I strongly believe the best way to learn MLOps is with hands-on practice, using the tools directly. This is necessary because different interconnected components are required to support these principles. No single tool is perfect, and all of them are tailored to the most common use cases their users deal with.

When companies are designing their MLOps platform, they must choose between two options:

1. They can build the end-to-end planform from separate heterogeneous best-of-breed tools—and then face all the complications that come with combining all the tools together. Organizations that choose this path often end up going for a combination of open source tools and closed source vendors.
2. Alternatively, they can use an existing “all-in-one” MLOps platform vendor.

This is a tough choice, and which option is the right one depends on many factors, such as the knowledge level and size of the team maintaining the platform, the specifics of the infrastructure setup (e.g., on premises, multi cloud, single cloud), what other tools the company uses, and who would be the end users of the platform.

Various “all-in-one” MLOps platforms exist. There’s Azure ML, AWS Sagemaker, Vertex AI, and Databricks, among others. Even though none of

them is perfect and they all have their own pros and cons, for me, Databricks has stood out from the pack throughout the years.

Databricks was founded by the creators of Apache Spark, and historically, it has been used widely by data engineering teams. Delta Lake was originally developed by Databricks, and it is the most widely used lakehouse storage format in the world. With its acquisition of Tabular, a company behind Apache Iceberg, and the open sourcing of Delta, Databricks has reinforced its commitment to open source. Databricks also created Unity Catalog, which it released as open-source software (OSS) in 2024. It's a governance layer that provides centralized access control, discoverability, and lineage for all data assets (including models). Since data is at the heart of any ML and AI system, it simplifies collaboration between data engineering and ML/AI teams.

As the ML field has evolved, Databricks has emerged as one of the leading players in MLOps, with many of its tools having become prevalent in the production ML space. In fact, Databricks was ranked in the top three across most categories in the 2025 State of Production ML Survey from the Institute for Ethical AI & Machine Learning.

One flagship tool is MLflow, which is well known and widely used. It was one of the first tools for tracking experiments and registering models. However, not everyone knows that MLflow was developed by Databricks and comes integrated with its enterprise platform's workspaces as a managed MLflow service, which simplifies a lot of aspects of MLOps.

Finally, Databricks has also acquired Tecton,[1] which is one of the leading players in feature store engines. It closes the gap between the data lake and ML data pipelines.

Databricks' focus on open source and making these tools part of its platform makes Databricks a strong player in the data and AI field. These are some of the reasons why Databricks has been my tool of choice for this book. This preference of mine is also shaped by my extensive hands-on experience using Databricks in real-world settings, both as an engineer and through my consulting work. Over the years, I've led the deployment of

production ML/AI systems, supported MLOps and LLMOps adoption in large enterprises, and trained hundreds of practitioners through courses and workshops.

So having worked across a range of tools and platforms, my preference for Databricks is not just based on my familiarity with it from using it extensively. I've repeatedly seen where it stands out from the crowd when the challenge shifts from building AI demos to operating AI systems reliably in production.

Who and What This Book Is For

This is a technical book. That means you should have some working experience in data science, data, ML, or AI engineering. All the code covered in the book is written in Python, which is commonly used for developing AI applications. It makes it easier to go through the book if you have already worked with Databricks before, but it's not a prerequisite.

This is also a book with a lot of code. To benefit from it the most, you should dive into the code and try it out step by step. On the other hand, I do not want you to execute the code blindly. That's why this book focuses on the MLOps principles first and explains how using different components of Databricks in the right way helps support these principles.

This book also covers architectures and offers different considerations for why a certain architecture would be a good choice for a particular use case. Data and AI architects will also benefit from the book even if they don't try out the code.

Writing a technical book covering different platform features of Databricks is not easy. Based on my experience, this is one of the fastest-evolving products. I realize that parts of the book may become outdated, and I hope to minimize this risk by taking a "principles first" approach. This means that although some of the code, tools, and technical details may become out of date, the theoretical foundations and best practices presented should not. I believe that the book will help a lot of data and AI professionals improve their ML workflows by making them more robust and reliable.

I have written this book as the handbook I wish I'd had when I was first getting started with Databricks. One of the pain points I've found is that while there are plenty of examples of how to use different components of Databricks, there are very few examples that bring a cohesive view of how all these components actually work together. I hope this book will provide those examples for you, save you a lot of time, and help you on your MLOps journey.

How This Book Is Organized

The book focuses on two main use cases: traditional ML application (which I use to demonstrate the concepts I present in Chapters 2 through 6) and an LLM-based application (which I cover in Chapters 7 through 9). It also covers how to deploy these applications in a robust way.

In Chapter 1, I start the book by breaking down the MLOps principles and components that are relevant for any AI application.

In Chapter 2, I take you on a deep dive into different options for developing an application on Databricks. I also explain the relevant details of the Databricks infrastructure, and I highlight the importance of following best software engineering practices.

In Chapter 3, I go through the main components of MLflow for traditional ML, and I showcase how you can use these components for experiment tracking and registering the models. MLflow is tightly connected to the Model Serving feature on Databricks.

In Chapter 4, I discuss the different Model Serving architectures and when you should use each of them, and I go through the practical implementation of those architectures.

In Chapter 5, I focus on the deployment of the whole ML pipeline using Declarative Automation Bundles. The chapter brings together all the code that we discussed in the previous chapters and shows how all the steps of the pipeline are related to each other. We will also talk about Databricks platform design, deployment strategies, and testing.

In Chapter 6, I cover the Data Quality Monitoring functionality. I highlight the importance of monitoring and show you how to use inference tables to simplify monitoring for Model Serving endpoints.

Chapters 7 through 9 form a special section in which I focus on the LLM capabilities of Databricks and show you how the principles I cover throughout the book apply to AI applications in practice. I specifically

wanted to include this material because of the industry's shift from traditional ML toward AI systems and the growing demand around them. Earlier drafts of this book covered all of this in just one chapter, but the operational surface turned out to be too large to do justice to it all in one place, so I split it into three. Each chapter shows, in practice, how much of what we covered in Chapters 2 through 6, also applies to LLM-based applications. Chapter 7 covers foundation models and context engineering. Chapter 8 turns to MLflow for GenAI and how it supports evaluation, tracing, and the development workflow for these systems. Chapter 9 covers deploying and monitoring these systems in production. This includes cost monitoring, since for GenAI, workload spend is one of the operational signals you must watch the closest.

In Chapter 10, I explore how to navigate AI regulations, implement security best practices, and establish model governance using the Databricks platform. I also tie these topics back to the operational, deployment, and lifecycle management principles discussed throughout the book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

> Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/mvechtomova/oreilly-mlops-with-databricks*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a

program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*MLOps with Databricks* by Maria Vechtomova (O'Reilly). Copyright 2026 Mariia Vechtomova, 979-8-341-60825-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/MLOps-with-Databricks*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book took a long time to write, and many people influenced it.

I am grateful to the reviewers who gave their time and care to the manuscript. Tristen Wentling, Alex Ott, Alejandro Saucedo, Denny Lee, and Jason Yip offered detailed, valuable feedback that made the book sharper than I could have ever made it alone. I also want to thank the anonymous reviewer whose comments were so thoughtful and detailed that I am still very curious who they are.

Some of the people who shaped this book did so without knowing it. My students Jan Koch, Mehmet Acikgoz, Victor Kuztentsov, Cao Tri Do, and Nikhil Komakula asked the kind of questions that made me think harder about ideas I assumed I had fully explored. Many of the ideas here came out of conversations. Thanks to Jeroen Meulemans for pointing out changes in Databricks features and implementation details. Thanks to Hamza Tahir, Eric Riddoch, Paul Iusztin, and Hugo Bowne-Anderson for discussions that push me to think in new ways. The SRE community sharpened the way I think about reliability and observability, and I am grateful for conversations at the SRE conference and beyond with Alejandro Saucedo, Charity Majors, Sylvain Kalache, Heinrich Hartmann, Niall Murphy, and Todd Underwood. The last two told me not to write a book, and I hope they will forgive me.

Thank you to Başak Eskili, with whom I taught; to my co-founders at Cauchy, Casper Lubbers and Eirini Papakosta, for the partnership and the conversations that run through this work; and to the rest of the Cauchy team.

Thanks to the wonderful team at O’Reilly, especially my editors Angela Rufino and Christopher Faucher, who made working on this book a great experience.

To my family, thank you for your patience while I wrote. And thanks to my parents who led me to this book by example. My father, Evgenii Vechtomov, has written more than five hundred works in mathematics, and my mother, Vera Varankina, has written many of her own. For as long as I can remember, the two of them have shown me what it means to sit down and write.

Finally, my cat Marx practically co-wrote this book, mostly by sitting between me and the laptop the entire time. Any mistakes or typos are his fault.

I have certainly left out people who should be here. The omission is a failure of my memory, not of my gratitude. This book is, in the end, an accumulation of lessons I have learned over the course of my career.

Everyone I have worked with has taught me something about what it takes to bring AI to production.

---

[1] For more information, check out Databricks' announcement of its acquisition of Tecton.

Chapter 1. MLOps Principles and Components

Machine learning operations (MLOps) is a set of practices that help to bring ML models to production in a scalable and efficient way. It can be seen as an extension of the development and operations (DevOps) philosophy that started around the end of 2000 as a movement against the established, traditional software development model, in which code developers and developers who deploy and support the code are separated functionally and organizationally. Siloed teams—with their own planning and key performance indicators (KPIs), inability to communicate code to the operations team, and misaligned development and production environments—contributed to long release cycles and created endless frustration.

MLOps addresses the same core challenge as DevOps did: building systems that can be reliably deployed, operated, and improved over time in production. The difference is that AI systems introduce additional parties—such as data scientists, ML engineers, and other specialized roles—as well as new types of challenges related to evaluation, testing, and reproducibility.

Unlike traditional software systems, ML applications are built not only from code but also from data. Their behavior is inherently probabilistic, which makes it significantly harder to define expected outcomes, write meaningful tests, or validate system performance before deployment.

These characteristics make the development lifecycle of ML systems fundamentally different from that of traditional software. This means that there needs to be closer collaboration between data practitioners and platform or DevOps teams to ensure that models can be deployed, monitored, and maintained in a reliable and repeatable way.

More broadly, these challenges extend to AI systems as a whole. Since ML is a subset of AI and forms the foundation of most modern AI systems, the same principles apply, even as systems become more complex.

In this chapter, I will walk you through the evolution of MLOps, explain where and why it diverges from traditional DevOps practices, and outline the key components required to operationalize ML systems. By the end of the chapter, you will understand why these components exist, which MLOps principles they support, and how to implement them within the Databricks platform.

The Rise of MLOps

The term *MLOps* began gaining widespread popularity around 2020. This was roughly five years after many large organizations started heavily experimenting with and investing in data science, only to realize that their return on investment (ROI) was often lower than expected. While building ML models proved relatively straightforward, integrating them into production systems in a reliable and scalable way turned out to be far more difficult.

In practice, bringing ML models into production requires far more than the model itself. ML code produced by data scientists represents only a small fraction of a real-world ML system. This is illustrated in Figure 1-1, which is an adapted diagram from the well-known Google paper "Hidden Technical Debt in Machine Learning Systems."[1] It shows that the majority of the system consists of surrounding infrastructure such as data pipelines, validation logic, monitoring, serving layers, and integration components.

To make the outputs of ML models valuable to a business, models must be integrated into existing software applications or entirely new applications must be built around them. This is a responsibility that has traditionally fallen within the domain of DevOps teams.

Historically, data analytics and later data science were closely aligned with business functions. As a result, many data scientists had limited exposure to

software engineering best practices and the broader IT landscape. Deploying scalable ML applications often required formal handovers to DevOps teams, and that created organizational bottlenecks. In many ways, this mirrored the disconnect that once existed between development and IT operations teams prior to the DevOps movement.

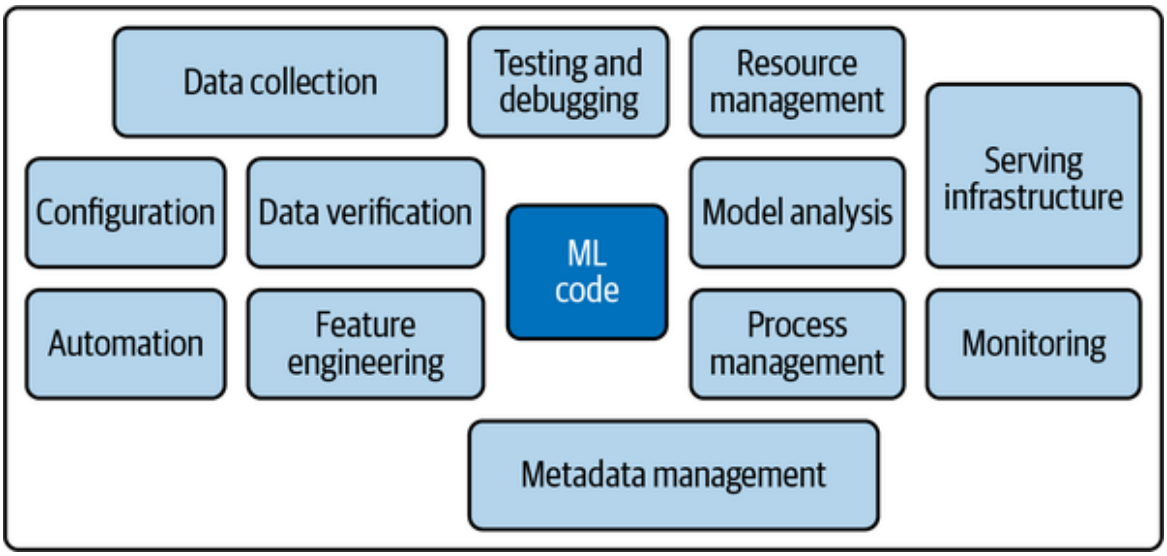


Figure 1-1. Diagram showing ML code represents only a small fraction of a real-world ML system

A shift was therefore needed. In recent years, many organizations have begun embracing MLOps practices by moving away from centralized data science teams and toward more product-centric, cross-functional teams. This shift has given rise to the role of the *machine learning engineer*, which is embedded within these teams to enable end-to-end ownership of ML-powered products.

At the same time, platform teams have started adopting MLOps principles by building and supporting internal tooling that enables ML product teams to reduce time to production. As a result, a broad ecosystem of tools and platforms has emerged to support the operationalization of ML systems. Similarly, leading tech companies have built end-to-end platforms for their ML operations, some which have become iconic in their influence in the field. Examples include Uber's Michelangelo,[2] LinkedIn's ProML,[3] and Netflix's ML Platform.[4]

Today, organizations are in a similar phase, this time with AI applications. Many teams are actively experimenting with LLMs and other AI technologies, and it is likely that a stronger operational focus will follow soon, probably faster than it did in the case of MLOps.

Because many organizations have already undergone the structural shift toward product-oriented teams and platform enablement, the adoption of LLMOps practices is expected to accelerate. In addition, the ecosystem of tools and platforms available today is significantly more mature than it was during the early wave of ML adoption, so it's easier for organizations to move from experimentation to reliable, production-grade AI systems.

MLOps Principles

We've established how MLOps became a discipline in its own right, so now, let's look at its principles. These principles matter because companies rely on ML models as part of their business operations, and real negative business impacts follow when model predictions are unavailable or of poor quality.

There are many examples out there of how things can go wrong. Zillow, a real estate company, lost $500 million in 2021 due to houses that were bought at too high a price because of inaccurate ML predictions.[5]

The goal of MLOps is to minimize the risks of things going wrong. This brings us to the *four main MLOps principles*: reliability, traceability, reproducibility, and observability. Each principle provides the answer to its own core question (see Table 1-1).

Table 1-1. The four main MLOps principles and the core questions they answer

Principle	Core question
Traceability	What happened?
Reproducibility	Can we recreate it?
Reliability	Can we depend on it?
Observability	Can we understand and diagnose it?

These are developer-centric principles and focused on what engineering teams can control and implement. Broader ethical principles, such as fairness, privacy, and accountability, are equally important but fall outside the scope of this book. The Linux Foundation's Trusted AI Reproducibility, Robustness, Equitability, Privacy, Explainability, Accountability, Transparency, and Security ((R)REPEATS) principles framework is a good reference you can use to get for the full picture on the broader ethical principles.[6]

The following sections will cover the four main MLOps principles in detail.

Traceability

Traceability means that for any given ML model run or deployment in any environment, you can look up and unambiguously find all of the following:

- Corresponding code and commits on Git
- The infrastructure used for training and serving
- ML model artifacts
- What data was used to train the ML model

In practice, this often breaks down. For example, a model may underperform months after deployment, and the team may not be able to determine what code, dataset snapshot, hyperparameters, and configurations were used. If the investigation depends on searching Slack messages, notebooks, or institutional memory, traceability has failed.

Effective traceability requires every run to automatically capture and link its inputs and outputs through a shared run identifier. Table 1-2 lists the traditional ML traceability components and gives examples for each of them.

Table 1-2. Traditional ML traceability components and examples

Traditional ML traceability components	Examples
Code and commits	Git SHA
Infrastructure	Training compute and serving environment configuration as code
Data lineage	Dataset snapshots and table version
Features	Feature definitions, transformations as code, and the feature store version
Model configuration	Hyperparameters and model architecture
Artifacts	Trained models and evaluation reports
Experiment tracking	Metrics, parameters, and runs

Modern LLM and agentic systems add more layers of complexity because behavior depends not only on models and data but also on prompts, retrieval pipelines, tools, and external APIs. Table 1-3 Lists LLM-based system traceability components and examples of them.

Table 1-3. LLM-based system traceability components and examples

LLM-based system traceability components	Examples
Foundation model usage	Endpoint, model version, token usage, and latency
Prompt lineage	Prompt templates, system prompts, and query enrichment
Retrieval pipelines	Retrieved chunks, ranking strategy, and metadata filters
Embedding generation	Embedding models, chunking strategy, and parsing logic
Vector database lineage	Chunk metadata, source documents, and indexing strategy
Tool and API usage	External tools invoked, parameters, and responses
Response lineage	Which model, prompt, context, and tools produced a response

Effective traceability requires that every run automatically capture and link its inputs and outputs—code commits, infrastructure, datasets, prompts, model parameters, retrieved context, tool invocations, artifacts, and evaluation results—and tie them all to a single identifier. The key is that traceability must be automatic and built into the workflow, not dependent on manual documentation.

Teams that invest in this capability spend far less time putting out fires and far more time improving their systems.

Reproducibility

Reproducibility extends traceability one step further. Traceability asks, “Can we determine what happened?” Reproducibility asks, “Can we make it happen again?”

For traditional ML systems, reproducibility means that the same code, data, configuration, and environment produce the same model and results. This makes experiments verifiable, debugging easier, and rollbacks reliable.

In practice, reproducibility takes effort. Without data snapshots, pinned dependencies, and versioned configuration, reproducing a previous result becomes difficult or impossible. Modern tooling helps address these issues. Data versioning systems, containerized environments, experiment tracking, and infrastructure-as-code (IaC) allow teams to re-create previous training conditions reliably.

For LLM and agentic systems, reproducibility becomes more complex because system behavior depends not only on code and data but also on foundation model versions, prompts, retrieval pipelines, tool usage, and inference settings such as temperature or token limits. Exact bit-for-bit reproducibility is impossible due to nondeterministic model behavior and provider-side updates. Instead, teams aim for *functional reproducibility*: the ability to re-create a system configuration that behaves consistently within an acceptable range.

It is worth noting that the boundary between traditional ML and LLM-based systems is not as clear as it might seem. Nondeterminism in deep learning predates LLMs, and evaluation for natural language processing (NLP) applications (such as translation, and summarization) was never straightforward.[7] These were known problems, but they affected a smaller community. LLMs have made nondeterminism and evaluation difficulties impossible to ignore, simply because the scale and visibility of these systems has brought long-standing issues to the surface.

Reliability

Reliability is the principle that an ML system should function correctly and consistently throughout its lifecycle, even as code, data, and infrastructure change around it. A reliable system is one in which the team can ship changes with confidence that things won't break silently.

While reliability engineering traditionally focuses on measurement and operational resilience, reliability begins with what you can control most directly: the quality of the code itself. If the code is poor, no amount of monitoring, deployment strategizing, or incident response will be able to make up for it. You can operate around bad code for a while, but the operational burden will grow with every release, and eventually, the system will become too fragile to change safely.

Clean code ensures that models can be updated, debugged, and improved over time without introducing unintended consequences. This is not just a best practice; it is a pattern backed by decades of evidence. The industry average is about 1 to 25 errors per 1,000 lines of delivered code. Microsoft's Applications Division achieved 0.5 defects per 1,000 lines in released products through code reading and independent testing. The space shuttle's software achieved zero defects in 500,000 lines of code.[8] The systems with the highest stakes consistently have the cleanest code. That is not a coincidence.

Google's site reliability engineering (SRE) handbook puts it differently but arrives at the same place: simplicity is a prerequisite to reliability, and every new line of code is a liability.[9] The tools and frameworks that ML systems depend on, however, are not simple. ML pipelines and infrastructure configurations carry inherent complexity that go beyond clean code. Code quality in this context is not about simplicity in the absolute sense but about managing unavoidable complexity through the following:

- Clear structure and modular design
- Testing at multiple levels, from unit tests to integration tests
- Consistent standards that are enforced across the team

The complexity you cannot remove is exactly the complexity that demands the most discipline.

A common counterargument is that AI-assisted code generation changes this equation—if code is cheap to produce, it can be treated as disposable. The data tells a different story:

- CodeRabbit's analysis of 470 open-source pull requests (PRs) found that AI-coauthored PRs contain approximately 1.7 times more issues overall than human-only PRs.[10]
- The 2024 DevOps Research and Assessment (DORA) report found that a 25% increase in AI adoption produced a 7.2% drop in delivery stability.[11]

Generating code faster does not eliminate the cost of poor code; it shifts that cost downstream into debugging, maintenance, and production incidents. The ease of generation makes discipline more important, not less.

But code quality alone is not enough. It has to be protected by layers that reinforce each other, such as the following:

- Continuous integration and continuous deployment (CI/CD) pipelines that enforce standards automatically and consistently
- Safe deployment strategies, like canary releases and rollbacks, that limit the blast radius when something does go wrong
- Dependency management and environment pinning that prevent the kind of silent drift where a system works on one machine but fails on another

Weakness in any one layer undermines the rest. The concrete practices that operationalize this, from linting and CI enforcement to code reviews and AI-assisted tooling, are covered in the team practices section later in this chapter.

Reliability also extends into infrastructure and operational resilience, which are areas that are well established in SRE. The book *Reliable Machine*

Learning by Cathy Chen et al. (O'Reilly) applies SRE principles specifically to ML systems by covering topics such as deploying across multiple availability zones, implementing authentication patterns for model endpoints, designing fallback strategies when a primary model is unavailable, and ensuring the system can scale to meet demand. These are important considerations for any production ML system, and we will touch on some of them throughout this book, but they are not its core focus.

Observability

Contrary to popular belief, the lifecycle of an AI system does not end at deployment; in many ways, deployment is where the real lifecycle begins. Once it's exposed to real users, changing data, evolving requirements, and production constraints, the system enters its most operationally complex phase. And that's where teams need observability the most.

An *observable* system is one whose internal state you can understand from its outputs, even for problems you did not anticipate. In practice, this means the system's features must include rich logging, distributed tracing, and the ability for you to ask it new questions without deploying new instrumentation. There is natural overlap between observability and the traceability principle we discussed earlier: traceability focuses on the ability to reconstruct what happened and why a decision was made, while observability is about having the instrumentation in place to detect and diagnose issues in real time. The two reinforce each other.

A core component of observability is *monitoring*: tracking known metrics and alerting when they cross a threshold. While observability helps you figure out why something went wrong, monitoring tells you that it did. It is the natural starting point, so here's a list of what an ML system needs to track:

Application-level data

> Response times, error rates, throughput, and latency distribution, all of which are relevant for a model served behind an API)

Infrastructure health

> CPU health, memory utilization, and health metrics for various system components

System-relevant business metrics

> For example, KPIs such as the click-through rate and conversion rate for a recommender system

ML systems differ from traditional software systems in one critical respect when it comes to monitoring: you need to monitor their data quality and detect data and model performance drift.

A software system will perform as expected as long as the related infrastructure, environment, code, and data models do not change. You have control over all that with software systems, but not with ML and AI systems. That's the key issue in ML: even if everything under your control remains unchanged, the model can start performing poorly when the data distribution shifts. Figure 1-2 illustrates this difference and this challenge.

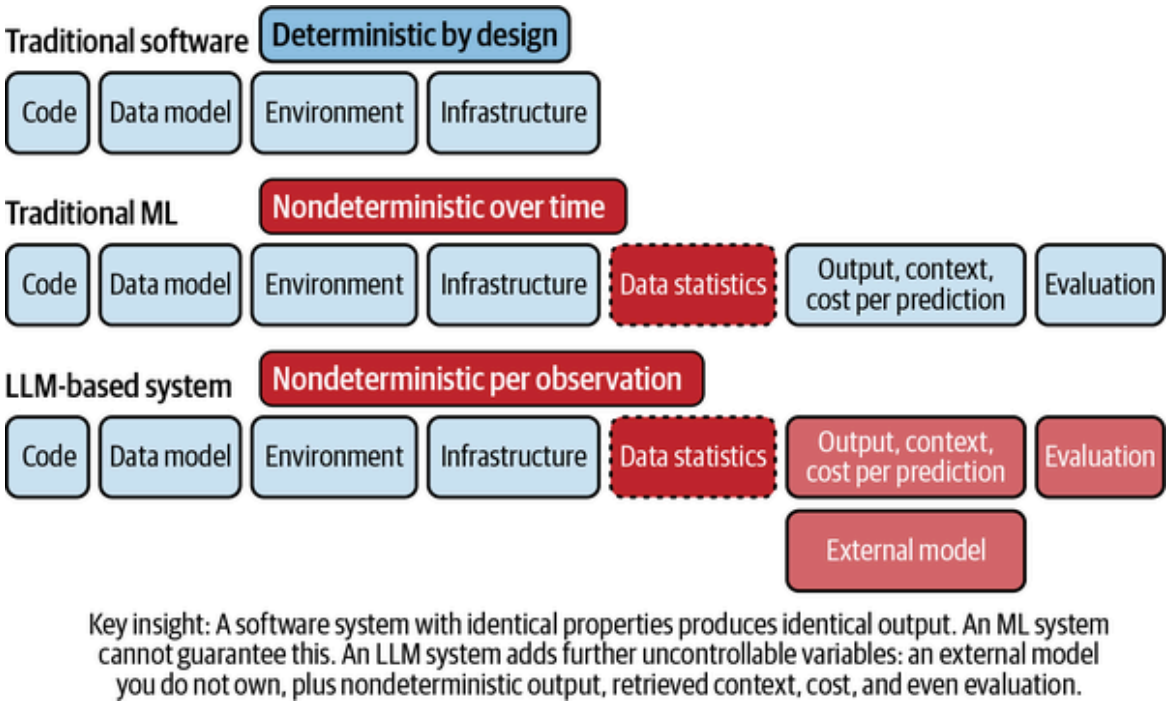


Figure 1-2. The complexity ladder: traditional software versus traditional ML versus an LLM-based system

In addition, data quality issues are some of the most common reasons why ML applications fail in production. Missing values, schema changes, incorrect labels, upstream pipeline failures, and delayed data availability can all break ML system performance, even when the underlying model and code remain unchanged. Unlike traditional software bugs, these issues often originate outside the application itself, so they're harder to detect and debug unless proper monitoring is in place.

In traditional ML, production monitoring of model performance is well established. You track metrics such as accuracy, precision, and the area under the curve (AUC) against incoming data and watch for degradation over time. When performance drops, you investigate whether the data has shifted, whether labels have changed, or whether something upstream has broken. The metrics themselves are deterministic and well defined.

For LLM-based systems, the challenge is not just tracking metrics but also defining them. Output quality is harder to quantify, and evaluation itself becomes a two-tier problem: using deterministic metrics where they apply and LLM-based evaluation, where another model judges the output, for

everything else. On top of that, cost and latency vary by request and become first-class monitoring concerns in their own right. We will cover production monitoring and evaluation in depth in later chapters.

Team Practices

Principles don’t survive without the right culture and knowledge sharing. Reliability, traceability, reproducibility, and observability all depend on tooling, but tools only work when teams use them consistently and intentionally. This section covers the practices that make that possible.

Documentation and Knowledge Preservation

People affected by a system should be able to understand what the system is intended to do, how it was designed, and where its limitations lie. In ML and AI systems, this is fundamentally about making systems legible to humans: the engineers who maintain them, the business stakeholders who rely on them, the auditors who review them, and the users who interact with them.

Proper documentation should include the following:

- The assumptions behind the system
- The reasoning behind architectural choices
- The intended scope of use
- The known limitations or risks

Several artifacts support this in practice:

Model cards

: These explain model behavior and failure modes.

Data dictionaries

: These clarify what data represents and where it originates.

Architecture decision records

> These capture why specific trade-offs were made.

Documentation should live alongside the code, evolve through the same review process, and remain tightly connected to implementation. Lightweight formats such as Markdown are effective because they are version controllable, easy to maintain, and easily consumed by both humans and AI tooling. Tools such as MkDocs can generate documentation sites from Markdown, and documentation updates should be part of the code review process. Tools like Swagger are also useful for generating live API documentation that can be embedded into internal portals such as GitHub Pages and Confluence.

The rise of AI-powered coding assistants like Claude Code further increases the value of transparent systems. Coding assistants can summarize implementations or generate code changes, but they depend heavily on high-quality contextual information. As documentation becomes part of the development interface for both humans and AI systems, it shifts from being viewed as overhead to becoming an asset that improves development speed, onboarding, and long-term maintainability.

Ultimately, documentation is part of the culture. Organizations need norms and processes that encourage explicit communication, shared understanding, and knowledge preservation. Assigning ownership of critical documentation and conducting periodic reviews helps organizations ensure that important context does not disappear when team members leave or projects change direction.

Code Hygiene

Code quality is the foundation that all four MLOps principles rest on. A system can have perfect observability dashboards and comprehensive traceability, but if the underlying code is messy, brittle, or inconsistent, it will undermine every other investment.

Today, AI is real leverage when writing code, but leverage without discipline just breaks things faster. Protecting the main branch and enforcing automated checks before pull requests are merged creates a reliable safety net. Therefore, I advocate the following code hygiene protocols:

Pre-commit checks

> You should run these as part of a CI pipeline. This point is worth clarifying. Traditionally, pre-commit hooks run locally, before a developer commits code, to enforce formatting and basic coding standards. This is ideal in theory, but in practice, developers can forget to configure pre-commit hooks or bypass them, and that can allow noncompliant code to enter the repository. Running pre-commit checks as part of the CI pipeline ensures that coding standards are enforced consistently across the team, even when individual developers' environments are misconfigured or incomplete.

Testing

> This is another essential component of automated quality assurance. Unit tests verify that individual components and functions are behaving as expected, while integration tests ensure that different parts of the system work correctly together. Together, these tests reduce the risk of failures when you're updating models, data pipelines, or serving infrastructure. We'll discuss testing in Chapter 5.

Code reviews

> Human-run code reviews are an important safeguard, but in practice, teams are often overloaded with PRs, and that makes thorough human review harder to sustain. LLM-based code review tools help close this gap by automatically surfacing issues, suggesting improvements, and flagging

> potential risks. Rather than replacing human reviewers, they act as a first-pass filter that reduces the review burden on humans and allows engineers to focus on higher-level concerns like design, correctness, and readability. As their adoption grows, these tools will become a standard complement to automated checks and human judgment.

Documentation and code quality come down to people and processes. However, MLOps principles depend heavily on tools. But MLOps isn't about which tools you pick; it's about how you use them. So next, we'll look at MLOps components, walk through some tool examples, and see how all of this maps onto the Databricks platform.

MLOps Components

You may be familiar with the Machine Learning, Analytics, and Data (MAD) landscape, which is an initiative by Matt Turck that visualizes the evolving data and AI tooling ecosystem. The first version, released in 2012, featured just 139 companies. By 2024, the landscape had expanded to more than 2,000 companies, reflecting the rapid growth of the space.

In 2025, however, the industry saw a significant consolidation, with the number of companies dropping to around 1,150. Figure 1-3 highlights a subset of tools that focus on ML and AI, as the full landscape is too dense to be meaningfully read in a book format, and even this subset contains a large number of logos.

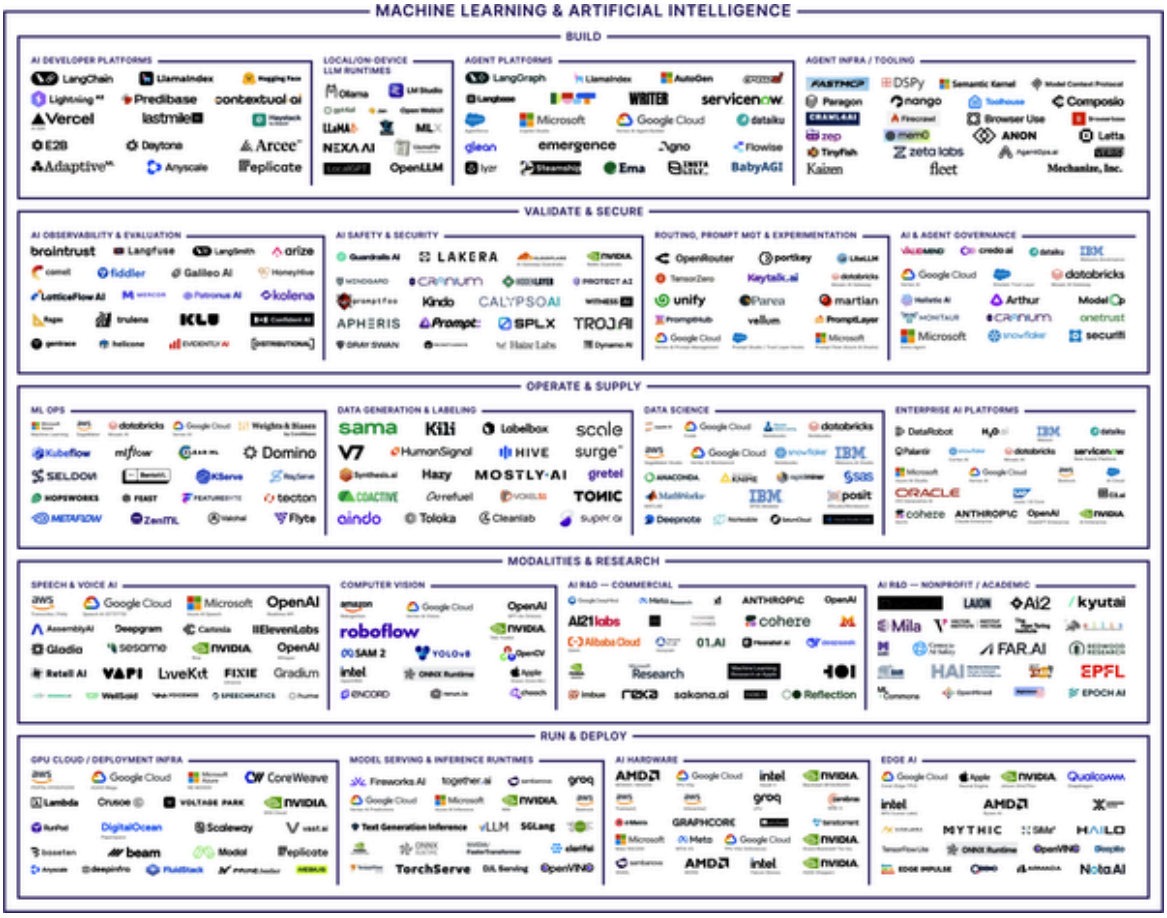


Figure 1-3. A subset of the [MAD landscape](#) focusing on ML and AI, as of 2025

When you see this many tools, it's hard to know where to start. You don't need all of them. What you need are certain types of tools, and the specific choice within each type depends on the complexity of your system. Many of these categories already exist in your DevOps stack; MLOps extends them rather than replacing them.

Each component exists to support one or more of the main MLOps principles: transparency, reliability, traceability and reproducibility, and observability. We will get to those principles after the tooling overview. First, I'll walk through the core component categories (which are summarized in Figure 1-4) and then cover what changes when your application is built on LLMs.

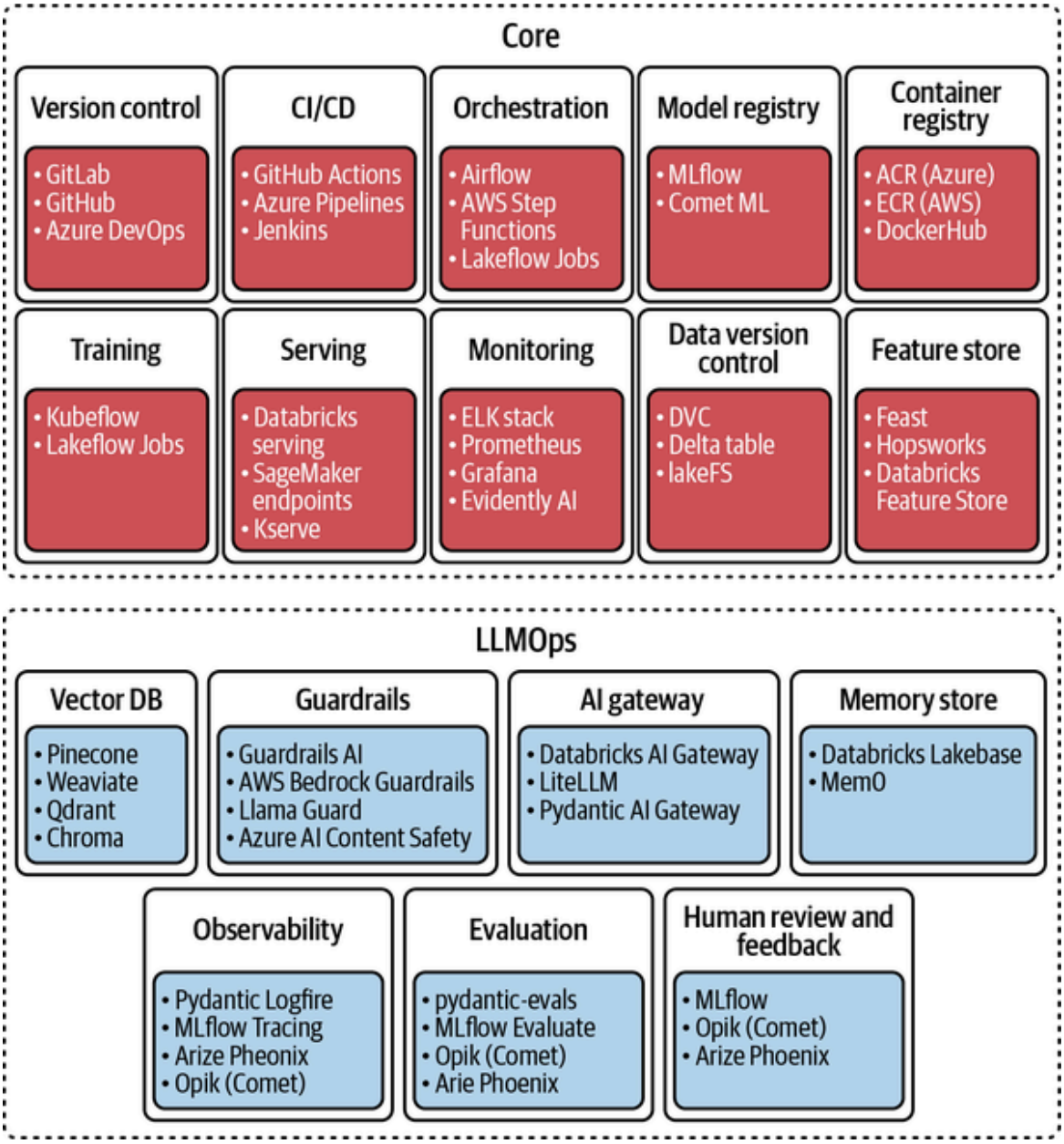


Figure 1-4. Core MLOps and LLMOps component categories spanning the ML/AI lifecycle, from code version control to monitoring

Core Components

The core components shown in Figure 1-4 are universal. Whether you're deploying a classical ML model or an LLM-based application, you'll need some form of each core component. Most of them will look familiar if you've worked in software engineering; the difference is in how they're

configured and what flows through them. In this section, I'll walk you through the core components

Code version control

Code version control is the practice of managing and tracking changes to source code and configuration files over time by using a version control system such as Git. It ensures the traceability and reproducibility of an ML model deployment or run. Without it, you'll have a hard time determining which exact version of the code was responsible for a particular training run, model artifact, or production error.

Platforms such as GitHub, Azure DevOps, and GitLab provide hosted environments built around Git that enable teams to collaborate on code more effectively. If you work in a corporate organization, you likely have such a platform (or even more than one) in place already. Beyond storing and managing repositories, these platforms offer features such as PRs and branch protection rules, which help enforce code quality by requiring reviews before changes can be merged into branches that allow deployment to production.

Continuous integration and continuous deployment (CI/CD)

CI/CD pipelines are automated workflows that build, test, and deploy code to production efficiently and with minimal manual intervention. In MLOps, they serve the same role as in DevOps, but what flows through them extends beyond application code to include training pipelines and serving components.

CI pipelines automatically build and test code changes whenever a PR is created, to validate that the application is still functioning as expected. No code should reach production without passing through this gate. *CD pipelines* then handle deployment, to ensure that production releases happen through a controlled, repeatable process, rather than through manual steps that are prone to error.

Most version control platforms provide built-in CI/CD solutions such as GitHub Actions, GitLab CI/CD, and Azure Pipelines. In this book, we use

GitHub Actions. We'll discuss this in detail in Chapter 5.

Workflow orchestration

Typical ML projects require *workflow orchestration* to manage the complex dependencies among different tasks, such as data preprocessing, feature engineering, and ML model training (which can also be parallel tasks when the model is trained separately by category). A workflow is often created for inference because inference requires a different frequency than training. This can be either *batch inference*, in which predictions get generated and written to a database or sent over SFTP, or *real-time inference*, which means we need a workflow for endpoint deployment.

Workflow orchestrators support various triggers, such as the CRON trigger, the file arrival trigger, and the conditional trigger (which runs a job if another workflow is finished successfully).

You can use tools like Airflow, Lakeflow Jobs, Azure Data Factory, and AWS Step Functions for workflow orchestration. If your organization does not provide a workflow orchestration tool as a service, it can be a lot of work to support such a tool on your own (mainly because of security patching). Therefore, it may be wise to choose managed services, and there are always some options available if your organization uses a cloud provider.

Model and container registry

Another important aspect of MLOps is the *model registry*. This includes storing and versioning trained ML model artifacts together with additional model metadata (hyperparameters, performance metrics, etc.). You should also store technical metadata, including the code version (git commit hash) you used to train the model and references to the data you used for model training. This is essential for traceability: you will always be able to pinpoint what went wrong and when and then roll it back when you need to.

In addition to storing ML models, you may need to store Docker images as part of your MLOps setup. For example, you could use Kubernetes to run model training jobs or to serve models, you may need to provide custom

Docker images to Amazon SageMaker/Azure Machine Learning (Azure ML), or you could use Docker images to run unit/integration tests in a CI/CD system. In all these scenarios, you'll need a tool to store your images, such as Azure Container Registry, JFrog Artifactory, or Amazon Elastic Container Registry.

Model training

Model training requires dedicated compute. The options depend on your setup: cloud-native services such as Azure ML, Amazon SageMaker AI Kubeflow for Kubernetes-native ML pipelines; or Lakeflow Jobs on Databricks.

Services like Lakeflow Jobs on Databricks combine orchestration and compute provisioning in one place, which allows you to define task dependencies and schedule training runs without managing the underlying infrastructure separately. In this book, we use Databricks compute for model training, but the principles apply regardless of the platform you use.

Model Serving

Serving and training do not always need the same infrastructure. *Training* is a batch workload that runs periodically, while *serving* handles real-time requests and needs to be always available. The requirements for compute, scaling, and networking are fundamentally different in these two processes.

For serving, Azure ML, SageMaker, and Databricks all provide managed solutions that simplify model deployment significantly. That said, many organizations, especially those with high Kubernetes adoption or specific latency requirements, prefer to deploy models on Kubernetes using frameworks like KServe. This can be practical when other services such as the frontend already run on the same cluster and you want to minimize network overhead between them.

Monitoring

Monitoring is an essential (but sadly often ignored) piece of an MLOps setup. Monitoring in ML systems goes beyond the standard processes for

monitoring regular software applications. The distinction lies in the fact that model predictions can be systematically wrong at scale, even when all typical health metrics appear to be in good condition.

Model performance should be checked regularly to prevent unexpected predictions from being consumed by end users. Tools like Azure ML, SageMaker, and Databricks Data Quality Monitoring provide features to monitor data and model drift. However, it's ideal to have all metrics in one place and use the monitoring systems already adopted by other teams. The ELK stack and a Prometheus + Grafana setup are options that you may already have.

Data version control

Your company's compliance policies may require training data to be reproducible, and that's hard to achieve without data versioning. Simply rerunning preprocessing scripts with a certain filter may not work and may erroneously include data that arrived later and was not included in the original training set, unless the scripts are designed in a specific way (which may not be straightforward).

On the other hand, data versioning tools allow you to re-create a training set exactly as it was at a given time. Tools like DVC, lakeFS, and the time travel functionality in Delta Lake help you tackle this problem.

Feature store

In general, a *feature store* is a data layer that connects data preprocessing (feature) pipelines, model training pipelines, and inference, to ensure training-inference alignment. (On the other hand, *Databricks Feature Store* is a specific product that serves as a centralized repository for ML model features; see the "Feature Engineering" section later in this chapter for details.) A feature store is useful for governance and lineage purposes (to determine what features were used for what models), and it allows for feature reusability across models.

However, not every application needs a feature store. For batch applications or real-time Model Serving that relies on precomputed features, you can

achieve governance and lineage with data version control tools alone. On the other hand, having a feature store is essential in real-time systems that require on-demand feature computation or lookup with millisecond-level latency and in scenarios where you need point-in-time correctness to ensure that training data reflects exactly what was known at the time of each prediction, without leaking future information. Fraud detection is a typical example that requires both point-in-time correctness and feature computation or lookup with ms-level latency.

LLM Application Components

The core components we covered in the previous section are universal, but when you're building LLM-based applications, several additional components come into play. Some of these, like vector databases, are not exclusive to LLMs but have gained significant traction because of them. We'll cover these additional components in this section.

Vector databases

Vector databases store and retrieve embeddings, numerical representations of text, images, and other unstructured data. In LLM applications, they are most commonly used for retrieval-augmented generation (RAG), in which relevant documents are retrieved and injected into the prompt to ground the model's response in internal knowledge. They are not limited to LLMs, though; recommender systems have relied on vector similarity search for years.

Tools like Pinecone, Weaviate, Qdrant, and Databricks AI Search (formerly Databricks Vector Search) make different tradeoffs when it comes to deployment, integration, filtering capabilities, and scalability.[12] Some traditional databases like PostgreSQL (with pgvector) also support vector search, which can be practical when you want to avoid introducing a separate system. On Databricks, Lakebase offers managed PostgreSQL with pgvector support, which can be a more cost-efficient alternative to Databricks AI Search for workloads that don't need a dedicated always-on endpoint.

Guardrails

Guardrails validate model inputs and outputs at inference time to enforce safety, compliance, and quality standards. They can prevent harmful or off-topic responses, enforce structured output formats, check for personally identifiable information (PII), and ensure that the model's behavior stays within business rules and domain-specific constraints.

Conceptually, guardrails are closely related to evaluation: they are evaluation checks that run in the production path, rather than offline. The reason they exist as a separate component is that certain controls, like PII detection and safety filtering, need to be enforced consistently at the platform level rather than left to individual developers to implement separately in each application.

The tooling splits into two categories:

- Managed platforms like Amazon Bedrock Guardrails, Azure AI Content Safety, and Databricks AI Guardrails intercept requests and block noncompliant inputs or outputs before they reach the user. The tools in this category enforce controls by default, regardless of what the application code does.
- Tools like Guardrails AI, OpenAI's Moderation API, and models like Llama Guard provide building blocks for implementing similar checks as part of the serving layer.[13] for details. In this category, enforcement becomes the responsibility of the application or platform team. It must either integrate checks into the application code directly or build a shared serving layer that enforces them centrally.

AI gateways

An *AI gateway* sits between your application and one or more LLM providers. It handles routing, fallback logic, rate limiting, API key management, and cost tracking among models and endpoints. This becomes important as soon as you start using more than one model or provider or when you need visibility into token consumption and spend.

Databricks Unity AI Gateway integrates natively with the Databricks platform. LiteLLM is an open-source, self-hosted alternative. Pydantic AI Gateway provides cost control and routing with an open-source core (AGPL-3.0). It can be managed through Pydantic Logfire or self-hosted for enterprise.

Memory stores

LLMs are stateless by default. Every API call is a blank slate with no knowledge of previous interactions. A *memory store* solves this problem by persisting conversation history, user context, and accumulated knowledge across sessions, so the application can behave as if it remembers previous sessions.

Databricks Lakebase provides a managed PostgreSQL backend for storing agent state, conversation history, and vector-based context. Mem0 takes a different approach with a three-tier memory model (user, session, and agent scopes) backed by a hybrid of vector, graph, and key-value stores. Which one you should use depends on whether you need infrastructure-level storage or a higher-level memory extraction and recall system.

Observability tools

LLM *observability* goes beyond traditional application monitoring. You need to trace full execution paths through your application: API calls to LLM providers, tool invocations, document retrieval steps, and chain-of-thought reasoning. Without this knowledge, debugging why an agent gave a wrong answer or took an unexpected action becomes guesswork. And while we've listed them under LLM application components, LLM observability tools can be equally useful for traditional ML serving with complex inference logic.

Major tools like Pydantic Logfire, MLflow Tracing, Opik (from Comet), and Arize Phoenix are all built on OpenTelemetry, which means any of them can trace your full application stack, not just the AI layer. And their core tracing capabilities are converging. The main differentiator is ecosystem integration: MLflow Tracing connects to Databricks Unity

Catalog for governance, lineage, and access control, while Logfire ties into the Pydantic stack. MLflow, Opik, and Arize Phoenix are open-source and can be self-hosted. Pydantic Logfire is available as a managed service or can be self-hosted with an enterprise license.

Evaluation tools

Evaluating LLM outputs is fundamentally harder than evaluating traditional ML models. There is no single accuracy metric. Outputs are open ended, context dependent, and often subjective. Evaluation in LLMOps combines automated metrics (LLM based or code based) with human scoring to build confidence that the system is behaving as intended.

Tools like Pydantic Evals, MLflow Evaluate, and Opik (by Comet) all support LLM-as-judge and custom metrics. Pydantic Evals integrates with the Pydantic ecosystem and Logfire. MLflow Evaluate ties into the Databricks platform. Opik is open source and can be self-hosted.

Human review and feedback

Automated evaluation provides scale, but human judgment remains the ground truth. Reviewers inspect model outputs, approve or reject agent actions, and provide corrections that feed back into improving the system. This is especially relevant for agentic applications in which the model takes actions (making API calls, performing database writes, and sending messages) that are difficult or impossible to reverse.

Most tools we've mentioned in the observability and evaluation sections also support human review as a core capability. MLflow, Opik, and Arize Phoenix all provide trace annotation and human scoring workflows. The boundaries between observability, evaluation, and human review are blurring: in practice, a reviewer looks at a trace, and scores it, and that score becomes both an evaluation data point and a feedback signal. The tooling reflects this convergence.

Choosing Your Tools

We've covered a lot of ground in this chapter: principles, team practices, and tools across both the core and LLM-specific components. The landscape is broad, and it's easy to feel overwhelmed. So how do you decide which tools to adopt?

Start with what your organization already has. The goal isn't to chase every new tool but rather to avoid unnecessary complexity in an already crowded space. Only introduce something new when there's a clear gap that your current stack can't fill. This matters more than it sounds. Every tool comes with onboarding, maintenance, integration work, and eventually migration. The ecosystem moves fast. What used to shift every few years now shifts every few months, driven by the rapid adoption of AI. The more tools you take on, the harder it becomes to adapt when things change again, and they will. A lean, intentional stack is easier to evolve than one built on every promising option that comes along.

This is also where platforms start to make a strong case for themselves. Rather than stitching together separate tools for each component, a platform like Databricks consolidates many of them under one roof. That doesn't mean it replaces everything, but it reduces the number of moving parts you need to manage independently. In the next section, we'll walk through how the Databricks platform maps to the components we've discussed and what the transition from general tooling to a platform-native approach looks like.

Databricks MLOps Components

Databricks provides all the features you need for MLOps. Table 1-4 maps the MLOps and LLMOps landscape component categories from Figure 1-4 (earlier in the chapter) to their Databricks equivalents.

Table 1-4. Mapping MLOps and LLMOps component categories to Databricks

MLOps and LLMOps component categories	Databricks equivalents
Workflow orchestration	Lakeflow Jobs
Model registry	MLflow and Unity Catalog
Model training	Lakeflow Jobs (using Databricks compute)
Serving	Model Serving, Feature Serving, Foundation Models
Monitoring	Data Quality Monitoring
Data version control	Delta tables
Feature stores	Feature Engineering in Unity Catalog
Vector DB	Databricks AI Search
Guardrails	Databricks AI Guardrails
AI gateways	Databricks Unity AI Gateway
Memory stores	Lakebase
Observability tools	MLflow Tracing
Evaluation tools	MLflow Evaluate
Human review and feedback	MLflow

In the next chapters, we will go through all of these components in detail and show you how to combine them in a way that promotes MLOps principles. In the remainder of this section, we'll cover the components from Table 1-4 in more detail and then discuss cross-cutting components that don't map to a single category.

Lakeflow Jobs

Lakeflow Jobs is an orchestration system that's integrated into the Databricks platform. There are three main concepts related to Lakeflow Jobs that you need to know about:

Jobs

A job is a primary unit of orchestration that supports various triggers.

Triggers

These include the CRON trigger, the file arrival trigger, table updates, the job dependency trigger (which runs the job when another job is completed), and the API trigger (which starts a job on API call).

Tasks

Databricks jobs consist of tasks that support various complex scenarios. Each task can perform a variety of operations. Here are some that are commonly used for ML workflows:

- A Databricks notebook with Python code and/or SQL queries
- A Python script or a wheel
- Lakeflow Spark Declarative Pipelines (formerly Delta

Live Tables) for Extract, Load, Transform (ELT) pipelines

- If/else conditions that can be handy if, for example, the workflow must run only on certain days or certain hours and the logic goes beyond what CRON definition supports
- For-each loops that can be useful when you need to loop over the same task with different parameters and these sub-tasks must be executed in parallel

Databricks is constantly improving the Lakeflow Jobs functionality, and new features, including new types of tasks, are released frequently.

Databricks Compute

You can specify the compute that will be used to run each of the tasks of a Lakeflow job. For some tasks (for example, a Python task), you can choose either a serverless compute or a classic compute.

Classic compute versus serverless compute

Classic compute is configured by the users, and it runs within the customer's cloud environment. The costs for classic compute are distributed between Databricks and the cloud provider.

For a classic compute, users have all the freedom to choose the configuration of the following:

- The *runtime* (the environment used to execute the code) that defines the Python version, Spark version, and OS version
- Pre-installed libraries
- The worker and driver type
- The autoscaling policy
- Advanced settings (for example, init scripts or Docker images)

Serverless compute means that the compute infrastructure is fully managed and optimized by Databricks. In this setup, there are no underlying infrastructure costs billed through the customer's cloud provider. Serverless compute also often delivers a better price/performance ratio than classic, customer-managed clusters. In addition, serverless workloads start almost instantly, so users don't have to wait for clusters to spin up. While users have limited control over low-level compute settings and runtime configuration, they can still manage Python versions and libraries through the environment settings provided with serverless compute.

Serverless compute works well for development, data engineering, and AI engineering workloads. However, it is not always suitable for certain ML model training tasks. Although Databricks is actively investing in its serverless offering—and the list of limitations is shrinking rapidly—some constraints remain.

One of the main current limitations on serverless compute is the maximum available size of the driver node. Many Python-based ML training workflows rely on libraries such as scikit-learn and the Light Gradient-Boosting Machine (LightGBM), which typically operate on in-memory data structures such as Pandas or Polars DataFrames. These libraries are not distributed, and they execute computations on the driver node. As a result, the maximum available driver size in a serverless environment may not always be sufficient for memory-intensive training workloads.

Traditional ML training workflows often still require classic compute for greater flexibility and resource control. In contrast, many modern AI workloads can run efficiently on serverless infrastructure. Since we focus on both traditional ML and AI applications in this book, we'll use both types of compute and discuss implications for development.

All-purpose versus job compute

For a task in a Lakeflow job that utilizes standard compute, you can choose between all-purpose (often named as interactive) compute and job compute. Overall, you should use a job compute for Lakeflow jobs and all-purpose clusters for development. While the costs associated with your cloud

provider will be the same, the Databricks costs, measured in Databricks Units (DBUs), will be significantly lower.

MLflow

MLflow is one of the oldest and most popular tools for experiment tracking and model registry. MLflow is an open-source tool that was originally created by Databricks, and it is also available as part of the Databricks data and AI platform.

Experiment tracking is managed at the Databricks workspace level. It allows you to store parameters, metrics, and metadata that belong to an experiment run. A model must be logged using supported flavors in the experiment tracking system before it can be registered.

While the model registry was previously also at the workspace level (which is now deprecated), the recommended approach today is to register models in the Unity Catalog using the MLflow library.

Given that organizations increasingly build AI applications powered by large language models, the *MLflow Tracing* product has been introduced as a key component for ensuring traceability and reproducibility. It integrates with many GenAI libraries and helps developers understand and debug what happens below the abstraction layer. We'll talk about MLflow Tracing in detail in Chapter 8.

Feature Engineering

Databricks Feature Store is a centralized repository for storing and sharing ML model features. Workspace Feature Store is marked as legacy and has been deprecated, and Databricks recommends Feature Engineering in Unity Catalog for handling model features.

Databricks provides a Python API to interact with the Databricks Feature Engineering in Unity Catalog. It is a key component that you can use to do the following:

- Create, update, and delete feature tables and feature specifications

- Create a training dataset
- Log an MLflow model packaged with information on how a training set was built and then use it for batch scoring

Feature engineering functionality is especially useful when out-of-the-box automatic feature lookup is required for real-time model inference.

Databricks Model Serving

Databricks provides Model and Feature Serving functionality, which can significantly simplify the deployment.

Databricks Model Serving is designed for real-time model inference. It requires you to register a model in Unity Catalog using MLflow, and after that, you can deploy the model behind a serving endpoint using MLflow's serving capabilities.

You can also use Databricks Model Serving to access proprietary foundation models from other providers such as Azure OpenAI, Amazon Bedrock, and Anthropic, as well as to serve (fine-tuned) open source foundation models and agents. Databricks-hosted pay-to-go foundation models are also available through the platform. We'll talk about these options in detail in Chapter 7.

Databricks Model Serving for ML applications integrates with feature engineering functionality to enable automatic feature lookup from tables stored in the Databricks Online Feature Stores. These stores are built on *Lakebase*—a fully managed PostgreSQL-based online transaction processing (OLTP) database engine—and they contain read-only copies of feature tables that are optimized for low-latency access. The tables are synchronized from Delta tables continuously or based on a trigger. The lookup logic is predefined within the feature engineering pipeline, so features can be retrieved automatically during inference. Lakebase is used in different parts of the Databricks ecosystem beyond Feature Serving, and we discuss it in the section "Cross-Cutting Components".

If you need more flexibility, you can implement feature retrieval logic directly within the model definition. In such cases, users are responsible for managing authentication and access to the underlying data sources themselves. We will implement an example of this approach in Chapter 9, where we will use Lakebase as memory for an AI agent.

Databricks Feature Serving is a great fit when you require a batch serving architecture. Precomputed features are stored in a feature table in Unity Catalog. A table in Databricks Online Feature Stores and a feature specification (FeatureSpec, a user-defined set of features and functions) are used behind the Feature Serving endpoint.

Data Quality Monitoring

Data Quality Monitoring helps you monitor data drift and detect model performance deterioration if the ground truth is provided. When you enable Data Quality Monitoring on a table, the profile metrics table, the drift metrics table, and a customizable dashboard are created out-of-the-box.

Data Quality Monitoring can be applied and incrementally updated on time series data or inference data (e.g., data containing information about payload for a serving endpoint). You can also use the third type of supported profile, a snapshot, for all other types of tables. The data quality metrics will then be calculated for all data in the table, and all the data in the table will be processed with every refresh.

It is possible to set alerting on the profile or drift metrics table and get notified in case a certain drift metric exceeds a predefined threshold.

Databricks AI Search

The Databricks AI Search functionality in Databricks allows you to create an AI Search index from a Delta table. To use an AI Search index, you need to create an AI Search endpoint first. Just like a Model Serving endpoint, it uses serverless compute. You can query the index by using a representational state transfer API (REST API) or the Databricks vector search Python client.

Like other vector databases, AI Search on Databricks uses the hierarchical navigable small world (HNSW) algorithm for its approximate nearest neighbor search, and it supports hybrid keyword similarity search.

Like an online table, the AI Search index can be updated automatically every time there is an update in a delta table.[14] Alternatively, it can be updated based on a trigger.

Unity AI Gateway

Unity AI Gateway is the central AI governance layer on Databricks for agents, LLM endpoints, model context protocol (MCP) servers, and coding agents. It provides a unified API across models from different providers—including Anthropic, OpenAI, Google, and open-source models—with consistent policy application. This eliminates the need for organizations to duplicate configurations for each provider.

Unity AI Gateway allows organizations to analyze usage, configure permissions, enforce guardrails, and manage capacity across providers. End-to-end observability is built in: you can monitor LLM and MCP calls, track usage, and attribute costs across models, teams, and workflows.

Databricks AI Guardrails

The AI Guardrails in Unity AI Gateway allow organizations to enforce safety, compliance, and business policies on model inputs and outputs. Guardrails can help prevent harmful or offensive content, detect or block sensitive information, keep conversations on topic, and ensure that responses align with organizational or branding requirements.

Databricks provides built-in guardrails for PII redaction and blocking, unsafe content filtering, jailbreak detection, and hallucination detection, along with support for custom prompt-based guardrails that are tailored to organization-specific policies. We will discuss guardrail configuration as part of the AI Gateway setup in Chapter 9.

Cross-Cutting Components

Each of the components we've described so far maps to one of the specific component categories in the MLOps and LLMOps landscape that we summarized in Table 1-4). However, several Databricks components play a different role: they cut across multiple categories, or they provide infrastructure that enables other components to work together. These components don't fit neatly into a single row in the table, but they play an important role in how the platform functions as a whole. We cover these cross-cutting components in this section.

Unity Catalog

Unity Catalog is a governance layer for data and AI that is built into the Databricks platform and was open-sourced in 2024. It provides centralized access control, lineage tracking, auditing, and discoverability for data and AI assets across Databricks workspaces. These assets include ML models, functions, Delta tables, views, and files stored in Databricks Volumes.

When enabled, Unity Catalog operates transparently beneath every data interaction. It enforces permissions when tables are queried, tracks lineage as data moves through pipelines, and provides centralized governance and auditability across data and AI assets.

Rather than fitting into a single component of the MLOps stack, Unity Catalog underpins many core capabilities across the platform, including model registry, feature engineering, data versioning, serving, and AI governance. Its introduction was one of the most significant changes in the Databricks ecosystem because it substantially simplified the implementation of production-grade MLOps practices.

Declarative Automation Bundles

Declarative Automation Bundles (DABs) are the "glue" that helps to deploy Databricks resources, such as Lakeflow jobs, with all the dependencies.

In general, there are several ways of deploying Lakeflow jobs without DABs: you could use Terraform or Databricks APIs, for example, but then,

you would need to take care of a lot of additional components. For example, if you wanted to execute a Python script (let's call it main.py) that requires a *project_config.yml* file, you would have to ensure that all files were uploaded to the workspace or Unity Catalog Volumes before deploying the job, and you would also have to verify that the location of those files was referenced in the job definition.

Declarative Automation Bundles help you manage these dependencies. It uploads all the files that belong to the project to a path in the workspace, which users can then define. It is a recommended and fully supported way to deploy jobs and other resources on Databricks.

We discuss Declarative Automation Bundles in detail in Chapter 5.

Lakebase

Lakebase is a fully managed Postgres database that's built into the Databricks platform. Databricks acquired the technology behind Lakebase through its acquisition of Neon,[15] a serverless Postgres company, and it thereby brought a production-grade serverless database engine into the platform. Lakebase serves multiple roles across the Databricks ecosystem: it powers the Online Feature Stores with read-only copies of feature tables that are optimized for low-latency access, and it can be used as a memory store for AI agents, which is an approach we discuss in Chapter 7.

Lakebase also supports several capabilities that make it well suited for operational AI workloads. It scales to zero when not in use, which means you won't pay for idle infrastructure. That's particularly relevant for development environments or workloads with variable traffic patterns. It also lets you restore databases from a previous state, so it provides a safety net for operational workloads. Because Lakebase is fully managed, there is no infrastructure to provision or maintain; compute and storage are handled by the platform.

Before the introduction of Lakebase, the Databricks platform had a gap: there was no native solution for the low-latency, transactional access patterns required for online feature serving and agent memory.

Organizations had to rely on external databases for these use cases, and that added operational complexity and broke the governance boundary.

Databricks Apps

Databricks Apps provide a hosting environment for custom applications built on Databricks serverless infrastructure. As Databricks standardizes AI application and agent deployment around Apps, they are becoming an important component of production AI architectures. Databricks Apps simplify deployment by providing built-in open authorization (OAuth) support, permissions management, and governance integration out of the box.

Databricks Apps are also the recommended way to host custom Model Context Protocol (MCP) servers. While Databricks provides managed MCP servers for its own platform capabilities, organizations often need to expose internal APIs, databases, or custom business logic as tools that agents can call.

Even though the code examples in this book do not utilize Databricks Apps directly, they are becoming an increasingly important part of the Databricks AI ecosystem and are likely to play a central role in how production AI systems are deployed and integrated going forward.

Conclusion

In this chapter, we walked through the MLOps principles and components, plus the tools within the Databricks ecosystem that are useful for MLOps. One of the pitfalls I have seen teams fall into is taking a tools-first approach without having a clear understanding why they need the tool and how to use it to support the principles. I wouldn’t recommend adding any tool (especially if it means extra costs and maintenance effort) unless it simplifies the development lifecycle or helps to make your system more reliable.

Next, before we proceed with practical implementation, I will walk you through the different developer tools that are available on Databricks and how you can use them to speed up the development cycle.

---

1 D. Sculley et al. "Hidden Technical Debt in Machine Learning Systems," *Advances in Neural Information Processing Systems*, 2015.

2 Read more about Uber's Michelangelo in the company's engineering blog.

3 Read more about scaling ML productivity at LinkedIn in the company's engineering blog.

4 Read more about Netflix's ML platform in the company's blog.

5 SphereOI, *How Zillow Could Have Avoided Its $500M AI Mistake*, September 1, 2022, *https://oreil.ly/ES7-w*.

6 For more information about the framework, consult the Linux Foundation's AI and data blog announcement of their principles for trusted AI.

7 One source of nondeterminism in deep learning inference is floating-point nonassociativity: because floating-point addition is not perfectly associative, graphics processing unit (GPU) operations that reduce large arrays can produce different results depending on accumulation order, which varies with batch size and parallelization strategy. This is a solvable engineering problem. Thinking Machines Lab has demonstrated that batch-invariant kernel implementations can achieve fully deterministic LLM inference, though at a performance cost. See Horace He and Thinking Machines Lab's "Defeating Nondeterminism in LLM Inference" for more information. For a practical walkthrough, see also Austin Zaccor's "On the Topic of LLMs and Non-Determinism".

8 S. McConnell, *Code Complete*, 2nd ed. (Cisco Press, 2004), ch. 22, "Developer Testing."

9 The handbook is available on Google's dedicated SRE site.

10 For more details, consult the CodeRabbit State of AI vs. Human Code Generation Report.

11 For more details, read the full DORA report.

12 Superlinked provides an extensive comparison of different vector databases.

13 Check out the Llama Guard 4 model card

14 Continuous sync mode is not supported for storage optimized endpoints.

15 For more details, check out Databricks' announcement.

Chapter 2. Developing on Databricks

Databricks has grown significantly in popularity over the past few years, yet many data scientists and ML engineers still struggle to find an effective way to develop their code within the platform.

Notebooks are first-class citizens in Databricks and are widely used by data scientists during the experimentation phase. They are excellent tools for iterative development, and they have helped popularize interactive Python workflows. However, notebooks were not designed to promote best software engineering practices, and code written during experimentation often requires substantial refactoring before it can be deployed to production.

This refactoring typically involves the following:

- Creating functions, classes, and modules
- Separating tasks into logical components
- Isolating configurations that may evolve over time
- Adding unit tests
- Implementing logging and documentation
- Packaging the project’s code

Throughout my career, I have often seen data scientists rely on ML engineers to carry out these steps. This dependency often slows down the path to production, even though much of this rework could be avoided if code were written with production in mind from the outset.

I believe data scientists should be empowered to write production-ready code themselves (while they still can use notebooks), and one of the main

goals of this book is to demonstrate how to do this. Despite Databricks' growing adoption, there is still a lack of best practices tailored to its unique development environment.

Developing on Databricks relies heavily on clusters, and you need to understand the impact of cluster selection. Therefore, before diving into development workflows, let's briefly review the different types of clusters that are available in Databricks.

Cluster Configurations on Databricks

I'll start by showing you how to set up an all-purpose cluster for connecting from a local environment or running notebooks within the Databricks workspace. Then, I'll walk you through the serverless compute option, which allows anyone to run the code using Databricks Free Edition. All code examples are compatible with both cluster types. In Chapter 1, I explained when and why you might choose serverless over classic compute and vice versa.

All-Purpose and Jobs Compute Configuration

Before we start developing on Databricks, we need to set up a cluster. All-purpose clusters are mainly used for development, while you should use jobs clusters to run Lakeflow jobs. Cluster configurations for both all-purpose and jobs clusters are similar, so I'll just go over the all-purpose cluster settings. You can set up an all-purpose cluster using an API or Terraform, but the easiest way to view the possible configurations of the cluster is via the user interface (UI) located in the compute tab.

Figure 2-1 shows the recommended configurations of the cluster you can follow through the code accompanying this book. The specific configuration settings you should set are as follows:

Policy

Set this to Personal Compute.

Databricks runtime

Set this to 17.3 LTS.

Preferred node type

Select 16 GB Memory, which is one of the smallest sufficient nodes. It is a single-node cluster without any autoscaling policy.

Terminate after _____ minutes of activity

Set this to 30 minutes to reduce costs.

Tags

Set whatever tags you'll need to make sure development costs can be tracked back to the project.

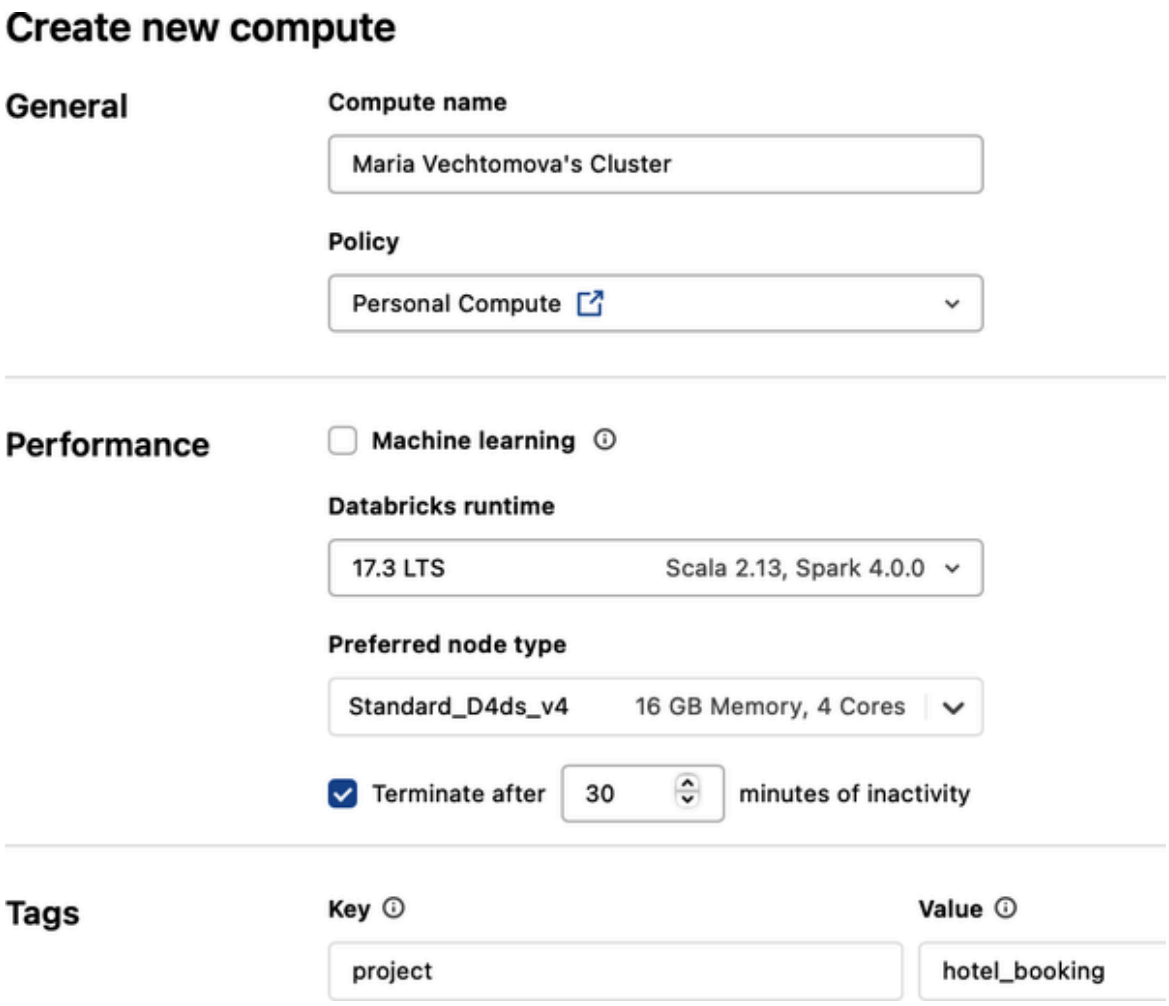


Figure 2-1. Recommended cluster configuration

Let's go through all the different components of the cluster configurations you can choose from.

Policies and access modes

When creating a cluster, you can choose from the following default policies (unless more policies are created by workspace admins):

Personal Compute

> This is ideal for creating single-node compute resources assigned to individual users with minimal setup.

Shared Compute

This is suitable for creating larger, multinode resources shared among users.

Power User Compute

This is an upgraded version of Personal Compute that allows for multinode setup and is designed for distributed single-user workloads.

Unrestricted

This allows full access to all compute configurations when setting up new compute resources.

By default, only workspace admins can create clusters with Shared, Power User, and Unrestricted compute policies. They can grant individual users or user groups access to compute policies. Workspace admins can also create dedicated compute policies, which can be handy if you need to pre-install libraries or restrict cluster creation.

You also need to pay attention to the *access modes*: Dedicated (formerly Single), Standard (formerly Shared), and No Isolation Shared. Different modes have different limitations. For example, you need to put init script locations and wheel locations in ALLOW_LIST to use them in standard access mode. Check out the documentation for details.

Overall, for ML workflows, I recommend using Dedicated access mode.

Runtime versions

Databricks Runtime is the set of software artifacts that run on Databricks clusters. It defines the environment where your code runs. The pyt version must match the Python version of your project (in our case, Python 3.12).

In this book, I am using Databricks Runtime 17.3 LTS, which is, at the moment of writing this book, the latest long-term support (LTS) runtime. For production, it is a common organizational practice to only use LTS runtimes: the support cycle for them is three years. Databricks provides an

overview of LTS runtimes and their end-of-support dates. For instance, the end-of-support date for Databricks Runtime 17.3 LTS is October 2028.

The runtime comes with a pre-set system environment. These are the specifications of the environment for the Databricks Runtime 17.3 LTS:

- Operating system: Ubuntu 24.04.3 LTS
- Java: Zulu17.58+21-CA
- Scala: 2.13.16
- Python: 3.12.3
- R: 4.4.2
- Delta Lake: 4.0.0

In addition to the pre-set environment, runtimes contain pre-installed Python libraries. That does not mean you have to use the exact versions of the libraries in the runtime. You can also install another version of the library you need, but keep in mind that you are not installing it in a fresh environment, which makes it challenging to reproduce the same environment locally.

Photon acceleration

Photon is Databricks' native vectorized query engine, and it's designed to accelerate Spark SQL workloads and DataFrame API calls. It replaces parts of the traditional Spark execution engine with highly optimized code that takes advantage of modern CPU architectures. While enabling Photon increases the DBU rate per hour, it often reduces total cost per workload because jobs complete significantly faster.

However, Photon only accelerates Spark operations and has no effect on single-node Python libraries. A typical ML training script that uses the scikit-learn library operates on Pandas or Polars DataFrames that are loaded into the driver's memory. Since these libraries bypass Spark entirely, enabling Photon provides no performance benefit for the training step itself.

My advice: enable Photon for data preprocessing tasks that use PySpark (joins, aggregations, and feature transformations at scale), then disable it for compute that runs single-node ML training. This gives you the speed benefits where they matter while avoiding unnecessary DBU costs during training.

Worker and driver nodes

In a single-node setup, you do not have an option to set up worker and driver nodes. You can only choose the size of the single node that acts as both master and worker node. If your task involves running Python code that does not require Spark's distributed computing capabilities, you need a single-node compute. You may also need a large single-node cluster if the data you are working with is large.

If your task involves large-scale data processing or distributed ML, you need a multinode setup. You can choose different worker and driver node sizes, depending on your task, and set autoscaling policies if you expect that the data you work with may increase in size. It is crucial to choose the right size for your nodes and the right amount of worker nodes, and you also need to choose proper autoscaling policies since they have a big impact on costs.

Advanced settings

In the advanced settings of the cluster, you can do the following:

- Set up environment variables (e.g., referencing secrets stored in a secret scope)
- Define init scripts
- Specify a Docker image to initialize your cluster
- Specify a logging location

Cost tracking

You can also track cluster-related costs by applying custom tags to a cluster. These tags are propagated to the underlying cloud provider, and you can use them for cost allocation and reporting. To obtain a complete view of infrastructure spend, you should combine Databricks billing data from system tables with cost data from the cloud provider.

Serverless Compute

When it comes to the all-purpose or jobs compute clusters, users have a lot of freedom to adjust configurations as they wish. But this comes with a cost: it will take time for them to find the perfect cluster configuration for their project (and even different cluster configurations for each task).

You can use serverless compute for both development purposes and running a Databricks job. Even though serverless compute may not be sufficient for machine learning model training pipelines, you could still use it for data preprocessing pipelines.

Unlike all-purpose or job compute, we cannot choose the runtime for serverless compute. Instead, we can curate an environment for dependencies. While the concept of an environment and a runtime are similar, environments are much more lightweight.

Serverless environments

In our project, we use Python 3.12, which means we also need to have an environment that supports that Python version. In this book, we use serverless environment version 4, which comes with the following configurations:

- Operating system: Ubuntu 24.04.2 LTS
- Python: 3.12.3
- Databricks Connect: 17
- Scala: 2.13.16

- JDK: 17

Similar to runtimes, serverless environments come with pre-installed libraries. However, the library versions differ slightly between serverless environment 4 and Databricks Runtime 17.3 LTS. This is something to watch out for if your workflows switch between serverless and classic compute, as subtle version mismatches can cause unexpected behavior. I recommend installing your own versions of the libraries you need, to ensure consistency across environments.

Like runtimes, environments have a support cycle of three years. The end-of-support date for environment 4 is in August 2028. You can check out the release notes for the full overview.

Cost tracking

While you can track the costs of classic compute clusters with tags, you would use usage policies for serverless compute. Serverless usage policies consist of tags that are applied to any serverless compute activity incurred by users assigned to the policy. These tags are recorded in system tables, so they let you attribute serverless usage to specific budgets.

Usage policies can be created programmatically, or you can create them through the UI (see Figure 2-2). You can also assign permissions to individual users or user groups (see Figure 2-3). It's also important to note that if you don't specify a usage policy ID when using serverless resources, the first available policy (in alphabetical order) assigned to the user will be automatically applied.

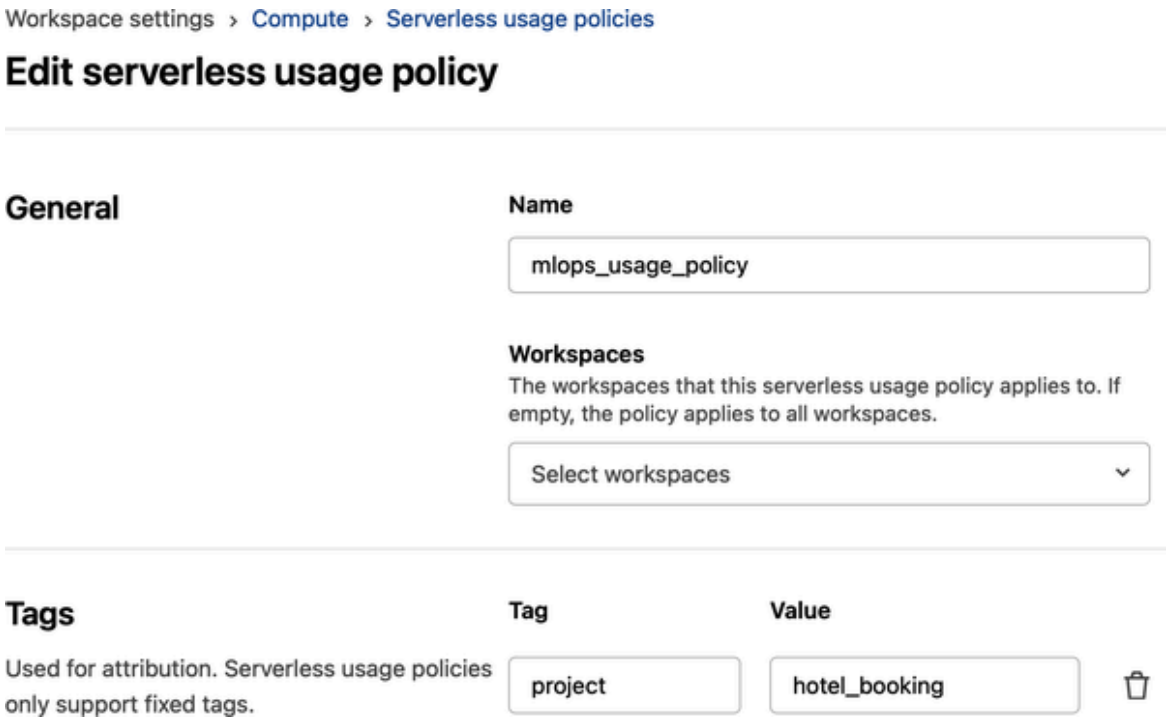


Figure 2-2. Creating a usage policy

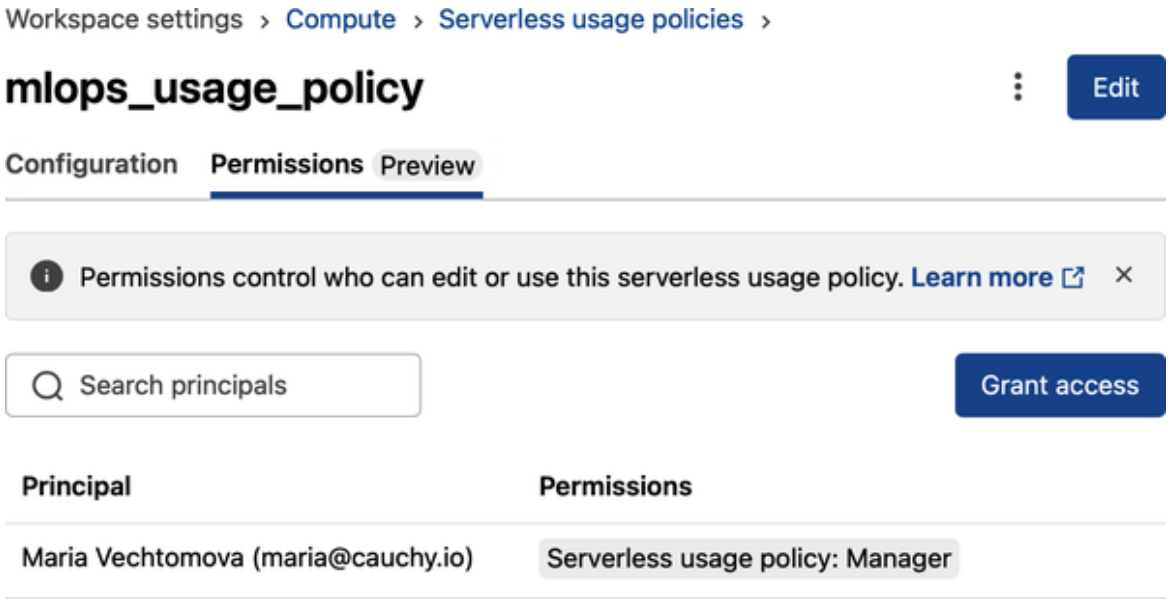


Figure 2-3. Updating permissions for the usage policy

Project Setup

In this section, I will walk you through the dataset used in the MLOps part of the book and the different ways you can execute the project's code.

The book repository can be found on GitHub at *https://github.com/mvechtomova/oreilly-mlops-with-databricks*. First of all, you should fork the repository so you can go through the next chapters. The repository contains the following folders: *mlops* (which covers Chapters 2 through 6) and *llmops* (which covers Chapters 7 through 9).

For now, we'll talk about the *mlops* folder, which is structured into the following subfolders and files:

notebooks/

: This folder contains notebooks used in different chapters.

scripts/

: This folder contains Python scripts that are executed as part of the Databricks job.

databricks.yml

: This is the file the Databricks job is defined in.

pyproject.toml

: This file contains the Python package definition for the project.

project_config.yml

: This file contains isolated configurations of the project, such as the location of the data, model hyperparameters, and model features. In other words, it captures the settings we expect to change during the project's lifecycle, to ensure that they remain flexible and not hard-coded into the codebase.

src

: This folder contains packaged code for the project.

data

This folder contains the required data for the project.

tests

This folder contains unit tests.

version.txt

This file contains a version number that is used for Python package versioning.

First, let's discuss the dataset in the `data` folder.

The Dataset

In Chapters 2 through 6, we use the Hotel Booking Cancellation Prediction dataset from Kaggle.[1] The dataset contains various features related to bookings, the number of week and weekend nights, the time between arrival and the reservation date, types of meals, whether guests requested parking, room types, and other features. We will use these features to predict the price of a room per night.

This is a dataset that is close to a real-life scenario. In Chapters 3 through 5, we will work on the following scenarios:

Model Serving

This is real-time inference in which all the features required for model prediction are provided by the user.

Model Serving with feature lookup

This is real-time inference in which some features come from the user and others are retrieved from a feature store. In this case, the model computes the predicted price per night in real time, using the full feature set.

Feature Serving

> This is real-time retrieval in which the predicted price per night is precomputed in a batch process and stored in a feature store. At serving time, the user only provides a booking ID, which is used to fetch the stored prediction.

In the *project_config.yml* file, I specified the target and features that will be used for model training. Our configuration file is relatively simple; for more complex projects with hierarchical configurations, it's worth checking out the Hydra project.

You'll also notice the `usage_policy_id` parameter. We created the usage policy in the section "Cost tracking", and I'm including it here so you can pass the usage policy when provisioning serverless resources (such as a Model Serving endpoint). This will allow you to properly attribute and track costs associated with those resources:

```
dev:
 catalog: mlops_dev
 schema: hotel_booking

acc:
 catalog: mlops_acc
 schema: hotel_booking

prd:
 catalog: mlops_prd
 schema: hotel_booking

parameters:
 learning_rate: 0.05
 n_estimators: 200
 max_depth: 3

num_features:
 - number_of_adults
 - number_of_children
 - number_of_weekend_nights
 - number_of_week_nights
 - car_parking_space
 - special_requests
 - lead_time
```

```
cat_features:
 - type_of_meal
 - room_type
 - arrival_month
 - market_segment_type

target: average_price

usage_policy_id: cdad6f29-a213-4525-9330-e25b90afbe7a
```

Next, the *project_config.yml* file contains model hyperparameters, plus catalogs and schemas used in different environments. You can also update this file to put in your own schema and catalog names if you wish. For now, we'll only focus on the development environment; we'll discuss other environments and deployment strategies in Chapter 5.

We will also get to hyperparameter tuning in Chapter 3, where we'll discuss how we can use MLflow to log all the metrics and parameters and how we got to the parameters specified in the file. It is a common approach to do periodic retraining with the same hyperparameters (as hyperparameter tuning is quite costly) and only perform fine-tuning when there are signs of performance degradation.

Python Tooling

Most people in the industry are convinced that we need to start packaging the code. But what is the right way? What tools should we use?

The Python ecosystem has changed significantly over the last several years, starting around 2016. Before that, everyone was using *setup.py* together with *requirements.txt*, plus the following set of tools:

venv or virtualenv

> They used these for virtual environment management. The venv tool is shipped with Python 3, and virtualenv is a package maintained by Python Packaging Authority (PyPa).

pip

They used this for package management.

setuptools

They used this for build backend.

twine

They used this for package publishing.

In data analytics and data science, *conda* was a popular choice. This was a result of the efforts of an organization called Anaconda that was founded in 2012 by Peter Wang and Travis Oliphant,[2] with the goal of bringing Python into business data analytics. The conda tool is a multipurpose tool that assists with package, virtual environment, and Python version management. It is worth mentioning that in 2024, Anaconda introduced changes in licensing that forced conda users to look for alternatives.

In 2016, the PEP 518 standard was introduced, and it specified that build system dependencies should be stored in a *pyproject.toml* file. In 2020, PEP 621 was introduced, and it defined how project metadata should be stored in the *pyproject.toml* file.

Between 2017 and 2020, multiple projects started using *pyproject.toml* to serve multiple purposes and try to eliminate the need to have so many tools. The best-known projects were hatch, poetry,[3] and pdm.

In 2023, Rust started taking over the Python ecosystem due to its speed, and another multipurpose tool called Rye was released. In February 2024, another Rust-based tool called uv developed by Astral entered the market, and it has won the hearts of the Python community since then. Astral also took over the development and maintenance of Rye.

The uv tool is, in my opinion, by far the most intuitive tool to use, and it is blazingly fast. It is therefore the tool I will be using in this book.[4] You should follow the official installation instructions to install uv.

The pyproject.toml file

Let's walk through the sections of the *pyproject.toml* file we will use for the MLOps project.

Project section

In the `project` section of the *pyproject.toml* file, we specify the Python version and all the project dependencies:

```
[project]
name = "hotel-booking"
dynamic = ["version"]
description = "Code accompanying MLOps with Databrick O'Reilly book"
requires-python = ">=3.12, <3.13"
dependencies = [
   "mlflow==3.11.1",
   "cffi==1.17.1",
   "cloudpickle==3.1.1",
   "numpy==2.4.0",
   "pandas==2.3.0",
   "pyarrow==22.0.0",
   "databricks-sdk==0.102.0",
   "scikit-learn==1.7.0",
   "lightgbm==4.6.0",
   "scipy==1.16.0",
   "pydantic==2.11.7",
   "loguru==0.7.3",
   "python-dotenv==1.1.1",
   "ray[tune]==2.52.1",
   "optuna==4.4.0",
   "sdv==1.24.0",
   "databricks-feature-engineering==0.14.0",
   "databricks-feature-lookup==1.11.0"
]

[project.optional-dependencies]
dev = ["databricks-connect>=17.3.2, <18",
      "ipykernel>=6.29.5, <7",
      "pip>=25.0.1, <26",
      "pre-commit>=4.1.0, <5",
      "pytest>=8.3.4, <9",
      "pytest-mock>=3.14.0, <4"]

ci = ["databricks-connect>=17.3.2, <18",
```

```
    "pre-commit>=4.1.0, <5",
    "pytest>=8.3.4, <9",
    "pytest-mock>=3.14.0, <4"]
```

First of all, it is important to note that we use Python 3.12, which matches the Python version of the Databricks Runtime 17.3 LTS runtime and environment version 4 for serverless.

Secondly, all the package dependencies are pinned to very specific versions. While this may seem beneficial at first glance, it is not considered best practice for packages that are intended to be reused across multiple projects, as strict version pinning can easily lead to dependency conflicts. So for reusable packages, it is generally recommended to define a supported range of Python versions and dependency versions instead. In this case, however, the Python package is not designed for reuse; it simply serves as a convenient way to define and manage project-level dependencies.

When you're installing the package, you don't need to manually specify all dependency versions listed in the *pyproject.toml* file. These dependencies are included in the package metadata and will be installed automatically as part of the package installation process.

Optional dependencies are also defined in the *pyproject.toml* file, and you can install them when you need them for development purposes or for tests to run in a CI pipeline, for example. Note that they will not be installed automatically when the package itself is added as a dependency, as they are not included in package metadata.

Build-system section

The `build-system` section of the *pyproject.toml* file specifies the build backend that is used to package and distribute the project. In this case, we are using setuptools as the build backend, and the package version is dynamically determined based on the contents of the *version.txt* file.

While several alternative build backends are available, such as build backend provided by uv, or Hatchling, setuptools remains the most mature and widely supported backend in the Python ecosystem:

```
[build-system]
requires = [
 "setuptools>=72.0"]
build-backend = "setuptools.build_meta"

[tool.setuptools.packages.find]
where =["src"]
include = ["hotel_booking*"]

[tool.setuptools.dynamic]
version = { file = "version.txt" }
```

Linting and formatting

For linting and formatting, we use Ruff, which is the fastest tool out there for this purpose. It was developed by Astral and written in Rust. Just like uv, it won the hearts of the Python community and replaced the functionality of many other tools. Configurations for Ruff are specified in the `tool` section of the *pyproject.toml* file:

```
[tool.ruff]
line-length = 90

[tool.ruff.lint]
select = [
   "F",    # pyflakes rules
   "E",    # pycodestyle error rules
   "W",    # pycodestyle warning rules
   "B",    # flake8-bugbear rules
   "I",    # isort rules
   "UP",   # pyupgrade rules
   "SIM",  # flake8-simplify rules
   "ERA",  # eradicate rules
   "C",    # pylint convention rules
   "ANN"   # flake8-annotations rules
]
ignore = [
   "D107", # Missing docstring in `__init__`
   "D203", # incorrect-blank-line-before-class
   "D213", # multi-line-summary-second-line
   "C901", # too complex
   "ERA001", # commented out code
]
```

```
[tool.ruff.format]
indent-style = "space"
```

These configurations are also used for `pre-commit` hooks that should be installed locally to enforce the rules. Pre-commit hooks are defined in the *.pre-commit-config.yaml* file:

```
exclude: ^tests/resources/
repos:
 - repo: https://github.com/pre-commit/pre-commit-hooks
   rev: v6.0.0
   hooks:
     - id: check-added-large-files
     - id: check-json
     - id: check-toml
     - id: check-yaml
     - id: end-of-file-fixer
     - id: trailing-whitespace
 - repo: https://github.com/astral-sh/ruff-pre-commit
   rev: v0.15.1
   hooks:
     - id: ruff
       args: [--fix, --exit-non-zero-on-fix, --show-fixes]
     - id: ruff-format
```

To make pre-commit hooks execute on every commit, install them with the **`pre-commit install`** command. To ensure all the files follow the predefined rules, run the **`uv run pre-commit run -all-files`** command as part of the CI pipeline.

Pytest configurations

The *pyproject.toml* file can also contain some pytest configurations that allow it to run tests without running into errors, and for that reason, we must specify the `testpaths` and `pythonpath` configurations:

```
[tool.pytest.ini_options]
addopts = "-s --no-header --no-summary"
testpaths = [ "tests" ]
pythonpath = [ "." ]
```

Now that we've gone through the *pyproject.toml* file and the dataset, let's take a deep dive into the code.

Project Code: Data Preprocessing

You can find the *chapter2_preprocess_data.py* script in the *notebooks* folder. Note that for the code to run, you must either create a catalog and a schema with the name specified in the *project_config.yml* file under the `dev` key or update the file with the catalog name and schema name you prefer to use instead.

First of all, we create a Spark session and load the project configuration:

```
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig

spark = SparkSession.builder.getOrCreate()

cfg = ProjectConfig.from_yaml("../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema
```

We use the `ProjectConfig` class from the `config` module of our custom `hotel_booking` package. This class is built on Pydantic, which provides robust data validation to ensure that all configuration values match the expected data types.

Beyond basic type checking, Pydantic supports enforcement of additional constraints, such as value ranges, allowed patterns (e.g., regular expressions), and custom validation rules. This helps catch configuration errors early by enforcing these rules at runtime, so misconfigurations will fail fast with clear error messages, instead of causing subtle bugs later in the pipeline.

In the following example, we enforce an opinionated valid value range for hyperparameters by using Pydantic's `Field()` validator. Specifically, we

use parameters such as `gt` (greater than) and `le` (less than or equal to) to define acceptable bounds for the field's value:

```
import yaml
from pydantic import BaseModel, ConfigDict, Field


class Parameters(BaseModel):
    """Model training hyperparameters with validation."""

    learning_rate: float = Field(gt=0, le=0.2)
    n_estimators: int = Field(ge=10, le=10000)
    max_depth: int = Field(ge=1, le=50)


class ProjectConfig(BaseModel):
    """Project configuration parameters loaded from project_config.yml."""

    model_config = ConfigDict(protected_namespaces=())

    num_features: list[str]
    cat_features: list[str]
    target: str
    catalog: str
    schema: str
    parameters: Parameters
    usage_policy_id: str

    @classmethod
    def from_yaml(
        cls: "ProjectConfig",
        config_path: str,
        env: str = "dev") -> "ProjectConfig":
        """Load and parse configuration settings from a YAML file.

        :param config_path: Path to the YAML configuration file
        :param env: Environment name to load environment-specific settings
        :return: ProjectConfig instance initialized with parsed configuration
        """
        if env not in ["prd", "acc", "dev"]:
            raise ValueError(
                f"Invalid environment: {env}. Expected 'prd', 'acc', or 'dev'"
            )

        with open(config_path) as f:
            config_dict = yaml.safe_load(f)
```

```
        config_dict["catalog"] = config_dict[env]["catalog"]
        config_dict["schema"] = config_dict[env]["schema"]
        for k in ["dev", "acc", "prd"]:
            config_dict.pop(k)
        return cls(**config_dict)
```

Before we proceed, we need to make sure the catalog and schema exist and, if not, create them:

```
spark.sql(f"CREATE CATALOG IF NOT EXISTS {catalog}")
spark.sql(f"CREATE SCHEMA IF NOT EXISTS {catalog}.{schema}")
```

After we create the Spark session, load the configuration, and make sure catalog and schema exist, we load and clean the data (note the dataset-specific cleaning commands) and store the result in a Delta table.[5] Also note that we enable `ChangeDataFeed` to track all row-level changes between versions of the Delta table:

```
from datetime import datetime

import pandas as pd

df = pd.read_csv("../data/booking.csv")
df.columns = df.columns.str.replace(r"[ -]", "_", regex=True)
df["date_of_reservation"] = df["date_of_reservation"].apply(
   lambda x: "3/1/2018" if x == "2018-2-29" else x
)
df["date_of_reservation"] = df["date_of_reservation"].apply(
   lambda x: datetime.strptime(x, "%m/%d/%Y")
)
df["arrival_date"] = df["date_of_reservation"] + pd.to_timedelta(
   df["lead_time"], unit="d"
)
df["arrival_month"] = df["arrival_date"].dt.month

dst_table = f"{catalog}.{schema}.hotel_booking"

spark.createDataFrame(df).write.mode("overwrite").saveAsTable(f"{dst_table}")

spark.sql(
 f"""ALTER TABLE {dst_table}
  SET TBLPROPERTIES (delta.enableChangeDataFeed = true);"""
)
```

Now that we've gone through the code (the full version is available in the notebook), our next step is to run it. Before that, however, it is worth taking a closer look at the Databricks developer tools.

Databricks Developer Tools

There are four main local developer tools Databricks provides: Databricks CLI, Declarative Automation Bundles (DABs), Databricks Connect, and Databricks extension for Visual Studio Code. Let's go through them one by one.

Databricks CLI

Databricks CLI is a command line interface that allows you to interact with the Databricks platform. It is a very powerful tool with a large range of commands that cover functionality that's similar to that of Databricks Terraform modules and APIs.

You should use Databricks CLI to authenticate toward Databricks from your local machine. We need to start by installing it, and Databricks provides comprehensive documentation on how to do it on different OS.

Databricks CLI provides multiple authentication options. I do not recommend using personal access tokens, as a leaked token could grant full access to whatever resources you have access to across your Databricks workspace and Unity Catalog, including data, models, jobs, and serving endpoints. Instead, use *user-to-machine authentication*,[6] which relies on short-lived OAuth tokens that expire automatically and do not carry the same risk if intercepted.

For the workspace-level commands, use the following command to authenticate:

```
databricks auth login --host workspace-url
```

It creates an authentication profile in the *.databrickscfg* file that contains the profile name and configurations for the profile. It is also a good practice to

use a user-understandable profile name, so that you can pass the -p option when authenticating:

```
[mlops-book]
host  = https://oreil.ly/DtiuJ
auth_type = databricks-cli
```

You use the profiles to authenticate with Databricks when you're using developer tools.

Declarative Automation Bundles

You may have noticed that the repository contains the *databricks.yml* file. This is the DAB configuration file, which must be located in the root of the bundle project folder.

At their core, DABs enable you to manage Databricks projects through an IaC approach. Using the *databricks.yml* file, you can define the artifacts and resources your project requires. In Chapter 5, we'll take a deep dive into specific resources you can deploy using DABs. For now, let's explore the syntax and core commands.

Syntax of the databricks.yml file

First, every *databricks.yml* file must contain a `bundle` section with a name. Optionally, you can specify the Databricks CLI version and information about your source code. You can also provide an interactive `cluster_id`, which will override any other cluster specifications in the bundle, but this will only work in development mode (which we'll discuss below). I recommend specifying the `cluster_id` under the target section for development purposes instead, as follows (note that the `cluster_id` specification is not relevant for serverless compute):

```
bundle: # Required.
 name: string # Required.
 databricks_cli_version: string
 cluster_id: string
 git:
```

```
    origin_url: string
    branch: string
```

The `artifacts` section is where you can specify which packages need to be built as part of the bundle deployment. It can run any build command (for example, uv build) given that the dependency (here, uv) is installed.

For the whl type, you can set the `dynamic_version` parameter. If you set it to `true`, a dynamic version with a timestamp will be created. This is an important consideration for jobs that use an interactive cluster, as it ensures that the cluster will use the latest code changes. However, this isn't necessary for job clusters or serverless clusters, as they handle this behavior automatically:

```
artifacts:
 some-unique-artifact-identifier:
   build: string
   files:
     - source: string
   path: string
   type: string
   dynamic_version: boolean
```

You can also set up *custom variables* directly in the *databricks.yml* file. Then, you can pass these variables to the deployment by entering the **`databricks bundle deploy -var`** command. For traceability purposes, it is a good idea to pass the `git_sha` variable. We will use this later to tag our experiment runs and models, to ensure that we can always link a model back to the exact version of the code that created it.

You define these variables within a `variables` section like this:

```
variables:
 some-unique-variable-name:
   description: string
   default: string or complex
```

To keep your *databricks.yml* file clean and maintainable, especially as your project grows, you can move job definitions into separate files and

reference them using the `include` section. This modular approach helps you avoid a single, monolithic configuration file that becomes hard to read and review:

```
include:
 - 'some-file-or-path-glob-to-include'
 - 'another-file-or-path-glob-to-include'
```

In the `sync` section of the *databricks.yml* file, you can specify what files or directories should be included or excluded from the bundle deployment:

```
sync:
 include:
   - 'some-file-or-path-glob-to-include'
   - 'another-file-or-path-glob-to-include'
 exclude:
   - 'some-file-or-path-glob-to-exclude'
   - 'another-file-or-path-glob-to-exclude'
 paths:
   - 'some-file-or-path-to-synchronize'
```

Next, in the `resources` section of the *databricks.yml* file, you can define all the resources that should be deployed as part of the bundle. DABs support a wide and continuously expanding set of resource types, including alerts, apps, clusters, dashboards, Lakeflow jobs, and SQL warehouses.[7]

This is how you define resources in the *databricks.yml* file:

```
resources:
 clusters:
   some-unique-programmatic-identifier-for-this-cluster:
     # See the REST API create request payload reference for clusters.
 experiments:
   some-unique-programmatic-identifier-for-this-experiment:
     # See the REST API create request payload reference for experiments.
 apps:
   some-unique-programmatic-identifier-for-this-app:
     # See the REST API create request payload reference for apps.
 jobs:
   some-unique-programmatic-identifier-for-this-job:
     # See the REST API create request payload reference for jobs.
 pipelines:
```

```
    some-unique-programmatic-identifier-for-this-pipeline:
      # See the REST API create request payload reference for
      # Lakeflow declarative pipelines.
```

In the `permissions` section, you can specify permissions for all the resources defined in the bundle configuration. You can set permissions for groups, users, or service principals. The allowed top-level permission levels are `CAN_VIEW`, `CAN_MANAGE`, and `CAN_RUN`. You can also specify permissions on the resource level:

```
permissions:
 - level: <permission-level>
   group_name: <unique-group-name>
 - level: <permission-level>
   user_name: <unique-user-name>
 - level: <permission-level>
   service_principal_name: <unique-principal-name>
```

The `run_as` section allows you to specify the identity, such as a service principal, to use to run a job. The user must have `CAN_USE` permissions for the service principal to use this feature:

```
run_as:
 service_principal_name: <service-principal-name>
```

Finally, you can use the `targets` section to override settings that are specified elsewhere in the *databricks.yml* file. For development purposes, it is convenient to specify an interactive `cluster_id` under the `dev` target and utilize the development mode.

The development mode enables the following behavior:[8]

- It adds the `[dev ${workspace.current_user.short_name}]` prefix to all resource names, and it tags each job with a `dev` Databricks tag.
- It allows you to override job clusters (which is not allowed in production mode).

- It pauses all schedules and triggers.
- It enables concurrent runs on all deployed jobs and disables the deployment lock for faster iteration.

This is how you can define targets in the *databricks.yml* file:

```
targets:
 some-unique-programmatic-identifier-for-this-target:
   artifacts:
     # See the preceding "artifacts" syntax.
   bundle:
     # See the preceding "bundle" syntax.
   cluster_id: string
   default: true | false
   mode: development
   presets:
     preset: value
   resources:
     # See the preceding "resources" syntax.
   sync:
     # See the preceding "sync" syntax.
   variables:
     preceding-unique-variable-name: non-default-value
   workspace:
     # See the preceding "workspace" syntax.
   run_as:
     # See the preceding "run_as" syntax.
```

It is common practice to define the *`workspace` configuration* under a target because workspace-related values, such as the host and authentication details, typically differ by environment.[9] The `workspace` section can contain the following information:

```
workspace:
 artifact_path: string
 auth_type: string
 file_path: string
 host: string
 profile: string
 root_path: string
 state_path: string
```

Note that in development mode, the bundle's root path must be under the user's home directory, so it is common to have different `root_path` values for different targets.

Now that we've covered the different sections of the *databricks.yml* file, let's take a closer look at the version used in this book's code repository.

The databricks.yml file

In the *databricks.yml* file, we specify the bundle name first:

```
bundle:
 name: oreilly-mlops-with-databricks
```

Then, we define the artifacts to build. In this case, we use the `uv build` command to create the package wheel, which we can reference later as the task library. The path is ".", which points to the root of the directory:

```
artifacts:
 default:
   type: whl
   build: uv build
   path: .
```

If you have multiple packages within the same repository, the path will be a folder that contains the *pyproject.toml* file.

Next, we define the following bundle variables:

`git_sha` *and* `branch`

> We will use these later as arguments for our Python tasks, so that we can tag experiment runs and registered models with actual `git_sha` and `branch` values.

`schedule_pause_status`

> We can use this to pause the scheduled job when the target is not set to development mode, which is convenient if we use the `acc` target.

`usage_policy_id`

> We can use this to pass to serverless resources deployed through the bundle for cost allocation. We will discuss this in detail in Chapter 9.

The following example shows how to deploy these bundle variables:

```
variables:
 git_sha:
   description: git_sha
   default: abcd
 branch:
   description: branch
   default: main
 schedule_pause_status:
   description: schedule pause status
   default: PAUSED
# matches project_config.yml
 usage_policy_id:
   description: usage/budget policy id for cost attribution
   default: cdad6f29-a213-4525-9330-e25b90afbe7a
```

In the `sync` section, we specify the `dist` folder to make sure the built artifacts are easily accessible when running the code on Databricks during the development phase:

```
sync:
 include:
   - dist/*
```

Note that we haven't defined any resources directly in our *databricks.yml* file. Instead, we've created separate YAML files for our jobs and placed them in a resources folder. Here's how you can include those resources in the bundle definition; make sure there are no irrelevant files in that directory:

```
include:
 - resources/*
```

Finally, in the `targets` section, we define three targets: `dev`, `acc`, and `prd`. The `dev` target is our default and is used primarily for development. It's configured to use development mode, which automatically adds the `dev ${workspace.current_user.short_name}` prefix to all deployed resources and pauses any schedules.

The `acc` target is intended for acceptance testing, and `prd` is used for deploying code to production. The following example shows how to deploy these three targets:

```
targets:
 dev:
   default: true
   mode: development
   workspace:
     host: <your_host>
     root_path: /Workspace/Users/${workspace.current_user.userName}/
                .bundle/${bundle.target}/${bundle.name}
 acc:
   workspace:
     host: <your_host>
     root_path: /Shared/.bundle/${bundle.target}/${bundle.name}
   variables:
     schedule_pause_status: PAUSED

 prd:
   workspace:
     host: <your_host>
     root_path: /Shared/.bundle/${bundle.target}/${bundle.name}
   variables:
     schedule_pause_status: UNPAUSED
```

In Chapter 5, we'll take a closer look at these environments and explore how to promote code among them.

Now that we've gone through the *databricks.yml* file, let's take a closer look at how we can interact with automation bundles.

Using DABs

Before we go into details, you need to make sure that the host defined in the target matches the Databricks profile you use to log in to Databricks. If

you've forked the repository and cloned it to your local machine, make sure to update the host value in the *databricks.yml* file. Without this change, you won't be able to run the code examples covered in the following sections.

There are four main Databricks CLI commands related to DABs:

`databricks bundle validate`

: This checks whether your *databricks.yml* file is configured correctly.

`databricks bundle deploy`

: This deploys the resources defined in your *databricks.yml* file.

`databricks bundle run <job resource name>`

: This executes the specified job.

`databricks bundle destroy`

: This removes all the assets that were created by the bundle.

After the `databricks bundle deploy` command is executed, all the project files and built artifacts are uploaded to the `root_path` location specified under target. There, you'll find the following:

The artifacts folder

: This holds the Python wheel files that were built for the project.

The files folder

: This contains all your project's source files.

The state folder

: This stores the Terraform state of the bundle.[10]

The resources folder

This contains the direct deployment state file.

It's important to pay attention to authentication when using DABs, specifically the evaluation order:

1. For any command run from the bundle working directory, evaluate and authenticate the values of fields within a project's bundle setting files.
2. Evaluate and authenticate the values of environment variables (`DATABRICKS_HOST` and `DATABRICKS_TOKEN`).
3. Evaluate and authenticate the configuration profile field values within the *.databrickscfg* file.

Note that if you happen to have an environment variable set up, it may mess up authentication.

Databricks Connect

Databricks Connect is a Python package that allows you to trigger an execution of PySpark code on a Databricks cluster from a local environment.

Before we proceed, you need to make sure that the host mentioned in the profile configuration has an active all-purpose cluster that matches the specifications we mentioned earlier, when we went through the all-purpose cluster configuration. Alternatively, you can use serverless.

We have already gone through the *pyproject.toml* setup and uv. Now, you need to create a virtual environment that you can activate later. To do this, from the *project* folder in a terminal, you can run the `uv sync` command as follows:

```
uv sync --extra dev #You can also run an alternative command if you use
Windows.
```

```
source .venv/bin/activate

python
```

Note that you need to use Python 3.12 and Databricks Connect version 17.3.x, to match the runtime. With the Python command, you get into the Python console, and then, you can run the following commands using a serverless cluster:

```
from databricks.connect import DatabricksSession

spark = DatabricksSession.builder.profile(
    "mlops-book").serverless(True).getOrCreate()

df = spark.read.table("samples.nyctaxi.trips")

df.show(5)
```

Alternatively, you can use an interactive cluster by providing the cluster ID.

Spark commands will run on a Databricks cluster.

There are a couple of things to pay attention to here:

- You need to choose the correct profile from the *.databrickscfg* file. If the profile is named DEFAULT, it is not needed.
- You'll use `DatabricksSession` to run Spark code. If you, for example, run `df.toPandas()`, which you may need for an ML library you are using, the dataframe will be loaded into your machine's memory. And if the data is too big, you will get out-of-memory errors.

It may be surprising, but you can import PySpark, even though PySpark is not listed as a dependency of Databricks Connect and uv does not show that PySpark is installed. You can see this if you try to install Databricks Connect in a clean virtual environment as follows:

```
(test) mariavechtomova@Mac test % uv pip install databricks-connect==17.3.5
Resolved 30 packages in 293ms
Prepared 8 packages in 593ms
Installed 30 packages in 301ms
  + certifi==2026.1.4|
  + cffi==2.0.0
  + charset-normalizer==3.4.4|
  + cryptography==46.0.5|
  + databricks-connect==17.3.5|
  + databricks-sdk==0.89.0|
  + google-auth==2.48.0
  + googleapis-common-protos==1.72.0|
  + grpcio==1.78.0|
  + grpcio-status==1.78.0|
  + idna==3.11|
  + numpy==2.4.2
  + packaging==26.0
  + pandas==2.3.3
  + protobuf==6.33.5
  + Py4J==0.10.9.9
  + pyarrow==23.0.1
  + pyasn1==0.6.2
  + pyasn1-modules==0.4.2|
  + pycparser==3.0
  + python-dateutil==2.9.0.post@
  + pytz==2025.2
  + requests==2.32.5
  + rsa==4.9.1|
  + setuptools==82.0.0
  + six==1.17.0
  + typing-extensions==4.15.0|
  + tzdata==2025.3|
  + urllib3==2.6.3
  + zstandard==0.25.0
```

This is because PySpark is “hidden” and embedded into Databricks Connect—which is why it’s a bad idea to list both Databricks Connect and PySpark in your project dependencies. You can view PySpark in this “hidden” location by going to `.venv/lib/python3.12/site-packages` as follows:

```
(test) mariavechtomova@Mac site-packages % ls -la | grep pyspark
drwxr-xI-x@ 56 mariavechtomova staff      1792 Feb 18 00:23 pyspark
```

If you try to use `SparkSession` instead of `DatabricksSession`, however, it would not work because it misses necessary context, as shown here:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
df = spark.read.table("samples.nyctaxi.trips")
df.show(5)
```

In this case, import works without issues because PySpark is part of Databricks Connect. However, when you try to run the second line, you will get the following error message:

```
RuntimeError: “Only remote Spark sessions using Databricks Connect are
supported. Use
DatabricksSession.builder to create a remote Spark session instead.”
```

The same command does work in the Databricks extension for Visual Studio Code, however, because the extension automatically injects the required connection details through a *.databricks.env* file it creates inside the *.databricks* folder of your project. It contains the following environment variables:

- `DATABRICKS_AUTH_TYPE`
- `DATABRICKS_BUNDLE_TARGET`
- `DATABRICKS_HOST`
- `DATABRICKS_METADATA_SERVICE_URL`
- `DATABRICKS_PROJECT_ROOT`
- `DATABRICKS_SERVERLESS_COMPUTE_ID`
- `PROFILE`
- `PYDEVD_WARN_SLOW_RESOLVE_TIMEOUT`

- `SPARK_CONNECT_PROGRESS_BAR_ENABLED`
- `SPARK_CONNECT_USER_AGENT`

Databricks Extension for Visual Studio Code

The Databricks extension for Visual Studio Code integrates with all other developer tools: Databricks CLI, DABs, and Databricks Connect.[11] This makes it very easy to connect to the Databricks workspace and execute your code using a Databricks cluster.

Since this is a monolithic repository (monorepo) setup, I highly recommend installing the Python Envy VS Code extension,[12] which will automatically detect the virtual environment based on the file you have opened.

After you've installed the extension, follow these steps:

1. Open the project; make sure to choose the *mlops* folder of the repository as the local folder, as shown in Figure 2-4).
2. Run the **`cd mlops & uv sync -extra dev`** command to sync dependencies and make sure that the .venv virtual environment has been created.
3. Select an interpreter using Python extension (or it will be automatically selected if Python Envy VS Code extension is installed)
4. Make sure the *databricks.yml* file in the *mlops* folder contains the host that matches the host of the profile that was created earlier.
5. Authenticate using the same profile.
6. Select the cluster that you created earlier or choose the serverless compute option.

In the end, Remote Folder must be present under CONFIGURATION and the Active Environment and Databricks Connect sections must have checkmarks next to them (see Figure 2-4).

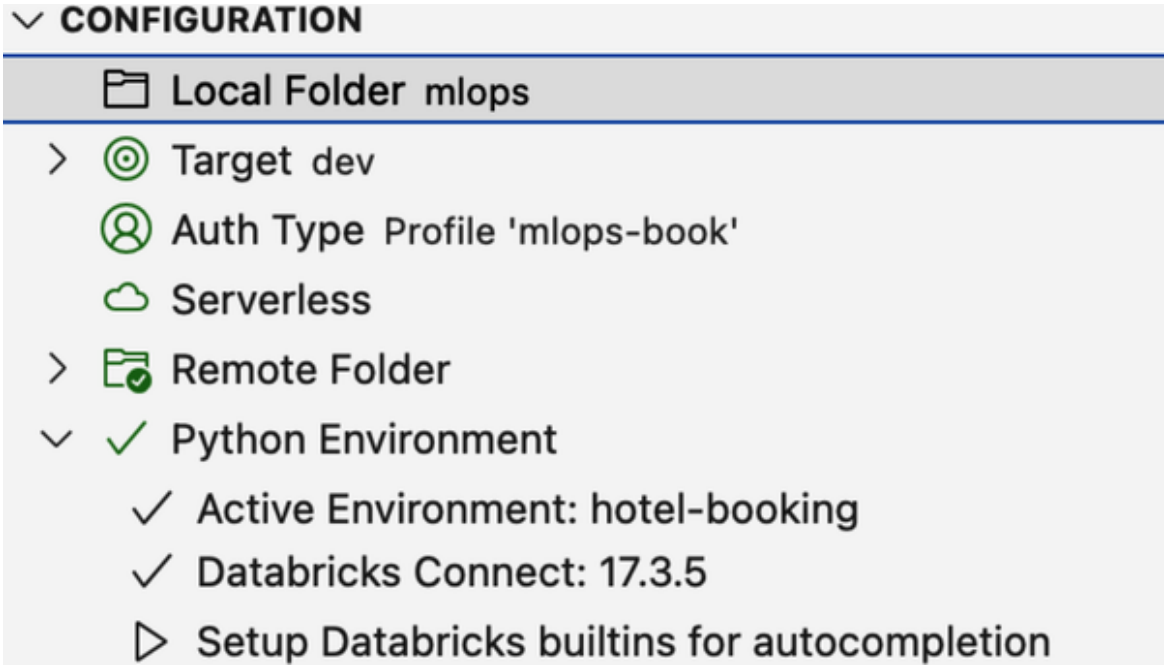


Figure 2-4. Databricks extension in the correct state

Now, let's create a *demo.py* file (the same code we tried to run from a terminal) and add `# Databricks notebook source` so that the code can be recognized as a notebook (see Figure 2-5). The cells are delimited by a line containing `# COMMAND ----------`.[13]

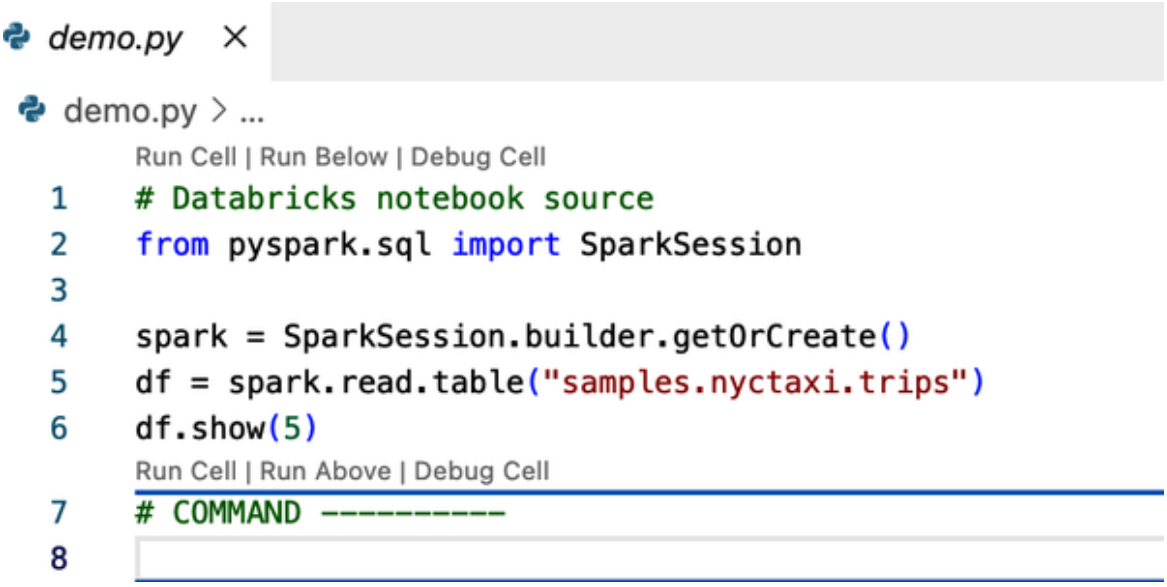


Figure 2-5. Databricks notebook view in Virtual Studio code

We can run the code interactively by clicking **`Run Cell`**. We can also execute project code (for example, *notebooks/chapter2_preprocess_data.py*) from VS Code in a similar way.

However, some code can only run in a Databricks environment. For example, the `databricks-feature-engineering` package is not supported locally, and some Delta table commands will fail outside Databricks. Secondly, even if you use PySpark for preprocessing, most ML libraries require Pandas (or sometimes Polars) DataFrames as input for training. Both Pandas and Polars compute on a single node, which means that even in a Databricks environment, you are not directly leveraging the distributed power of Spark. The advantage of using them, however, is that you can attach your notebook to a larger cluster with more memory. If you run the same code locally, it will use your laptop's memory, and you can't easily scale that up.

Let's explore how to make notebooks available in the Databricks workspace and execute them there instead.

The Databricks extension for Visual Studio Code allows you to sync the entire project to the workspace. To do this, click the **`Sync`** button in the Remote Folder tab on the screen shown previously in Figure 2-4, and the extension will run the `databricks sync <folder> /Users/<user_name>/<folder> --watch` command in the background. As a result, any local changes you make will be automatically reflected in the workspace without you having to perform manual synchronization.

You can also use this approach to make a wheel file for your project available in the workspace. Simply run the **`uv build`** command, and the wheel you generate will be synced along with the rest of the files (unless you add the wheel location to the *.gitignore* file).

It is important to note that the `databricks sync` command provides one-way synchronization. Changes made directly in the Databricks workspace are not reflected locally, and any subsequent sync will overwrite workspace code.

Alternatively, you can make the code accessible in the workspace by using the Git folder functionality. After creating a Git folder, you can authenticate by linking your Git account to Databricks or by using a personal access token from GitHub or another supported version control provider. Git folders enable bidirectional synchronization, which means changes made in notebooks can be pushed back to Git (although this may become inconvenient when dealing with merge conflicts). In addition, managing project dependencies through Git Folders can be challenging.

Finally, you can use the **`databricks bundle deploy`** command to make all project files available in the workspace together with the built artifact, which is the main advantage this approach has over syncing. (Make sure to run the command from the *mlops* folder that contains the *databricks.yml* file.) However, this method also provides one-way synchronization. Note that using the `sync` command together with `bundle deploy` may corrupt the bundle state.

Running Code in a Databricks Notebook

You may have noticed that the *chapter2_preprocess_data.py* script is also just a Python file (not an *IPython* notebook with the *.ipynb* extension) that has `#Databricks notebook source` as the first line and that the cells are delimited by `# COMMAND ----------`.

Let's assume we used the `databricks bundle deploy` command, which makes this notebook available under the files folder in the bundle root in the workspace. Figure 2-6 shows what the notebook looks like if we open it.

```
▶                                                                    1

from datetime import datetime

import pandas as pd
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig
```

```
▶                                                                    2

# Create Spark Session, load configuration
spark = SparkSession.builder.getOrCreate()

cfg = ProjectConfig.from_yaml("../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema
```

Figure 2-6. Databricks notebook view in the Databricks workspace

To run a Databricks notebook, you need to select a cluster and install the project package. Let's see how you can do this for serverless and all-purpose compute.

Using serverless compute

When using serverless compute, you must choose an environment before proceeding. In the example in Figure 2-7, we select a Base environment of Standard v4 and install the custom package wheel that was built together with the bundle (which is available under the bundle root in the *artifacts/.internal* folder). Notice that we selected the `mlops_usage_policy` here so that the costs related to project development will be attributed correctly.

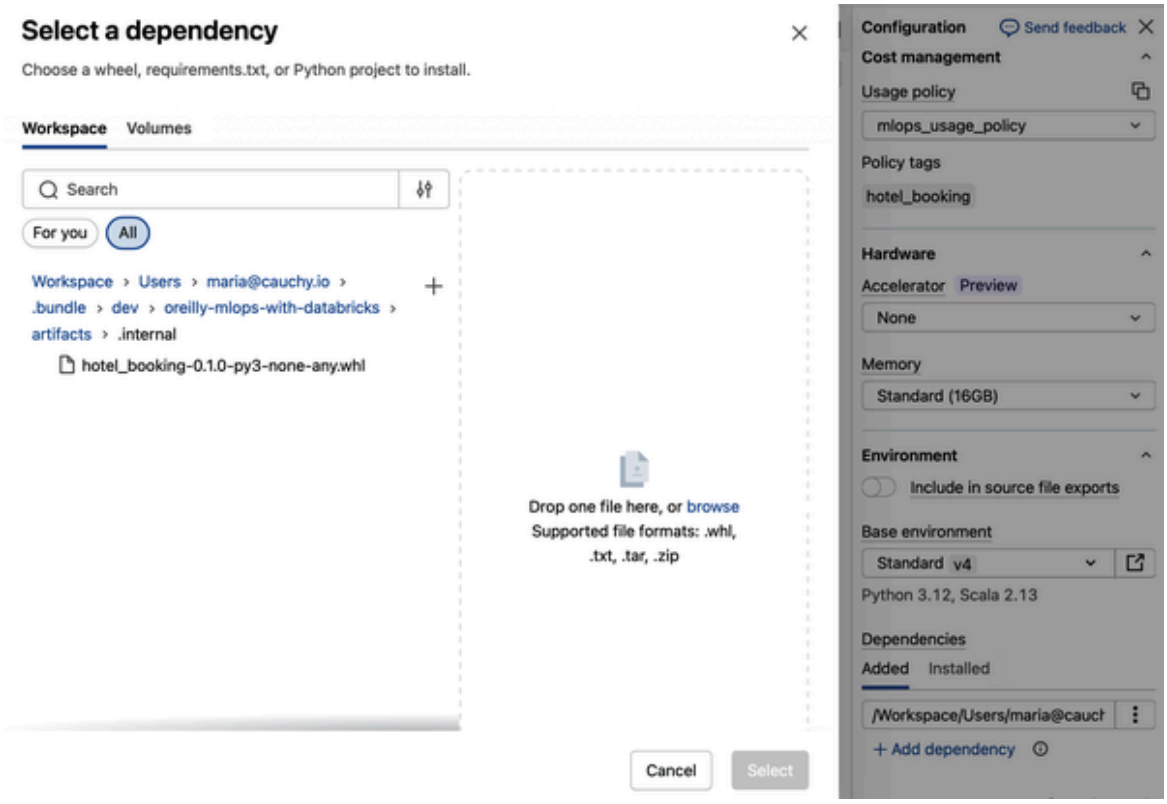


Figure 2-7. Serverless environment configuration

Since selecting an environment is a manual process, developers may forget to choose one. Therefore, you can configure a default custom base environment for serverless compute to ensure that the correct environment is applied to the project.

This environment will be automatically applied to all notebooks when serverless compute is selected. The following *env_v4.yaml* file will be uploaded to */Volumes/mlops_dev/utils/envs/*:

```
environment_version: "4"
dependencies:
 - mlflow==3.11.1
 - numpy==2.4.0
 - pandas==2.3.0
 - pyarrow==22.0.0
 - databricks-sdk==0.102.0
 - scikit-learn==1.7.0
 - lightgbm==4.6.0
 - scipy==1.16.0
 - pydantic==2.11.7
```

Figure 2-8 demonstrates how a workspace admin (who will have the necessary permissions) can set this environment as the workspace base environment via the *Workspace Compute* settings. The *Compute* settings also allow you to configure a default package repository, which is useful when your organization uses JFrog or another private package repository.

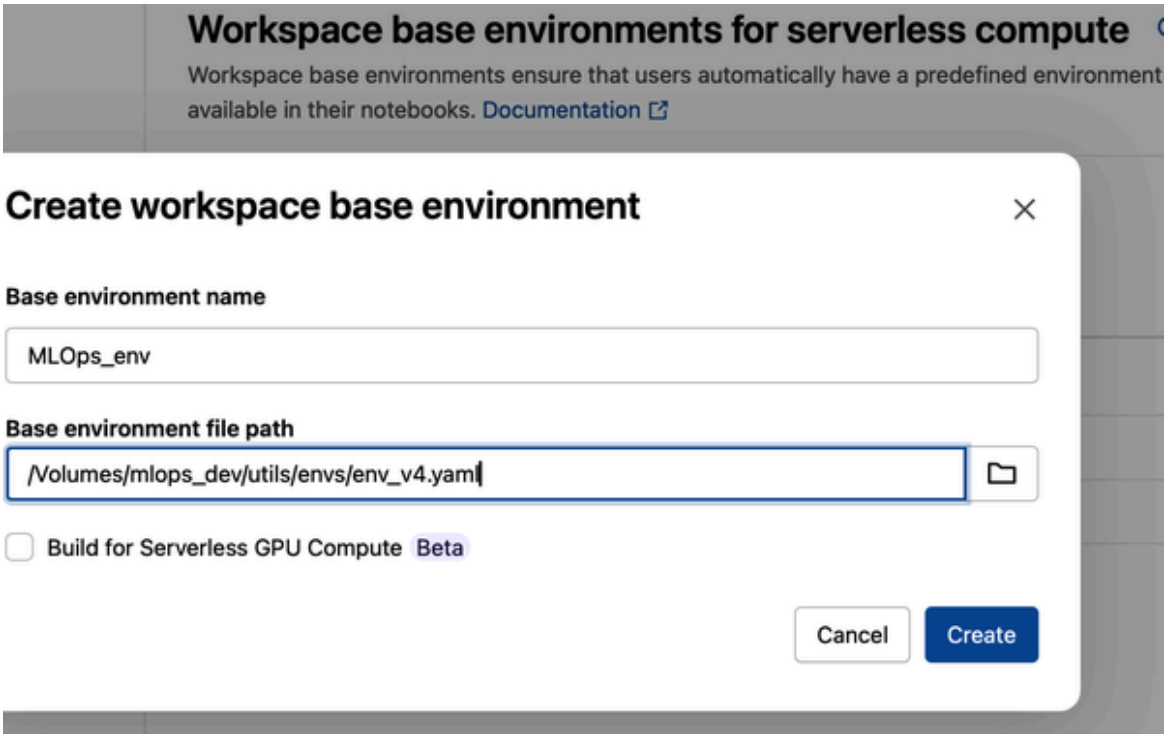


Figure 2-8. Setting a workspace base environment

Using all-purpose compute

If serverless compute does not satisfy your development needs, then you can always opt for all-purpose compute. Earlier in the chapter, we went through configuration of all-purpose compute. We can create a cluster with these specifications, meaning that we will be using Databricks Runtime 17.3 LTS (which is close to environment 4 in terms of its operating system, Python version, and installed packages).

Much like how you installed a custom library in the serverless environment, you can install it on an all-purpose cluster. The custom package wheel is built together with the bundle, and you can find it under the bundle root in the *artifacts/.internal* folder (see Figure 2-9).[14]

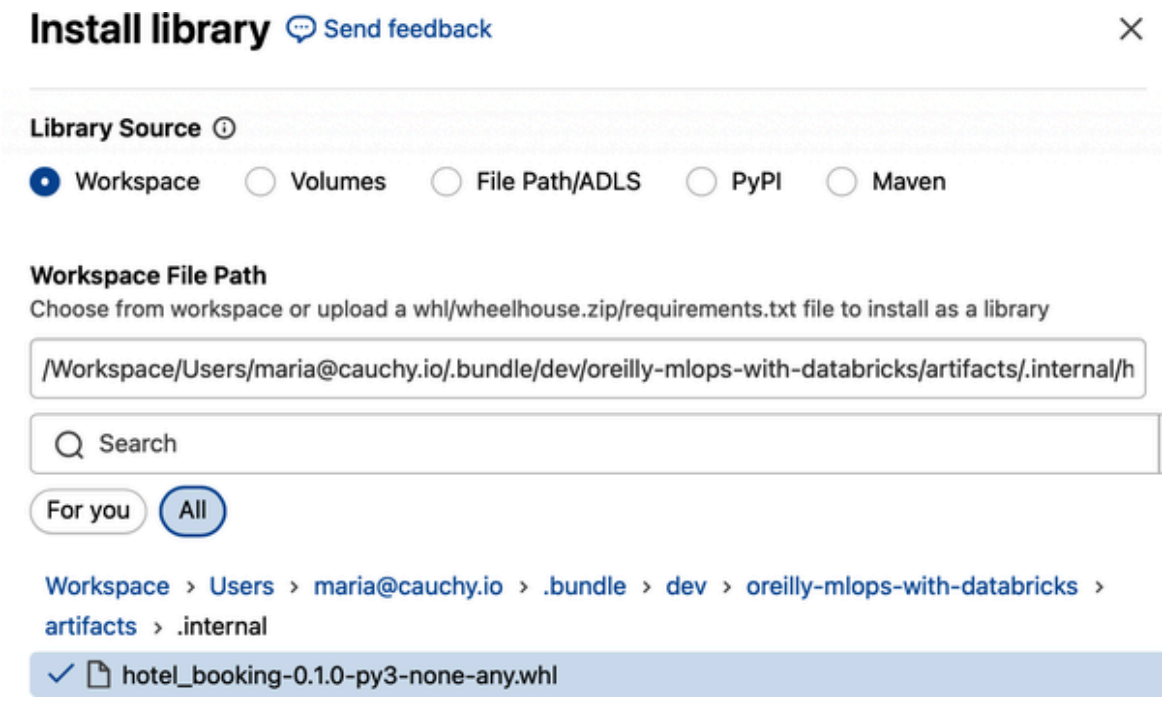


Figure 2-9. Installing a custom library on an all-purpose compute

If the custom package has dependencies hosted in a private package repository, you need to make sure that the default package repository is configured in the Compute settings.

We have already gone through the options for making sure that custom dependencies are available in the notebook for both serverless and all-purpose compute options. However, this is not an interactive way to develop, especially if you want to change the modules in your package. Also, we have assumed that we've been using the bundle deploy option—but what if we used Git folders instead?

To install specific libraries across all nodes of a cluster from a Databricks notebook, you can also use the **`%pip install`** command. This allows you to install them from PyPi, a private repository, or a wheel located in the workspace or a volume.

You can also install all the project dependencies specified in the *pyproject.toml* file by running the **`pip install -e .`** command and restarting Python with the **`dbutils.library.restartPython()`** command. To make the project itself importable, you also need to run the

following code to adjust the path, which is considered a bad development practice:

```
from pathlib import Path
import sys

sys.path.append(str(Path.cwd().parent / 'src'))
```

However, this is necessary in this example because our package code is under the *src* folder.

Overall, developing directly in a Databricks notebook is not my first choice. Notebooks make it difficult to structure code into modules, run tests, and apply consistent formatting. However, to be 100% sure that the code works as intended, you ultimately need to execute it in a Databricks environment.

However, you cannot fully reproduce the same environment you have on Databricks locally—you can only achieve an approximation (as we did earlier). This is because the Databricks Runtime is curated and preconfigured with a set of installed libraries. When additional dependencies are installed on top of it, the resulting dependency tree may differ from what you have locally. Installing all runtime libraries locally would not solve this problem either, as the requirements are not OS-agnostic.

The closest you can get locally is by using a Databricks Runtime image in combination with development containers. Even then, the Python environment will not be identical, as Databricks adds additional components that cannot be reproduced outside of the platform.

Fortunately, Databricks developer tools are quite mature today. As an alternative to running notebooks interactively, you can wrap a notebook as a job with specified compute and dependencies and execute it using the `databricks bundle run <job-resource-name>` command. We will go through this approach in Chapter 5 (for classical ML) and Chapter 9 (for LLM-based applications).

Conclusion

In this chapter, we went through the cluster configurations, the project setup, best practices, and different ways of developing on Databricks. We also showed you some alternatives to developing in a Databricks notebook. But even if you use notebooks, there are some steps you can take that will reduce time to production in the future:

- Refactor your code into functions and create classes. Structuring code using classes can make the codebase more organized, readable, and modular. With classes, you can bundle class attributes and methods (class functions) into a single unit. You should also separate the data loading, processing, and output steps. This will make the code easier to test and modify.
- Move code to separate modules. Create a new module for each set of classes. This will make your codebase more modular and reusable. It will also keep your repository nice and tidy and well organized.
- Package your code. Treat every Python project as a package and specify all the project dependencies and tool configurations in the *pyproject.toml* file.
- Create a configuration file or module (like the *project_config.yaml* file) to store hyperparameters and settings. Keeping configuration settings separate from the main code will allow you to easily adjust the parameters without modifying the main code. It will also be easier to create test scenarios with different configuration settings.
- Add logging to provide a detailed record of what happens during the execution of your program. When errors or unexpected behavior occur, these logs can help you diagnose and debug issues.
- Test your code to “fail fast” so you can avoid facing issues in the middle of running your code.

1 You can download the dataset from Kaggle.

2 You can read more about Anaconda on their official website.

3 Poetry did better than other tools, became one of the most popular backends in the ML community, and contributed to the wider adoption of *pyproject.toml*. You can read a blog about the popularity of Python build backends over time.

4 As of 2026, the company behind uv, Astral, has been acquired by OpenAI. Although it has committed to continue support, if anything changes, the *pyproject.toml* standard we follow in this book should ensure stability for other package managers if required.

5 Note that we use the "`overwrite`" mode here as we are dealing with a static dataset.

6 For more details, consult the Databricks documentation.

7 Consult the Databricks documentation at *https://oreil.ly/6gL1u* for all supported DABs resources.

8 Consult the Databricks documentation at *https://oreil.ly/qSSdT* for more details on development mode.

9 You can also define the `workspace` configuration at the top level of the file, but it's not done much in practice.

10 DABs were originally built on top of the Databricks Terraform provider. Databricks CLI version 0.279.0 and above support two different deployment engines: *Terraform* and *direct*. The Terraform deployment engine will eventually be deprecated, and the direct deployment engine will become the default thereafter. It uses Databricks Go SDK.

11 To install the extension, follow the official documentation.

12 You can find Python Envy in the Visual Studio Code marketplace.

13 This works because the extension created a configuration in the *.vscode* folder (`settings.json`).

14 Given that we also synchronized the *dist* folder in the bundle configuration, we could reference the wheel from the *dist* folder instead.

Chapter 3. MLflow for Traditional ML

MLflow is an open-source tool that was originally created by Databricks for experiment tracking and model registry, and it's one of the most popular tools in the MLOps ecosystem. It is available both as a standalone tool and as an integrated component of the Databricks platform.

Early in my career, before MLflow existed, we built our own internal tooling to store metrics, parameters, and models. It worked, but it was painful to maintain and hard to standardize across teams. When MLflow came along, it solved exactly that problem, and it has only grown since.

MLflow provides a wide range of features, for both standard machine learning models and LLM-based applications. Here's a list of the most important ones (which we'll also cover in Chapter 8):

MLflow Tracking

> This is a centralized repository that logs ML model parameters, metrics, artifacts, and a reference to data used for model training. Experiment tracking gives insights into changes in models over time, and makes it possible for you to search for runs and experiments, so it's crucial for traceability and reproducibility.

MLflow Model Registry

> This is a centralized model store that allows users to manage a model's lifecycle and add tags, aliases, and new model versions. It has a different backend than experiment tracking. To register a model, you must first log it using MLflow Tracking. On Databricks, you should use the Unity Catalog model registry to register models.

MLflow evaluation

> This is a set of tools that facilitate model evaluation and model comparison. In traditional ML applications, you can use it to compare a new model against the baseline model or a threshold. MLflow supports a range of metrics for LLM evaluation (such as answer correctness, answer relevance, and faithfulness), as well as custom metrics that a user can define.

MLflow Tracing

> This is a feature that is integrated into MLflow Tracking and that enhances the observability of GenAI applications by recording inputs, outputs, and metadata associated with each step within the application. This allows users to pinpoint the source of unexpected behavior when it occurs.

This book focuses on the Databricks-managed version of MLflow, which differs in several ways from the open-source project. As a result, instead of covering features like the MLflow AI Gateway and MLflow Model Serving in their open-source form, we'll highlight the corresponding alternatives that are available within the Databricks platform.

We'll begin by exploring *experiment tracking*, which is one of the most mature and widely used components of MLflow. It serves as the foundation for managing ML model runs.

Experiment Tracking

Experiment tracking helps teams keep track of model performance, configuration, and results across iterations. As ML projects grow in complexity, it gets easy to lose track of which model version was trained with which parameters or dataset.

MLflow addresses this challenge with its experiment tracking component. In MLflow, *experiments* serve as the primary unit of organization for model training runs. Every *run*, which captures metadata such as parameters, metrics, artifacts, and source data, is associated with an experiment. That makes it easy to compare and reproduce results.

Tracking URI

By default, MLflow will track experiment runs using the local file system, and all the metadata will be stored in the *./mlruns* directory. Since we use the Databricks-managed version of MLflow, you need to set tracking URI to `"databricks"` when running code locally so that all the experiment runs will be tracked within a Databricks workspace. Even though experiment tracking does not use registry URI, starting with MLflow version 3, you still need to set the MLflow registry URI to `"databricks-uc"`.

You do this by setting the **`MLFLOW_TRACKING_URI`** and **`MLFLOW_REGISTRY_URI`** environment variables or running the `mlflow.set_tracking_uri()` and `mlflow.set_registry_uri()` commands. Please note that you need to provide the profile you used to authenticate to Databricks in the tracking and registry URI. These profiles are defined in the *.databrickscfg* file (an example of the profile name could be `mlops-book`). If you use the `DEFAULT` profile, you do not need to pass the profile name to the tracking URI:

```
import mlflow

profile="mlops-book"
mlflow.set_tracking_uri(f"databricks://{profile}")
mlflow.set_registry_uri(f"databricks-uc://{profile}")
```

If we run the code on Databricks, however, we don’t need to set the tracking URI. We also want multiple developers with a different profile name to collaborate on the same code base. The solution would be to store the profile name in an *.env* file (which should be ignored by Git via a

.gitignore file) and set the `PROFILE` environment variable by using the python-dotenv package.

We can then define whether the code runs on Databricks by checking whether or not the `DATABRICKS_RUNTIME_VERSION` environment variable is available. We can define that function under the common module of the custom package (*src/hotel_booking/utils/common.py*) and use this code in all the scripts that use MLflow:

```
import os

import mlflow
from dotenv import load_dotenv

def set_mlflow_tracking_uri() -> None:
    if "DATABRICKS_RUNTIME_VERSION" not in os.environ:
        load_dotenv()
        profile = os.environ["PROFILE"]
        mlflow.set_tracking_uri(f"databricks://{profile}")
        mlflow.set_registry_uri(f"databricks-uc://{profile}")
```

We can then check the current tracking and registry URI by using the `mlflow.get_tracking_uri()` and `mlflow.get_registry_uri()` commands. After we have set the tracking URI, we can define the MLflow experiment.

MLflow Experiment

You can create an MLflow experiment by using the `mlflow.create_experiment()` or `mlflow.set_experiment()` function. The `mlflow.set_experiment()` function will activate the existing experiment by name or will create a new one if such an experiment does not exist yet:

```
mlflow.set_experiment(experiment_name="/Shared/demo")
```

You can add tags to your experiment by passing it as a parameter to the `mlflow.create_experiment()` command, or if the experiment already

exists, you can use the `mlflow.set_experiment_tags()` command after the experiment gets activated:

```
mlflow.set_experiment_tags(
    {"repository_name": "mvechtomova/oreilly-mlops-with-databricks"})
```

You can then retrieve an experiment by its experiment id, by using the `mlflow.get_experiment()` command as follows:

```
mlflow.get_experiment(experiment.experiment_id)
```

You can also search for the MLflow experiment by name or by tag. The `search_experiments()` function outputs a list of instances of an experiment class that contains information about the experiment, such as its `experiment_id`, creation time, and artifact location, plus tags created by Databricks and by the users:

```
experiments = mlflow.search_experiments(
   filter_string="tags.repository_name='mvechtomova/oreilly-mlops-with-
databricks'"
)
```

You can also easily view how the class is constructed. First, you save all class attributes in a JSON file:

```
import json
import os

if not os.path.exists("../demo_artifacts"):
   os.mkdir("../demo_artifacts")
with open("../demo_artifacts/mlflow_experiment.json", "w") as json_file:
   json.dump(experiment.__dict__, json_file, indent=4)
```

Then, you visualize it with JSON Lens or a similar Visual Studio Code extension (see Figure 3-1).

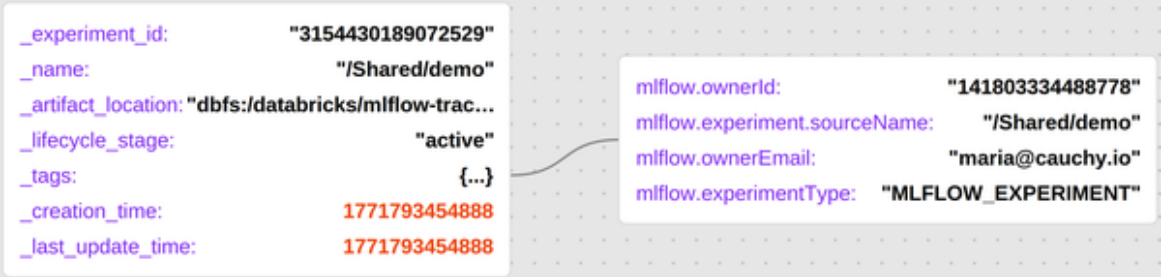


Figure 3-1. Visualization of the mlflow.entities.experiment.Experiment class attributes

MLflow Run

We can create an MLflow run by running the `mlflow.start_run()` command. Let's start a run and log some metrics and parameters:

```
with mlflow.start_run(
   run_name="demo-run",
   tags={"git_sha": "1234567890abcd",
         "branch": "main"},
   description="demo run",
) as run:
   mlflow.log_params({"type": "demo"})
   mlflow.log_metrics({"metric1": 1.0, "metric2": 2.0})
```

Note that we use the command within the `with` block. If you didn't use it, you would have to run the `mlflow.end_run()` command to end the run. In this run, we logged a parameter and some metrics.

We can retrieve the run by using the `mlflow.get_run()` command:

```
run_info = mlflow.get_run(run_id=run_id)
```

You can easily visualize how the MLflow run class is constructed. First, save the run information as a JSON file:

```
run_info_dict = run_info.to_dictionary()
with open("../demo_artifacts/run_info.json", "w") as json_file:
   json.dump(run_info_dict, json_file, indent=4)
```

Then, visualize it with JSON Lens (see Figure 3-2).

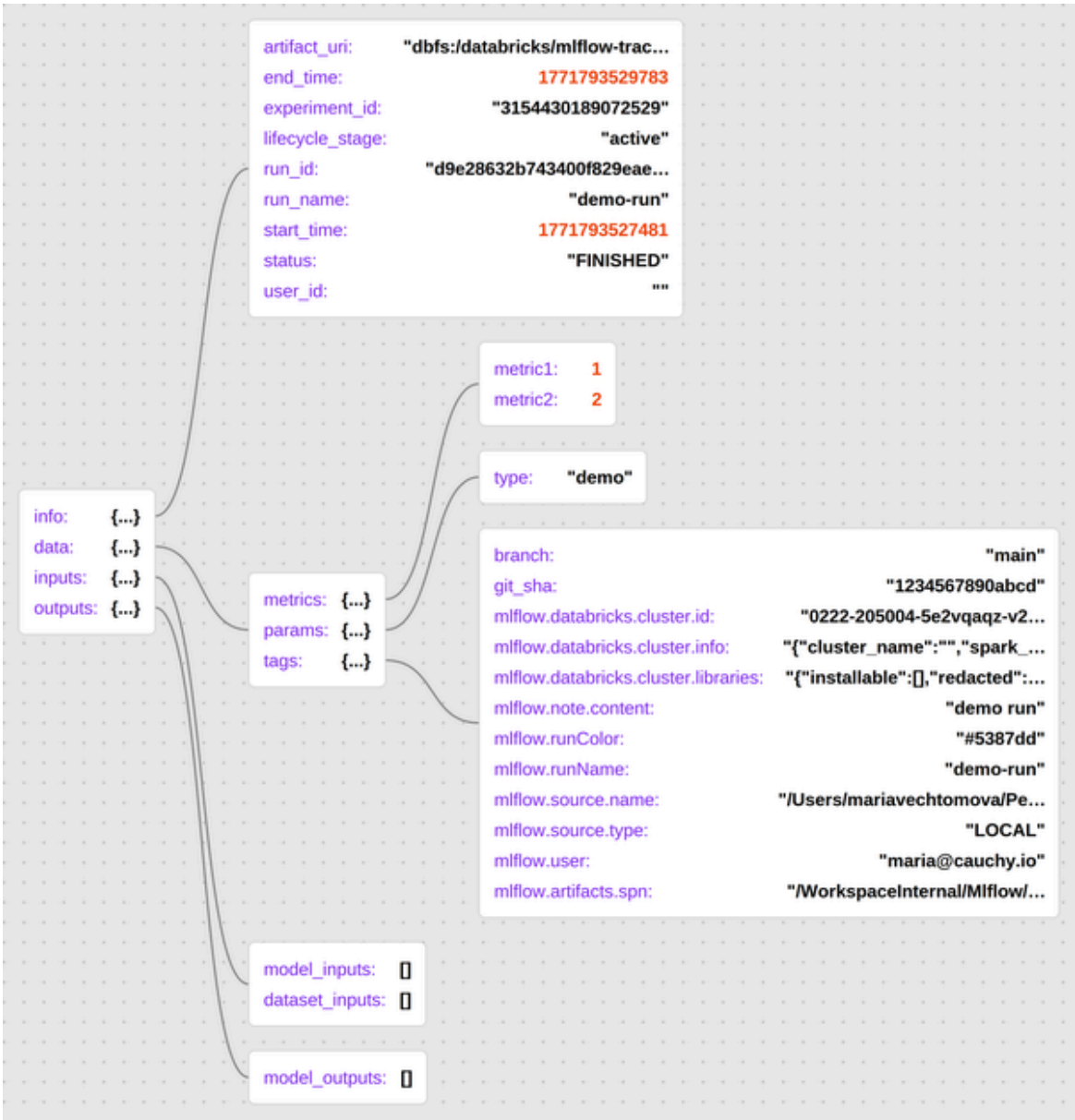


Figure 3-2. Visualization of the mlflow.entities.run.Run class attributes

We can access the logged metrics and parameters via the dictionary that contains run information that we defined earlier:

```
run_info_dict["data"]["metrics"]
run_info_dict["data"]["params"]
```

And we can also search for a run by using the `mlflow.search_runs()` function:

```
from time import time

time_hour_ago = int((time() - 3600) * 1000)

runs = mlflow.search_runs(
   search_all_experiments=True,  # or experiment_ids=[], or experiment_names=
[]
   order_by=["start_time DESC"],
   filter_string="status='FINISHED' AND "
   f"start_time>{time_hour_ago} AND "
   "run_name LIKE '%demo-run%' AND "
   "metrics.metric3>0"
)
```

The function supports complex string filtering, which we can use to filter by tag, metrics, run name, start time, and nearly any other data related to the run.

We can also start the existing run again and log extra parameters, metrics, and artifacts. Note that we can't overwrite the old data. For example, if `metric1` already exists and has a value of 1.0, we cannot assign a value of 2.0 to it.

Let's start the run again and log some artifacts and metrics:

```
mlflow.start_run(run_id=run_id)

for i in range(0,3):
   mlflow.log_metric(key="metric1", value=3.0+i/2, step=i)
mlflow.log_artifact("../demo_artifacts/mlflow_meme.jpeg")
mlflow.log_text("hello, MLflow!", "hello.txt")
mlflow.log_dict({"k": "v"}, "dict_example.json")
mlflow.log_artifacts("../demo_artifacts", artifact_path="demo_artifacts")
```

We can log metrics dynamically, like during each training epoch. We can log any file using the `mlflow.log_artifact()` command. To log a folder, we can use the `mlflow.log_artifacts()` function instead. We can also log text and a dictionary with a dedicated command.

MLflow experiment tracking supports logging images and figures. You can even log images dynamically, for example, after each training epoch, or in a

loop like in the following example. Since you're not using a `with` block, remember to end the run once you've finished logging:

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
ax.plot([0, 1], [2, 3])
mlflow.log_figure(fig, "figure.png")

for i in range(0,3):
   image = np.random.randint(0, 256, size=(100, 100, 3), dtype=np.uint8)
   mlflow.log_image(image, key="demo_image", step=i)

mlflow.end_run()
```

At this point, we've covered the most important logging functionalities that are relevant to ML projects, except logging the models. We'll get to that in the next section of this chapter. All the functionalities we've discussed so far weren't marked as experimental at the time of writing. Earlier, we showed you how to retrieve parameters and metrics; now, let's look at how to retrieve logged artifacts.

First of all, we need to retrieve the artifact URI, which we must do if we want to use the `load_dict()`, `load_image()`, and `load_text()` functions from the `mlflow.artifacts` module:

```
artifact_uri = runs.artifact_uri[0]
dict_example = mlflow.artifacts.load_dict(
   f"{artifact_uri}/dict_example.json")
figure = mlflow.artifacts.load_image(f"{artifact_uri}/figure.png")
text = mlflow.artifacts.load_text(f"{artifact_uri}/hello.txt")
```

Then, download artifacts using the `mlflow.artifacts.download_artifacts()` function:

```
if not os.path.exists("../downloaded_artifacts"):
   os.mkdir("../downloaded_artifacts")
mlflow.artifacts.download_artifacts(
   artifact_uri=f"{artifact_uri}/demo_artifacts",
   dst_path="../downloaded_artifacts")
```

Figure 3-3 demonstrates all the logging and loading functionality for MLflow.

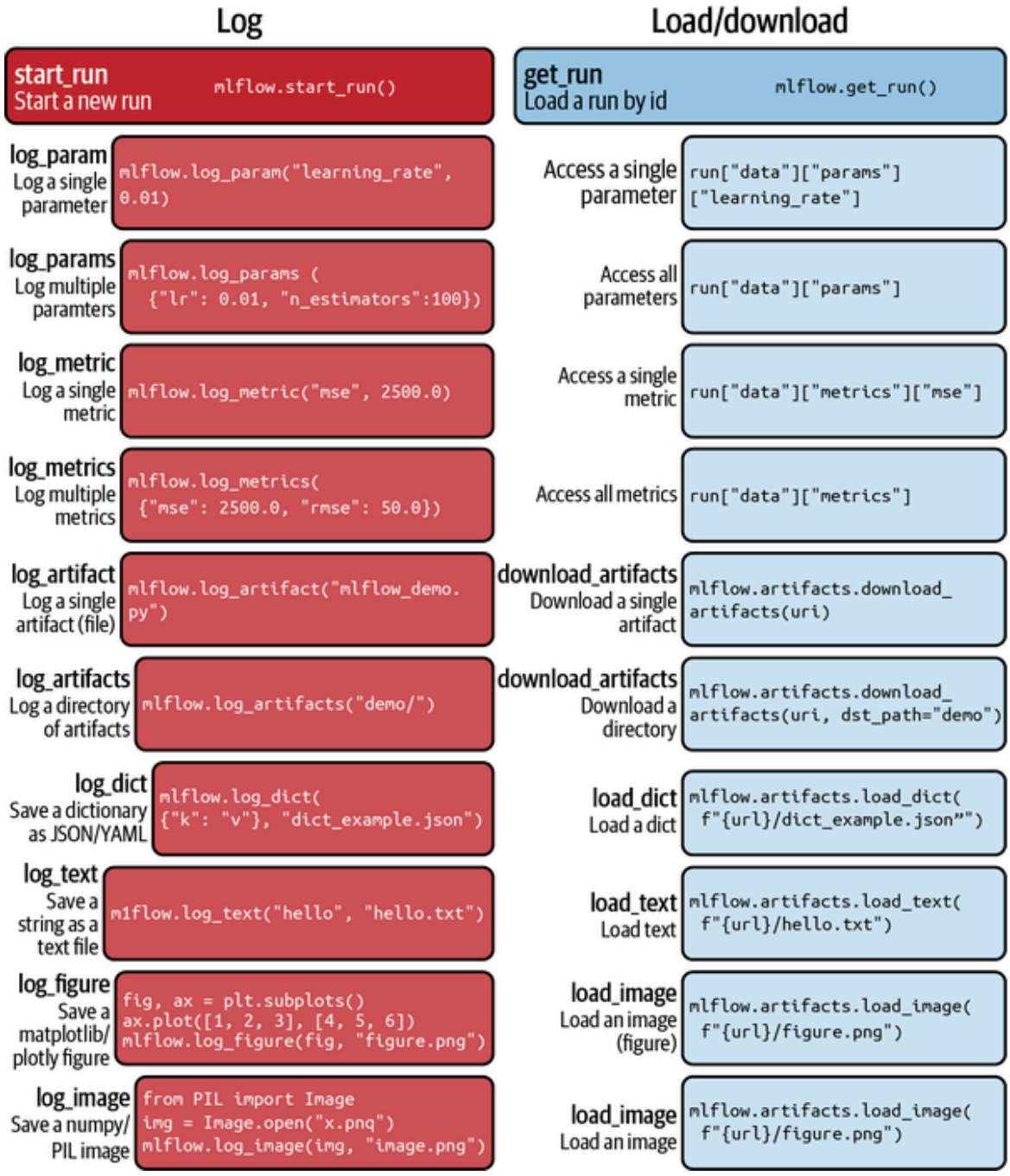


Figure 3-3. MLflow logging and loading functionality

All the logged information can be also found in the UI. Figure 3-4 demonstrates the top-level overview of the run. There, we can find logged

metrics and parameters, together with the information about the run and logged artifacts.

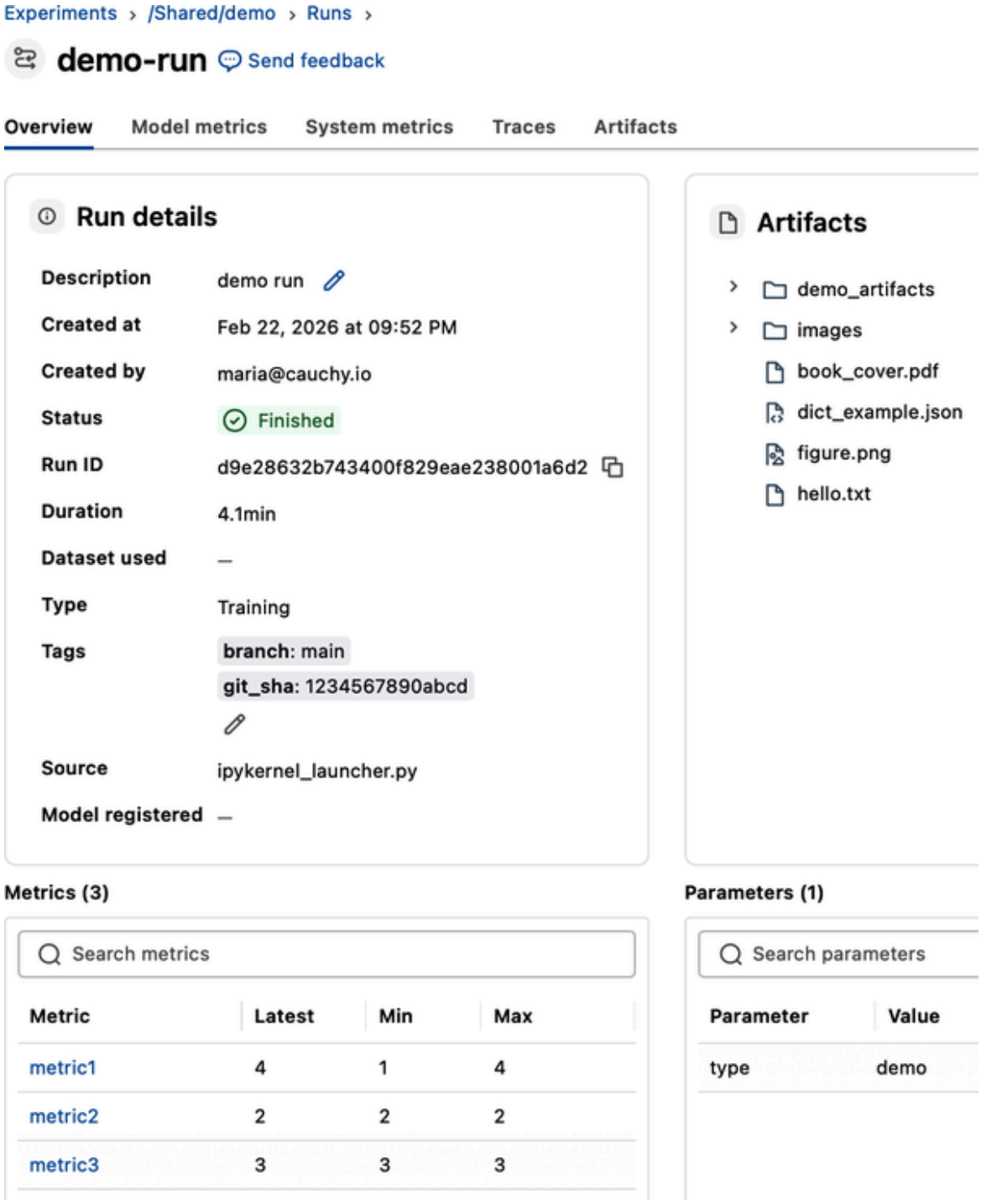


Figure 3-4. View of the run in the UI

Another important function for experiment tracking is `mlflow.log_input()`. This feature improves traceability and reproducibility by allowing you to reference the data used for model training. Later, we'll demonstrate how to log input data when logging the model.

For certain model flavors, we can auto-log using the `mlflow.<model_flavor>.autolog()` command, which will store metrics and parameters for ML models without explicitly coding them.

Training Data

Let's define how to split the data into the training, test, and validation sets. Our example dataset contains records from 2017-05-20 to 2018-12-31. Hotel prices are seasonal, and it makes sense to include the whole year of data in the training set. However, by including more than 12 months, we'll have more samples of data for certain months, which may harm model performance. Therefore, we'll use *time-based splits*, which means that we'll train on 12 months of data, use the subsequent month for validation (we'll use the validation set in the hyperparameter tuning), and use one month after that for testing.

In general, whenever possible, you should use time-based splits. For example, you should train your model on two weeks of data and evaluate it on the following week. This is the better approach because some features, like product recommendations that change with the seasons, may be time-correlated in ways that aren't immediately obvious.

This works for datasets in which the timestamp of data generation is known. But sometimes, it's not, and we've often seen that the split is redefined every time the model is retrained. This leads to *data leakage*: IDs from the train set end up in the test set and vice versa. A random state won't help. A better approach in this case is to fix the train-test split from the start and keep it separate. Then, when new data arrives, it will be divided between the training and testing sets without any reshuffling.

Now, let's define a `DataLoader` class under the `hotel_booking.data` module:

```
from datetime import datetime, timedelta

import pandas as pd
from dateutil.relativedelta import relativedelta
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig
from hotel_booking.utils.common import get_delta_table_version


class DataLoader:
    """Loads, splits, and accesses datasets in both Spark and pandas
formats."""

    def __init__(self, config: ProjectConfig, spark: SparkSession) -> None:
        """
        :param config: Project configuration containing table and feature info.
        :param spark: Active Spark session.
        """
        self.cfg = config
        self.spark = spark
```

The split method outputs `X_train`, `y_train`, `X_test`, and `y_test`, which belong to the right time period from the `hotel_booking` table and are used further to train the model. The split method accepts an optional version argument, which refers to the version of the delta table:

```
def split(self, test_months: int = 1, train_months: int = 12,
    max_date: datetime = None, version: str = None,
) -> tuple[pd.DataFrame, pd.Series, pd.DataFrame, pd.Series]:
    """
    #Splits the dataset and loads it into pandas DataFrames for training and
testing.

    :param test_months: Number of months for the test set.
    :param train_months: Number of months for the training set.
    :param max_date: The maximum date for splitting. If None, uses the latest
date.
    :param version: Optional Delta table version to query.
    :return: A tuple (X_train, y_train, X_test, y_test) with pandas
```

```
    DataFrames.
        """
        self.train_query, self.test_query = self.generate_queries(
            test_months, train_months, max_date, version)

        self.train_set_spark = self.spark.sql(self.train_query)
        self.test_set_spark = self.spark.sql(self.test_query)

        train_set = self.train_set_spark.toPandas()
        test_set = self.test_set_spark.toPandas()

        X_train = train_set[self.cfg.num_features + self.cfg.cat_features]
        y_train = train_set[self.cfg.target]

        X_test = test_set[self.cfg.num_features + self.cfg.cat_features]
        y_test = test_set[self.cfg.target]

        return X_train, y_train, X_test, y_test
```

The split() method relies on `generate_queries()` to construct queries for reading the training and test datasets. These queries are based on the start and end dates of the training and testing periods, which are calculated by `_get_start_end_dates()` using the provided period length and `max_date`. If the `max_date` is not specified, the latest available date will be retrieved from the `hotel_booking` table. Similarly, if no Delta table version is provided, the most recent version will be obtained from the table history:

```
def generate_queries(
    self, test_months: int = 1, train_months: int = 12,
    max_date: datetime = None, version: int = None,
) -> tuple[str, str]:
    """
    #Generates SQL queries for the training and test splits.

    :param test_months: Number of months for the test set.
    :param train_months: Number of months for the training set.
    :param max_date: The maximum date for splitting. If None, uses the max
date.
    :param version: Optional Delta table version to query.
    :return: A tuple containing the training and test SQL queries.
    """
    table_ref = f"{self.cfg.catalog}.{self.cfg.schema}.hotel_booking"
```

```
        if max_date is None:
            max_date = self.spark.sql(
                f"SELECT MAX(date_of_reservation) AS max_date FROM {table_ref}"
            ).collect()[0]["max_date"]

        if version is None:
            version = get_delta_table_version(self.spark, table_ref)

        train_start, train_end, test_start, test_end = self._get_start_end_dates(
            max_date, test_months, train_months)

        train_query = f"""
            SELECT * FROM {table_ref} VERSION AS OF {version}
            WHERE arrival_date BETWEEN DATE('{train_start}') AND
    DATE('{train_end}')
        """

        test_query = f"""
            SELECT * FROM {table_ref} VERSION AS OF {version}
            WHERE arrival_date BETWEEN DATE('{test_start}') AND DATE('{test_end}')
        """

        return train_query, test_query
```

We won't go into the implementation details of `_get_start_end_dates()` and `get_delta_table_version()` from the common module here; refer to the code for more details. Most importantly, the generated queries will allow us to get the exact version of data used for training, even if data is modified later, thanks to the time travel functionality of delta tables.

> **NOTE**
>
> Remember to set a proper retention period on the delta table (the default is seven days). Otherwise, you may not be able to access the exact version of the table if the VACUUM command has been executed. Most accounts have predictive optimization enabled by default, which means that Databricks automatically triggers the VACUUM operation as part of its optimization process.

Notice that the split method uses PySpark to read the Delta table and then converts the PySpark DataFrame to Pandas, which is required as input for

the LightGBM model that we will train later. We chose PySpark here for simplicity and because it is currently the easiest way to access data stored in Delta tables.

However, for large datasets, this conversion is not very efficient. A faster alternative is to use credential vending together with `pyarrow` and the `deltalake` library, but that approach requires a more complex setup. Additionally, we will need a PySpark DataFrame later to log the input data within the MLflow run.

Now, we need to load a training and validation set for hyperparameter tuning (where the validation set ends on 2018-11-30). You can find the code in the `chapter3_2.model_finetuning.py` notebook:

```
import os
from datetime import datetime

import pandas as pd
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig
from hotel_booking.data.data_loader import DataLoader
from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()

cfg = ProjectConfig.from_yaml(config_path="../project_config.yml")

spark = SparkSession.builder.getOrCreate()
data_loader = DataLoader(spark=spark, config=cfg)

X_train, y_train, X_valid, y_valid = data_loader.split(
   test_months=1,
   train_months=12,
   max_date=datetime.strptime("2018-11-30", "%Y-%m-%d")
)
```

Now that we have loaded the data, let’s see how we can use nested MLflow runs for hyperparameter tuning.

Hyperparameter Tuning and Nested Runs

Every ML project starts with setting a business problem, analyzing the data, choosing a model, and tuning the hyperparameters. However, hyperparameter tuning does not just happen at the beginning of the project. It's important to keep reevaluating hyperparameters periodically when a model is in production. However, hyperparameter tuning is an expensive exercise, and it should not occur every time the model gets retrained.

There are several popular libraries for hyperparameter tuning, including Hyperopt, Optuna, and Ray Tune. Historically, Hyperopt was the go-to solution, and it's still widely referenced in older examples and blog posts. However, the ecosystem has evolved, and Hyperopt is no longer maintained.

Today, it's important to distinguish between the optimization algorithm and the orchestration layer. Optuna provides state-of-the-art hyperparameter search algorithms such as Tree-structured Parzen Estimator (TPE), while Ray Tune focuses on scalable and distributed execution of tuning workloads.

In this chapter, we use Ray Tune together with Optuna: Optuna defines the search strategy, and Ray Tune handles parallelization and distributed execution. This approach reflects current best practices for running large-scale tuning workloads on Databricks with MLflow.

MLflow nested runs are especially convenient for hyperparameter tuning: all the child runs will be grouped under one parent run, which will help with comparison and evaluation. Here is the most basic example of MLflow nested runs:

```
with mlflow.start_run(run_name="top_level_run") as run:
   for i in range(1,5):
       with mlflow.start_run(run_name=f"subrun_{str(i)}", nested=True) as
subrun:
           mlflow.log_metrics({"m1": 5.1+i,
                               "m2": 2*i,
                               "m3": 3+1.5*i})
```

Next, let's demonstrate how we can use MLflow nested runs with Ray Tune for our project. Earlier, we defined the training and validation sets. Let's now go through the model logic. You can find all the code in the *chapter3_2.model_finetuning.py* notebook.

The `LightGBMModel` class is defined in the `models` module of the `hotel_booking` package:

```
from datetime import datetime

import mlflow
import pandas as pd
import pyspark
from lightgbm import LGBMRegressor
from loguru import logger
from mlflow import MlflowClient
from mlflow.models import infer_signature
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

from hotel_booking.config import ProjectConfig, Tags


class LightGBMModel:
    """A basic model class for house price prediction using LightGBM.

    This class handles data loading, feature preparation,
    model training, and MLflow logging.
    """

    def __init__(self, config: ProjectConfig) -> None:
        """Initialize the LightGBM model."""
        self.cfg = config
        self.cat_features = self.cfg.cat_features
        self.parameters = self.cfg.parameters.model_dump()
```

In the `train` method, we define a `sklearn` pipeline that consists of two steps: encoding categorical variables using the custom encoder `CatToIntTransformer` and `LGBMRegeressor`:

```
def train(
```

```
    self, X_train: pd.DataFrame, y_train: pd.Series, parameters: dict = None
) -> None:
    """Prepare features and train the model.

    :param X_train: Training features as a DataFrame
    :param y_train: Training target as a Series
    """

    class CatToIntTransformer(BaseEstimator, TransformerMixin):
        """
        Transformer that encodes categorical columns as integer codes
        for LightGBM.
        Unknown categories at transform time are encoded as -1.
        """

        def __init__(self, cat_features: list[str]) -> None:
            self.cat_features = cat_features
            self.cat_maps_ = {}

        def fit(self, X: pd.DataFrame, y=None) -> None:
            self.fit_transform(X)
            return self

        def fit_transform(self, X: pd.DataFrame, y=None) -> pd.DataFrame:
            X = X.copy()
            for col in self.cat_features:
                c = pd.Categorical(X[col])
                # Build mapping: {category: code}
                self.cat_maps_[col] = dict(
                    zip(c.categories, range(len(c.categories)), strict=False))
                X[col] = (
                    X[col]
                    .map(lambda val, col=col: self.cat_maps_[col].get(val,
-1))
                    .astype("category"))
            return X

        def transform(self, X: pd.DataFrame) -> pd.DataFrame:
            X = X.copy()
            for col in self.cat_features:
                X[col] = (
                    X[col]
                    .map(lambda val, col=col: self.cat_maps_[col].get(val,
-1))
                    .astype("category"))
            return X

    if parameters is not None:
```

```
            self.parameters = parameters
        preprocessor = ColumnTransformer(
            transformers=[
                ("cat", CatToIntTransformer(self.cat_features),
    self.cat_features)],
            remainder="passthrough")
        self.pipeline = Pipeline(
            steps=[
                ("preprocessor", preprocessor),
                ("regressor", LGBMRegressor(**self.parameters))])
        logger.info("Starting training...")
        self.pipeline.fit(X_train, y_train)
```

LightGBM supports integer-encoded categorical features, which often perform better than one-hot encoding; consult the LightGBM documentation for more details. We also need a custom encoder to make sure the LightGBM model treats integer-encoded features as categorical features and earlier unseen categories get assigned a value of –1 to prevent errors while computing predictions.

Note that the `CatToIntTransformer` class is defined inside the train method of the `LightGBMModel` class. While this isn't ideal from a design standpoint, it helps avoid the need to make the `hotel_booking` package available when serving the model. This keeps the model self-contained and easier to deploy. We'll explore better ways to handle private dependencies when we discuss custom `pyfunc` models later.

Now that we've defined the model class and covered the relevant methods, let's take a look at how Ray Tune works. Figure 3-5 shows the six main components of Ray Tune.[1] We begin by defining the search space, what objective to tune, the search algorithm, and when to stop. Then, we run the experiments and analyze the results.

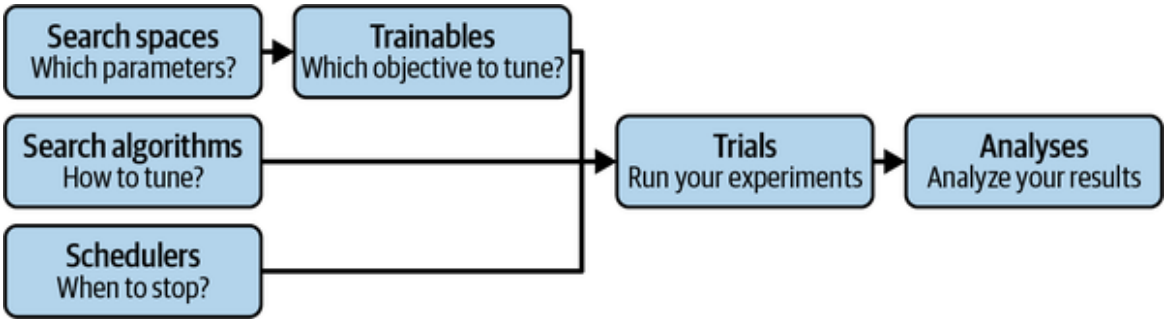


Figure 3-5. Key components of Ray Tune (adapted from Ray documentation)

Continuing with the *chapter3_2.model_finetuning.py* notebook, we define three hyperparameters to tune: `n_estimators`, `max_depth`, and `learning_rate`. Ray Tune supports a large variety of search space functions, and here, we use `tune.choice()` to sample `max_depth` uniformly from the specified choices, `tune.loguniform` to sample `learning_rate`, and `tune.qrandint` to sample `n_estimators`:

```
from ray import tune

param_space = {
    "n_estimators": tune.qrandint(50, 700, q=50),
    "learning_rate": tune.loguniform(0.01, 0.2),
    "max_depth": tune.choice([3, 5, 10, 15])}
```

The `tune.qrandint` function samples integers in fixed steps of size *q* within the specified range. For example, with interval [50, 700] and `q=50`, it draws the `n_estimators` value from {50, 100, 150, ..., 650, 700}. This is convenient because it lets us avoid wasting trials on near-duplicates.

For `max_depth`, we also provide a small set of sensible choices that are sampled using `tune.choice`. A value of –1 (unlimited) is also provided as an option. The sampling strategy for `max_depth` and `n_estimators` allows us to keep the number of unique configurations manageable, thus reducing the runtime.

Log-uniform distribution is useful for sampling the values that vary over several orders of magnitude, which makes `tune.loguniform` a perfect fit for the `learning_rate` hyperparameter.

Let's now define a *trainable*, where we pass the `train_with_nested_mlflow` function. Within that function, we start a nested run that takes `parent_run_id` and `experiment_id` as arguments, and then, we train the model and log the metrics. Config is passed by Ray Tune and contains the chosen hyperparameters:

```
import mlflow
import numpy as np
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

from hotel_booking.models.lightgbm_model import LightGBMModel


def train_with_nested_mlflow(config, X_train: pd.DataFrame,
                             X_valid: pd.DataFrame,
                             y_train: pd.DataFrame,
                             y_valid: pd.DataFrame,
                             project_config: ProjectConfig,
                             experiment_id: str,
                             parent_run_id: str):

    n_estimators, max_depth, learning_rate = (
        config["n_estimators"],
        config["max_depth"],
        config["learning_rate"],
    )
    with mlflow.start_run(experiment_id=experiment_id,
        run_name=f"trial_n{n_estimators}_md{max_depth}_lr{learning_rate}",
        nested=True, parent_run_id=parent_run_id
    ):
        model = LightGBMModel(config=project_config)
        model.train(
            X_train=X_train,
            y_train=y_train,
            parameters={
                "n_estimators": n_estimators,
                "max_depth": max_depth,
                "learning_rate": learning_rate
            }
        )
        y_pred = model.pipeline.predict(X_valid)
        mse = mean_squared_error(y_valid, y_pred)
        rmse = np.sqrt(mse)
        metrics = {
            "mse": mse,
```

```
            "rmse": rmse,
            "mae": mean_absolute_error(y_valid, y_pred),
            "r2_score": r2_score(y_valid, y_pred),
        }
        mlflow.log_params(config)
        mlflow.log_metrics(metrics)
        tune.report(metrics)
```

In the `train_with_nested_mlflow` function, we must establish a connection with MLflow to log metrics and parameters. To accomplish that, we must set up the `DATABRICKS_HOST` and `DATABRICKS_TOKEN` environment variables. We also need to be aware of the lifetime of the token; it should not expire before the hyperparameter tuning experiment stops. We can retrieve the Databricks token and host by using the workspace client:

```
from databricks.sdk import WorkspaceClient

w = WorkspaceClient()
os.environ["DATABRICKS_HOST"] = w.config.host
os.environ["DATABRICKS_TOKEN"] = w.tokens.create(
   lifetime_seconds=1200).token_value
```

You can set up a Ray cluster on Databricks to utilize all the worker nodes (not just the driver node) in your Databricks cluster for hyperparameter optimization. Note that this operation does not make sense on a single node cluster (your cluster must have a sufficient number of workers), and it also does not work on a serverless compute or outside of a Databricks environment.

> **NOTE**
>
> At the time of writing this book, this code is not supported on serverless compute. Therefore, we also cannot execute it in Databricks Free Edition.

You'll need to use the `ray.util.spark.setup_ray_cluster()` command and define the `RAY_ADDRESS` environment variable so that all the processes can find and connect to the same Ray cluster:

```
from ray.util.spark import setup_ray_cluster

ray_conf = setup_ray_cluster(min_worker_nodes=2, max_worker_nodes=8)
os.environ['RAY_ADDRESS'] = ray_conf[0]
```

Now, we can set the experiment, start the MLflow parent run, and define and fit the tuner. Here, we use the `tune.with_parameters()` function, which allows us to pass constant, non–search space arguments to the trainable. It will store objects like datasets in the Ray object store and retrieve them when calling the function. Notice that we pass `experiment_id`, `parent_run_id`, project configuration, and datasets for training and validation using this function.

As part of the tuner, we define the `TuneConfig`, where the search algorithm is specified. In this example, we use Optuna with its default sampler, the TPE algorithm—a Bayesian optimization method that works well for complex and mixed hyperparameter search spaces. It may be less effective for very low-dimensional or purely continuous search spaces. Consult the Optuna documentation for more details. Here, we minimize the root mean square error (RMSE) as the chosen evaluation metric:

```
from ray.tune.search.optuna import OptunaSearch

mlflow.set_experiment("/Shared/hotel-booking-finetuning")
experiment_id = mlflow.get_experiment_by_name(
   "/Shared/hotel-booking-finetuning").experiment_id

n_trials = 5

with mlflow.start_run(
   run_name=f"optuna-finetuning-{datetime.now().strftime('%Y-%m-%d')}",
   tags={"git_sha": "1234567890abcd", "branch": "main"},
   description="LightGBM hyperparameter tuning with Ray & Optuna"
) as parent_run:

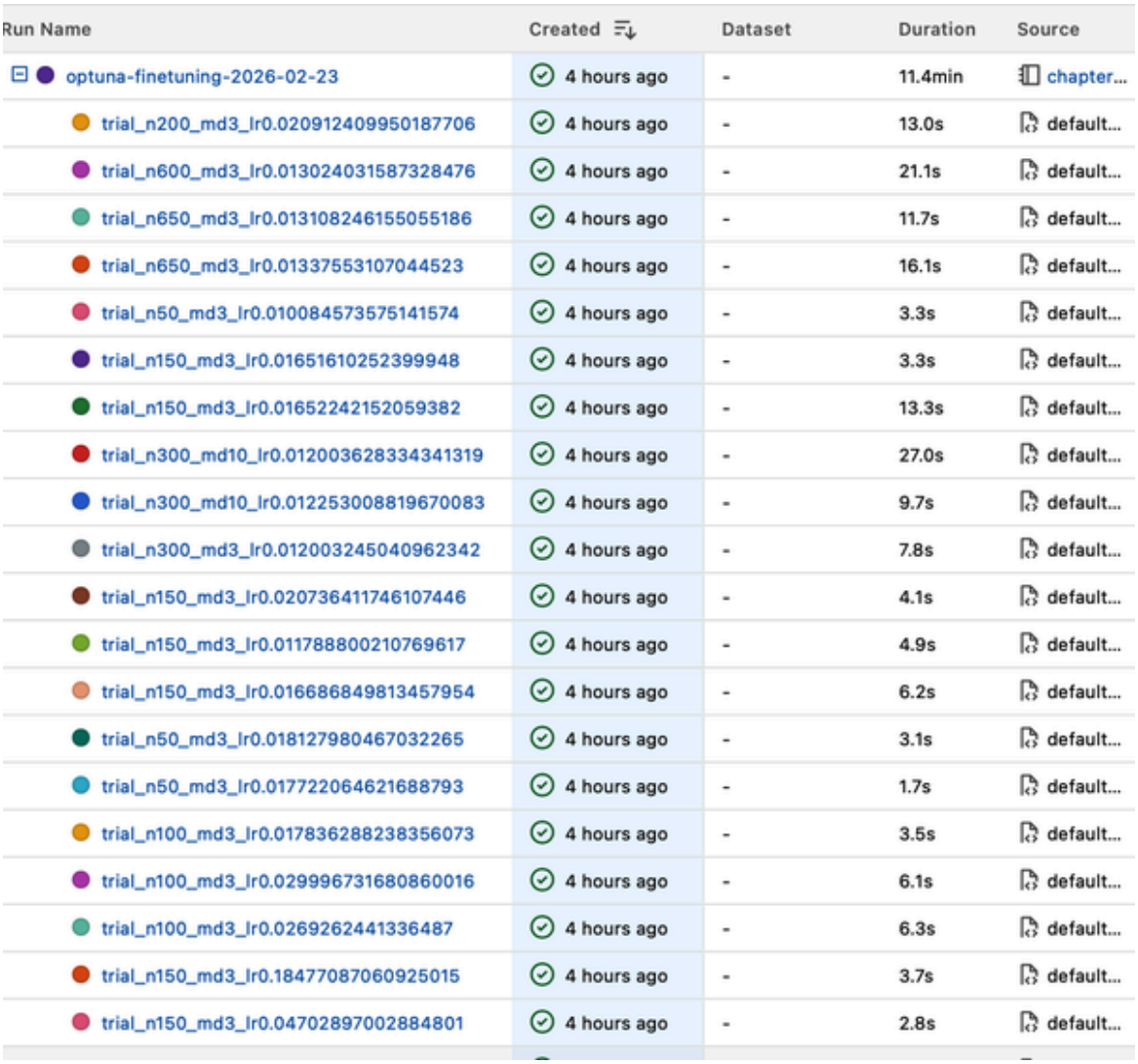
   tuner = tune.Tuner(
       tune.with_parameters(
              train_with_nested_mlflow,
              X_train = X_train,
              y_train = y_train,
              X_valid = X_valid,
```

```
                y_valid = y_valid,
                project_config=cfg,
                parent_run_id=parent_run.info.run_id,
                experiment_id=experiment_id,
            ),
        tune_config=tune.TuneConfig(
            search_alg=OptunaSearch(metric="rmse", mode="min"),
            num_samples=n_trials,
        ),
        param_space=param_space,
    )
    results = tuner.fit()
```

After we perform hyperparameter tuning, we can get the model parameters that belong to the best run. Those will be stored in the *project.config.yml* file:

```
best_result = results.get_best_result(
   metric="rmse", mode="min")
best_result.config
```

We can also use the UI to view all the nested runs under the parent run that belongs to the `hotel-booking-finetuning` experiment (see Figure 3-6).



Run Name	Created	Dataset	Duration	Source
optuna-finetuning-2026-02-23	4 hours ago	-	11.4min	chapter...
trial_n200_md3_lr0.020912409950187706	4 hours ago	-	13.0s	default...
trial_n600_md3_lr0.013024031587328476	4 hours ago	-	21.1s	default...
trial_n650_md3_lr0.013108246155055186	4 hours ago	-	11.7s	default...
trial_n650_md3_lr0.01337553107044523	4 hours ago	-	16.1s	default...
trial_n50_md3_lr0.010084573575141574	4 hours ago	-	3.3s	default...
trial_n150_md3_lr0.01651610252399948	4 hours ago	-	3.3s	default...
trial_n150_md3_lr0.01652242152059382	4 hours ago	-	13.3s	default...
trial_n300_md10_lr0.012003628334341319	4 hours ago	-	27.0s	default...
trial_n300_md10_lr0.012253008819670083	4 hours ago	-	9.7s	default...
trial_n300_md3_lr0.012003245040962342	4 hours ago	-	7.8s	default...
trial_n150_md3_lr0.020736411746107446	4 hours ago	-	4.1s	default...
trial_n150_md3_lr0.011788800210769617	4 hours ago	-	4.9s	default...
trial_n150_md3_lr0.016686849813457954	4 hours ago	-	6.2s	default...
trial_n50_md3_lr0.018127980467032265	4 hours ago	-	3.1s	default...
trial_n50_md3_lr0.017722064621688793	4 hours ago	-	1.7s	default...
trial_n100_md3_lr0.017836288238356073	4 hours ago	-	3.5s	default...
trial_n100_md3_lr0.029996731680860016	4 hours ago	-	6.1s	default...
trial_n100_md3_lr0.026926244133648 7	4 hours ago	-	6.3s	default...
trial_n150_md3_lr0.18477087060925015	4 hours ago	-	3.7s	default...
trial_n150_md3_lr0.04702897002884801	4 hours ago	-	2.8s	default...

Figure 3-6. View of the nested run in the UI

It’s also possible to plot and compare metrics that belong to different child runs (see Figure 3-7).

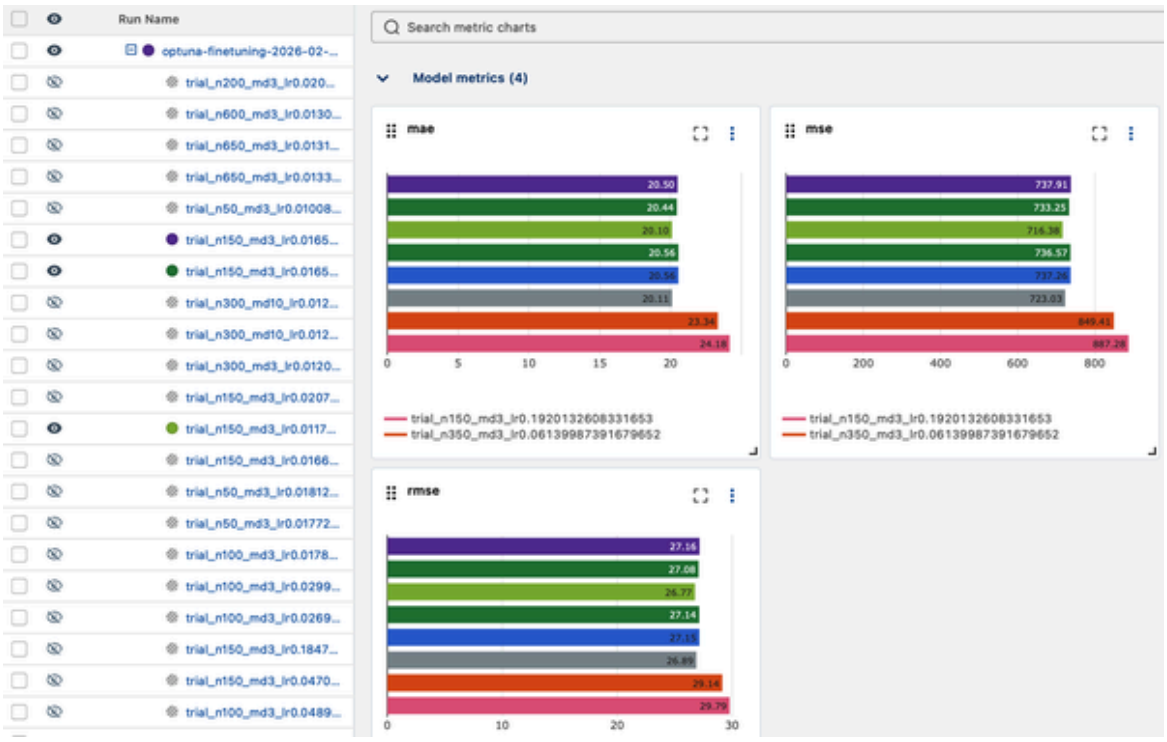


Figure 3-7. Comparing metrics across child runs

At this point, we have gone through experiment tracking, which is the backbone of everything that happens within MLflow. We've logged metrics, parameters, and various artifacts—but we have not yet logged a model, which is the next logical step after experiment tracking.

Logging and Registering Models with MLflow

Once you've tracked your experiments and identified a high-performing run, your next critical step is to save and package the trained model for future use. To let you go beyond simply saving a file, MLflow provides a standardized format called an MLflow Model. This convention defines how a model, its dependencies, and its code are stored. This is essential for downstream tasks like real-time serving, which is a topic we'll cover in Chapter 4.

MLflow supports a wide range of flavors "out-of-the-box." Here are some popular ones:[2]

- `mlflow.lightgbm`
- `mlflow.prophet`
- `mlflow.pytorch`
- `mlflow.sklearn`
- `mlflow.spacy`
- `mlflow.tensorflow`
- `mlflow.xgboost`

You can log a model by using the `mlflow.<model_flavor>.log_model()` function.

To demonstrate logging, we'll use a LightGBM model, even though we've covered the specific model code previously. We will log this model with the `sklearn` flavor rather than `lightgbm` because it's integrated within a scikit-learn pipeline.

Model Logging with the sklearn Flavor

Much like how we defined the training and validation split, we'll now define the train and test sets. The train set now contains 12 months of data, ending on 2018-11-30, while the test set will contain data exclusively from December 2018. Additionally, we'll use the latest version of the `hotel_booking` table:

```
import json
from datetime import datetime

import mlflow
from mlflow import MlflowClient
from mlflow.models import infer_signature
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig, Tags
from hotel_booking.data.data_loader import DataLoader
from hotel_booking.models.lightgbm_model import LightGBMModel
```

```
from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()
cfg = ProjectConfig.from_yaml(config_path="../project_config.yml")

spark = SparkSession.builder.getOrCreate()
data_loader = DataLoader(spark=spark, config=cfg)
X_train, y_train, X_test, y_test = data_loader.split()
```

Later on in the script, we use these:

- `train_query` and `test_query`, which contain a query that's used to retrieve the data for training and testing (attributes of the data loader object)
- The `train_set_spark` and `test_set_spark` PySpark DataFrames (attributes of the data loader object)
- `X_train`, `y_train`, `X_test`, and `y_test`

We begin by training the model, using the same method we used earlier during hyperparameter tuning:

```
model = LightGBMModel(config=cfg)
model.train(X_train=X_train, y_train=y_train)
```

Then, we set the MLflow experiment, start an MLflow run, and log parameters:

```
mlflow.set_experiment("/Shared/hotel-booking-training")
tags = Tags(**{"git_sha": "1234567890abcd", "branch": "main"})

run = mlflow.start_run(
   run_name=f"lightgbm-training-{datetime.now().strftime('%Y-%m-%d')}",
   description="LightGBM model training",
   tags=tags.to_dict(),
)
run_id = run.info.run_id
mlflow.log_params(cfg.parameters)
```

Next, we want to proceed with logging the model, and to be able to register it later in Unity Catalog, we need to provide a model signature that defines the structure of the model input and output. The easiest way to do this is to us the `infer_signature` function from the `mlflow.models` module:[3]

```
signature = infer_signature(
   model_input=X_test, model_output=model.pipeline.predict(X_test)
)
```

To ensure full traceability and reproducibility, we must log the model input. To do this, we pass a PySpark dataframe and a specific SQL query that also contains the delta table version:

```
training = mlflow.data.from_spark(
   df=data_loader.train_set_spark,
   sql=data_loader.train_query
)
testing = mlflow.data.from_spark(
   df=data_loader.test_set_spark,
   sql=data_loader.test_query
)
mlflow.log_input(training, context="training")
mlflow.log_input(testing, context="testing")
```

Now, we can finally log the model. Here, we also provide an input example, which we can also use as an example for Model Serving inference later:

```
model_info = mlflow.sklearn.log_model(
   sk_model=model.pipeline,
   name="lightgbm-pipeline",
   signature=signature,
   input_example=X_test[0:1]
  )
```

Notice that we have not computed and logged any model metrics yet. This time, we'll use MLflow Evaluation to compute the metrics. The `mlflow.models.evaluate()` function takes the model URI, evaluation data, target, model type (`"regressor"` and `"classifier"` are supported

values), and evaluators as inputs. Here, we'll use the default evaluator, but you can define custom evaluators as well.

The `mlflow.models.evaluate()` function uses an active run to log metrics (or starts a new run if no active run is available):

```
eval_data = X_test.copy()
eval_data[project_config.target] = y_test

result = mlflow.models.evaluate(
   model_info.model_uri,
   eval_data,
   targets=project_config.target,
   model_type="regressor",
   evaluators=["default"],
  )
mlflow.end_run()
```

For a regression task, the following default metrics are logged:

- `score`
- `example_count`
- `mean_absolute_error`
- `mean_squared_error`
- `root_mean_squared_error`
- `sum_on_target`
- `mean_on_target`
- `r2_score, max_error`
- `mean_absolute_percentage_error`

For classification tasks, a different set of default metrics is logged, such as these:

- `precision_score`

- `recall_score`
- `f1_score`
- `roc_auc`

In this book, we focus on demonstrating a regression task and will not cover classification examples. However, the underlying processes remain the same: training runs are tracked in MLflow, models and datasets can be logged, and evaluation metrics are automatically captured to help compare and select the best-performing models.

Viewing the Logged Model

We can now view our logged model that belongs to the `hotel-booking-training` experiment in the Models tab in the UI . We can view all the computed metrics and logged parameters in the Overview tab (see Figure 3-8).

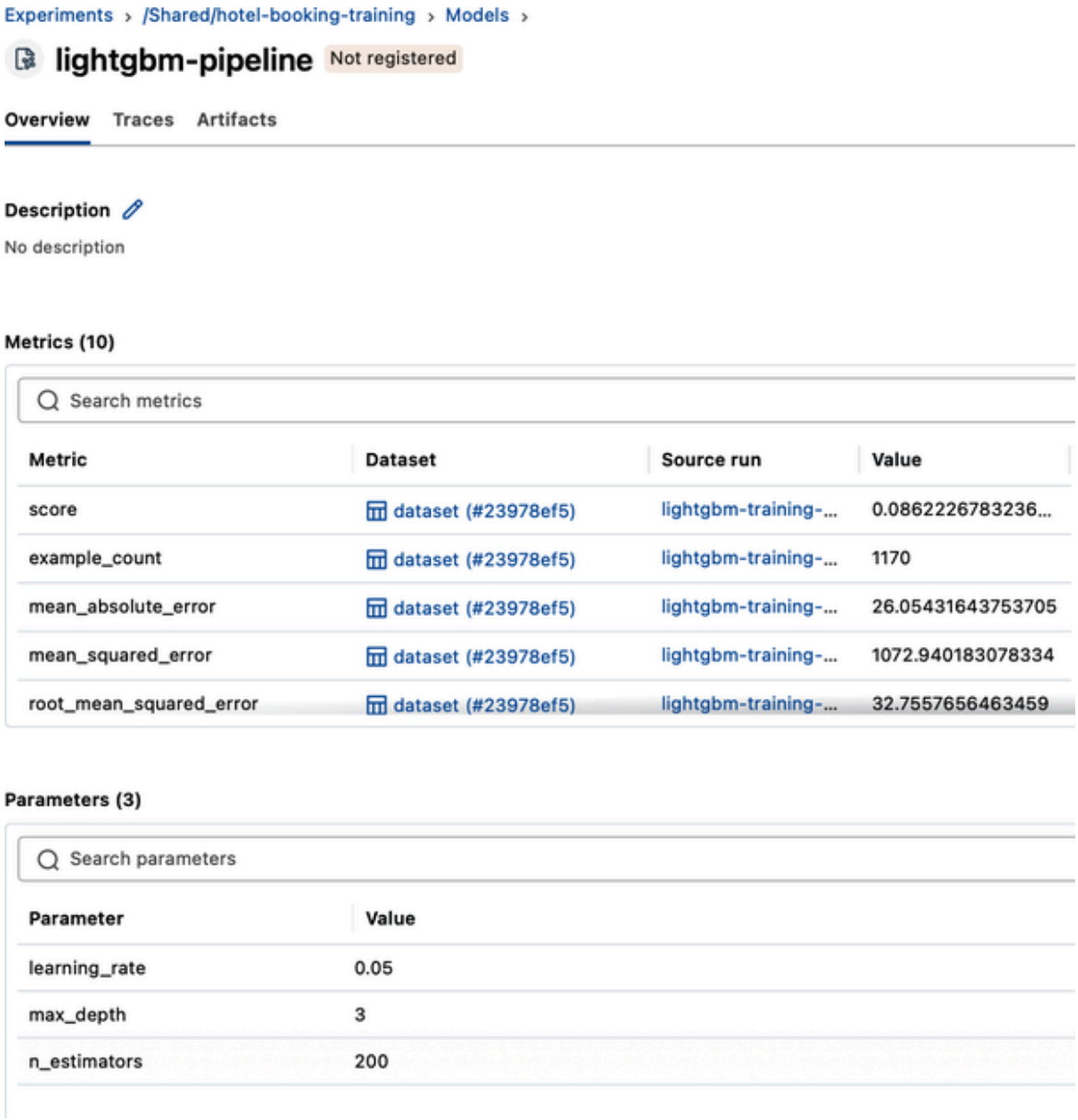


Figure 3-8. Logged model view in the UI

We can find logged artifacts in the Artifacts tab. Under "lightgbm-pipeline," we can find the *MLmodel* file that defines the flavors we can use the model with. There is always a `python_function` flavor that describes how to run the model as a Python function, even if the model is logged with a `sklearn` flavor. The `python_function` flavor is used for Model Serving. In our case, here's how we log the flavors in the *MLmodel* file:

```
flavors:
  python_function:
```

```
    code: code
    env:
      conda: conda.yaml
      virtualenv: python_env.yaml
    loader_module: mlflow.sklearn
    model_path: model.pkl
    predict_fn: predict
    python_version: 3.12.12
  sklearn:
    code: code
    pickled_model: model.pkl
    serialization_format: cloudpickle
    sklearn_version: 1.7.0
```

Fields in the *MLmodel* file include `artifact_path`, runtime information (if available), the MLflow version, `model_id`, the MLflow `run_id`, the model signature in JSON format, and the timestamp of when the model was logged. All this information is also available in the `ModelInfo` object that gets returned when we run the `mlflow.log_model` command.

Next to the *MLmodel* files, we'll find the following logged artifacts:

model.pkl

> This is the serialized model object.

conda.yaml

> This file defines the conda environment: when logging a model, we can pass a conda environment parameter with (additional) model dependencies.

python_env.yaml

> This file contains information that we need to restore a model environment using virtualenv: Python version, build dependencies, and reference to the project requirements.

requirements.txt

> This file contains inferred project requirements. It's the pip portion of the *conda.yml* file.

input_example.json

: This file contains a model input example.

serving_input_example.json

: This file contains a sample payload that we can use to send requests once we've deployed the model behind an endpoint.

LoggedModel

: This object was introduced with MLflow 3. It allows users to interact with the logged model outside of the experiment run. We've added these functions:

`mlflow.get_logged_model()`

: This allows us to retrieve a logged model by its model_id.

`mlflow.last_logged_model()`

: This allows us to retrieve the most recently logged model.

`mlflow.search_logged_models()`

: This allows us to search for logged models, in much the same way we can search for experiment runs.

Let's retrieve the model using the `mlflow.get_logged_model()` function:

```
logged_model = mlflow.get_logged_model(model_info.model_id)
```

We can now inspect the `LoggedModel` object by saving all class attributes in a JSON file:

```
import json
logged_model_dict = logged_model.to_dictionary()
logged_model_dict["metrics"] = [x.__dict__ for x in
```

```
logged_model_dict["metrics"]]
with open("../demo_artifacts/logged_model.json", "w") as json_file:
    json.dump(logged_model_dict, json_file, indent=4)
```

And then, we can visualize it with JSON Lens or a similar VS Code extension (see Figure 3-9).

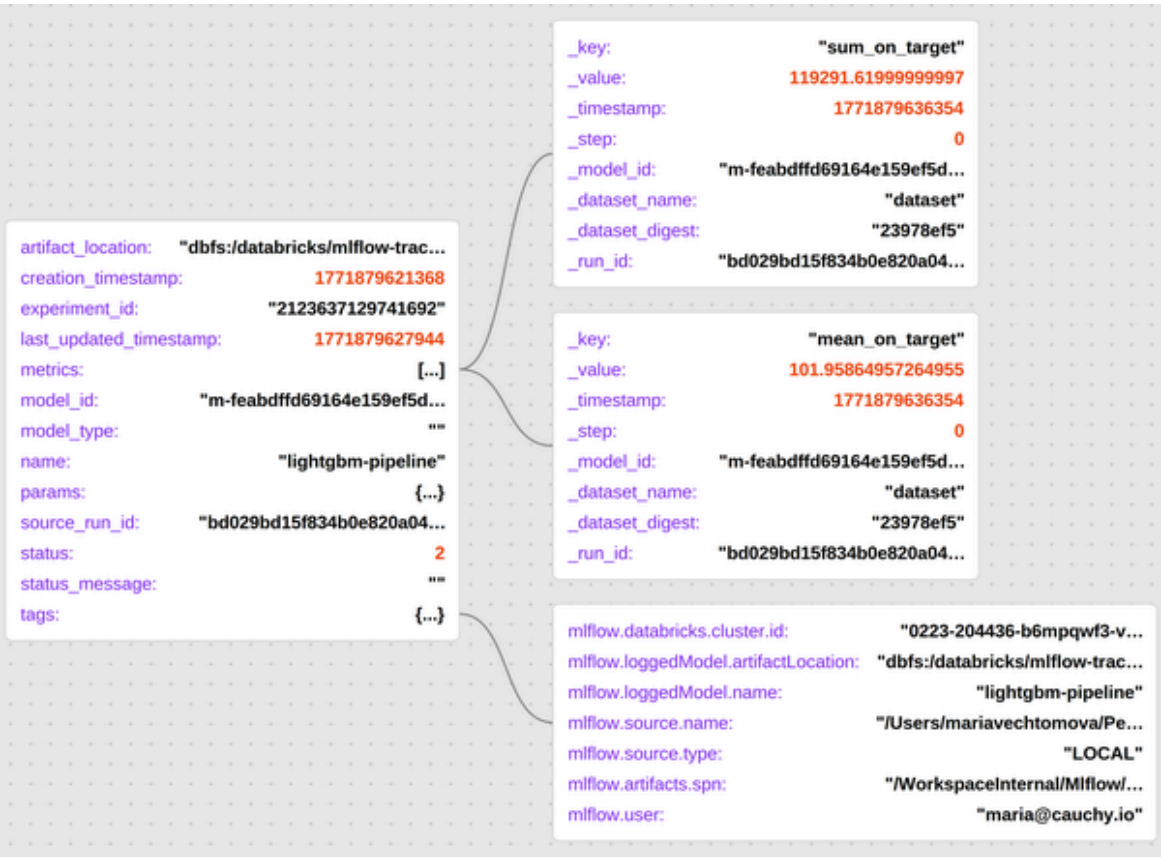


Figure 3-9. Visualizing the LoggedModel class

Notice that model metrics and model parameters are now attributes of the `LoggedModel` class, so we can access it in the following way:

```
logged_model.params
logged_model.metrics
```

Previously, it was only possible to retrieve metrics and parameters through experiment runs. Similarly, we can now also load a model by passing a `model_id` in the model URI (which we could previously only do by using the `run_id`):

```
model = mlflow.sklearn.load_model(f"models:/{model_info.model_id}")
```

While logging the model, we've also logged the input train and test datasets. We can load the logged datasets at any point later, as long as the table version used for logging is within the retention period:

```
run = mlflow.get_run(run_id)

inputs = run.inputs.dataset_inputs
training_input = next(
    (x for x in inputs if len(x.tags) > 0 and x.tags[0].value == 'training'),
     None)
training_source = mlflow.data.get_source(training_input)
training_source.load()

testing_input = next(
    (x for x in inputs if len(x.tags) > 0 and x.tags[0].value == 'testing'),
     None)
testing_source = mlflow.data.get_source(testing_input)
testing_source.load()
```

Registering the Model

To use the logged model for Model Serving on Databricks or make it accessible to other teams, we must register the model in Databricks Unity Catalog.

We can do this by providing the `registered_model` argument to the `log_model()` function. In that case, the `ModelInfo` object will also have a `registered_model_version` attribute. However, we may not want to register the model right away without first evaluating it against the latest registered version of the model. Since no model is registered yet, let's now register a model using the `mlflow.register_model()` function:

```
model_name = (
   f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"
)
registered_model = mlflow.register_model(
   model_uri=logged_model.model_uri,
   name=model_name,
```

```
    tags=tags.to_dict(),
)
```

We'll demonstrate the evaluation process in the next section of this chapter.

We can also register the model using the MLflow client. In this case, we can pass the `deployment_job_id` as an argument for the `create_registered_model` function, which can facilitate a model approval flow using the MLflow 3 deployment jobs functionality:

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()
job_id = "1234567890abcdef" # Example job ID
registered_model = client.create_registered_model(model_name,
deployment_job_id=job_id)
```

Passing a non-existing `job_id` will result in an error, as the MLflow client checks whether the job exists in the background.

After we register the model, we can assign an alias that refers to a particular version of a registered model. We can use an alias in a model URI to simplify the way we can access the registered model.

We also find it convenient to assign the `"latest-model"` tag to the most recently registered model. We can't use the `"latest"` alias directly because it's reserved, nor can we use `"latest"` in the model URI:

```
client.set_registered_model_alias(
    name=model_name,
    alias="latest-model",
    version=registered_model.version,
)
model = mlflow.sklearn.load_model(f"models:/{model_name}@latest-model")
```

It's a good idea to use aliases, as model registry search functionality is not widely supported on Unity Catalog. For instance, it's only possible to search for model versions by name:

```
v = mlflow.search_model_versions(
   filter_string=f"name='{model_name}'")
```

We also can't search for model versions using filter strings. For instance, this would not work:

```
# not supported
v = mlflow.search_model_versions(
   filter_string="tags.git_sha='1234567890abcd'")
```

However, we can get the model version by using an alias:

```
model_version = client.get_model_version_by_alias(
   alias="latest-model", name=model_name)
```

Wrapping the Model Using pyfunc

The model signature in MLflow defines the way different interfaces interact with the model. For instance, it defines the payload of the endpoint if the model gets served with Databricks Model Serving. (We'll go into detail on that in Chapter 4.)

We previously registered a `sklearn` pipeline. If we deploy it behind an endpoint now, we will get an output in the format: `{"Predictions": [120.1234567]}`. We may want to have a custom output, if so, and a `pyfunc` model flavor will come in handy.

There are other situations in which `pyfunc` becomes important:

- When we want to use an ensemble of models trained separately and serve it behind an endpoint
- When Model Serving requires a specific context (for instance, a file that defines what predictions must be adjusted)
- When we need to access other systems (for example, a vector database) to return predictions

Essentially, we are using `pyfunc` as a wrapper. Keeping the definition of the payload separate from the model itself is quite useful: we can easily adjust the `pyfunc` wrapper definition without touching the registered model itself. In a certain sense, it's very similar to the functionality of a FastAPI wrapper.

We defined the `HotelBookingModelWrapper` class under the `models` module of the `hotel_booking` package. It loads a `sklearn` pipeline as part of the `load_context()` method and modifies the predictions. Here, we are adding a 5% commission on top of the price and rounding the total price down to two decimals:

```
from datetime import datetime

import mlflow
import numpy as np
import pandas as pd
from mlflow import MlflowClient
from mlflow.models import infer_signature
from mlflow.models.model import ModelInfo
from mlflow.pyfunc import PythonModelContext
from mlflow.utils.environment import _mlflow_conda_env

from hotel_booking.config import Tags


def adjust_price(price: float) -> float:
    """Adjust the price by a fixed percentage."""
    return round(price * 1.05, 2)

class HotelBookingModelWrapper(mlflow.pyfunc.PythonModel):

    def load_context(self, context: PythonModelContext) -> None:
        self.model = mlflow.sklearn.load_model(
            context.artifacts["lightgbm-pipeline"]
        )

    def predict(self, context: PythonModelContext,
                model_input: pd.DataFrame | np.ndarray) -> dict:
        preds = self.model.predict(model_input)
        return {"Total price per night": [adjust_price(pred) for pred in
preds]}
```

Notice that the `adjust_price` function is defined outside of the class, and it will not be accessible to the model at the serving step unless we log our custom package using `code_paths` and `conda_env`. This is the cleanest way to log the package and make sure all the modules are accessible, no matter how your code is structured.

The `HotelBookingModelWrapper` class has the `log_register_model()` method. It requires the `ModelInfo` object from the model we are going to wrap, the model name we'll use to register the `pyfunc` model, the experiment name, tags, and `code_paths`:

```
    def log_register_model(self, wrapped_model_info: ModelInfo,
                           pyfunc_model_name: str,
                           experiment_name: str, tags: Tags,
                           code_paths: list) -> None:
        """Log and register the Pyfunc model."""
        wrapped_model_uri = wrapped_model_info.model_uri
        input_example_dict = mlflow.artifacts.load_dict(
            f"{wrapped_model_info.artifact_path}/input_example.json")
        input_example = pd.DataFrame(input_example_dict['data'],
                                     columns=input_example_dict['columns'])
        additional_pip_deps = []
        for package in code_paths:
            whl_name = package.split("/")[-1]
            additional_pip_deps.append(f"code/{whl_name}")
        conda_env = _mlflow_conda_env(additional_pip_deps=additional_pip_deps)

        mlflow.set_experiment(experiment_name=experiment_name)
        ts = datetime.now().strftime('%Y-%m-%d')
        with mlflow.start_run(run_name=f"wrapper-lightgbm-{ts}",
            tags=tags.to_dict()):

            signature = infer_signature(
                model_input=input_example,
                model_output={'Total price per night': [100.00]})
            self.model_info = mlflow.pyfunc.log_model(
                python_model=self,
                name="pyfunc-wrapper",
                artifacts={
                    "lightgbm-pipeline": wrapped_model_uri},
                signature=signature,
                code_paths=code_paths,
                conda_env=conda_env,
```

```
            input_example=input_example
        )

    client = MlflowClient()
    registered_model = mlflow.register_model(
            model_uri=self.model_info.model_uri,
            name=pyfunc_model_name,
            tags=tags.to_dict())
    client.set_registered_model_alias(
        name=pyfunc_model_name,
        alias="latest-model",
        version=registered_model.version,
    )
    return registered_model.version
```

There are a couple of points to pay attention to. First of all, we infer the signature using the input example from the basic model. When logging the model, we need to pass `code_paths` and `conda_env` arguments. The `code_paths` list contains a local path to the `hotel_booking` package wheel, and `conda_env` contains the location of the wheel in the artifacts folder (it will be saved in the code folder).

To provide the `code_paths`, we had to build a wheel using the `uv build` command first. We can pass the wheel's location as an argument if we run this code as part of a pipeline we deployed through a DAB. The bundle automatically builds and uploads the wheel to a location within its root folder:

```
__version__ = version("hotel_booking")
code_paths = [f"../dist/hotel_booking-{__version__}-py3-none-any.whl"]
```

Finally, we introduced Pydantic validation for tags. Now, let's initialize, log, and register the wrapper:

```
from hotel_booking.models.pyfunc_model_wrapper import HotelBookingModelWrapper

wrapper = HotelBookingModelWrapper()
pyfunc_model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_pyfunc"

wrapper.log_register_model( #Defined earlier when logging model with sklearn
flavor
```

```
    wrapped_model_info=model_info,
    pyfunc_model_name=pyfunc_model_name,
    experiment_name="/Shared/hotel-booking-wrapper",
    tags=tags,
    code_paths=code_paths,
)
```

A `pyfunc` wrapper works in the following way:

- When logging the model, we provide an artifacts argument, which is a dictionary containing the paths of artifacts needed for the model.
- The artifacts dictionary is used in the `load_context()` method, which runs at the `mlflow.pyfunc.log_model()` step. The `load_context()` method downloads the referenced artifacts and stores them together with the wrapper (see Figure 3-10).

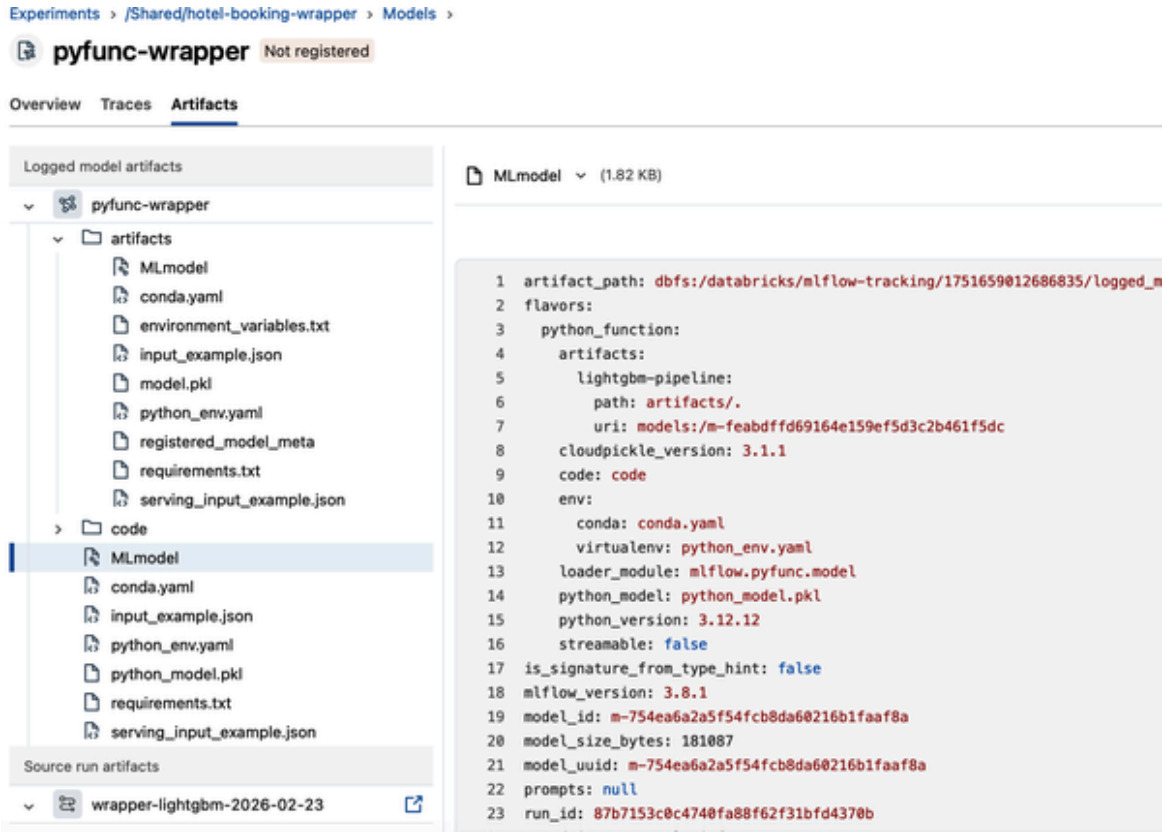


Figure 3-10. UI view of a logged `Pyfunc` wrapper model

The top-level *MLmodel* file has one flavor (pyfunc), and it includes a reference to the wrapped model artifact location:

```
flavors:
  python_function:
    artifacts:
      lightgbm-pipeline:
        path: artifacts/.
        uri: models:/mlops_dev.hotel_booking.hotel_booking_basic@latest-model
```

If we load the model, we will get back an object of the `PythonModel` type:

```
loaded_pyfunc_model = mlflow.pyfunc.load_model(pyfunc_model.model_uri)
```

When we load the model, we can use the `predict` function without passing the context:

```
loaded_pyfunc_model.predict(X_test[0:1])
```

However, this is not the `HotelBookingModelWrapper` object we logged. To get our object back and be able to access underlying attributes, we need to unwrap the loaded model:

```
unwrapped_model = loaded_pyfunc_model.unwrap_python_model()
```

To run the `predict` function, we need to pass context here (just with a value of None):

```
unwrapped_model.predict(context=None, model_input=X_test[0:1])
```

Earlier in the chapter, we mentioned that we do not want to register models right away every time we log them. First, we need to make sure that the model has actually improved since the previous iteration, which leads us to the evaluation step.

MLflow Evaluation

To evaluate the impact of updated data and compare a newly trained model with the previous one, we need additional data. Since we are working with a static dataset, we'll simulate the arrival of new data by generating a synthetic dataset that represents a full month of hotel bookings.

The `DataProcessor` class includes a `generate_synthetic_df()` method, which takes the number of observations to generate and an optional `max_date` parameter that indicates the timestamp after which the new data should be created. If `max_date` is not provided, it is retrieved from the `hotel_booking` table.

We use the `sdv` package and the `GaussianCopulaSynthesizer` to generate data with a distribution similar to that of the original dataset, and then, we adjust the reservation and arrival dates:

```
import calendar
import datetime
import random
from datetime import datetime, timedelta

import pandas as pd
from dateutil.relativedelta import relativedelta
from pyspark.sql import SparkSession
from sdv.metadata import Metadata
from sdv.single_table import GaussianCopulaSynthesizer

from hotel_booking.config import ProjectConfig


class DataProcessor:
    """A class for preprocessing and managing DataFrame operations.

    This class handles data preprocessing and saving to UC tables.
    """

    def __init__(self: "DataProcessor",
                 df: pd.DataFrame,
                 config: ProjectConfig,
                 spark: SparkSession) -> None:
        self.df = df
```

```
        self.cfg = config
        self.spark = spark
        self.target_table = f"{self.cfg.catalog}.
{self.cfg.schema}.hotel_booking"

    def preprocess(self: "DataProcessor") -> None:
        """Preprocess the DataFrame."""
        self.df.columns = self.df.columns.str.replace(r"[ -]", "_", regex=True)
       # Code to clean the data
        self.df["arrival_date"] = self.df["date_of_reservation"] +
pd.to_timedelta(
           self.df["lead_time"], unit="d")
        self.df["arrival_month"] = self.df["arrival_date"].dt.month

    def generate_synthetic_df(self: "DataProcessor",
                              cn: int = 1000, max_date: datetime = None) ->
None:
       """Preprocess the DataFrame stored in self.df."""
       # Code to generate the data

    def save_to_catalog(self: "DataProcessor") -> None:
        """Preprocess the DataFrame stored in self.df."""
        self.spark.createDataFrame(self.df).write.mode("append").saveAsTable(
            f"{self.target_table}")
        self.spark.sql(
        f"""ALTER TABLE {self.target_table} SET TBLPROPERTIES
            (delta.enableChangeDataFeed = true);"""
        )
```

For implementation details, refer to the `hotel_booking.data.data_processor` module.

Now, take a look at the notebook called *chapter3_4.model_evaluation.py*. We start by generating synthetic data and storing it via Unity Catalog:

```
import pandas as pd
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig, Tags
from hotel_booking.data import DataLoader, DataProcessor

from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()
spark = SparkSession.builder.getOrCreate()
```

```
cfg = ProjectConfig.from_yaml("../project_config.yml")

# Load and process the data
df = pd.read_csv("../data/booking.csv")
data_processor = DataProcessor(df=df, config=cfg, spark=spark)
data_processor.preprocess()
data_processor.generate_synthetic_df(n=1000, max_date=None)
data_processor.save_to_catalog()
```

Then, we load the data (where the test set includes the newly generated month of data and the train set 12 months before) and train the model:

```
from hotel_booking.models.lightgbm_model import LightGBMModel

data_loader = DataLoader(spark=spark, config=cfg)
X_train, y_train, X_test, y_test = data_loader.split()

model = LightGBMModel(config=cfg)

model.train(X_train=X_train,
            y_train=y_train)
```

The code for model logging will then be added to the `LightGBMModel` class. This is how we can be use it:

```
tags=Tags(**{"git_sha": "1234567890abcd", "branch": "main"})

model_info = model.log_model(
   experiment_name="/Shared/hotel-booking-training",
   tags=tags,
   X_test=X_test,
   y_test=y_test,
   train_set_spark=data_loader.train_set_spark,
   train_query=data_loader.train_query,
   test_set_spark=data_loader.test_set_spark,
   test_query=data_loader.test_query
   )
```

We can access metrics of the newly logged model through the model object:

```
metrics_new = model.metrics
```

Let’s now evaluate the latest currently registered model against the new test set:

```
import mlflow

sklearn_model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"
model_uri = f"models:/{sklearn_model_name}@latest-model"
eval_data = X_test.copy()
eval_data[cfg.target] = y_test

result = mlflow.models.evaluate(
       model_uri,
       eval_data,
       targets=cfg.target,
       model_type="regressor",
       evaluators=["default"],
   )
metrics_old = result.metrics
```

Previously, we were evaluating model performance based on the RMSE. If this metric for the new model is lower than for the old model, we can proceed with registering the model:

```
if metrics_new['root_mean_squared_error'] <
metrics_old['root_mean_squared_error']:
   model.register_model(model_name=sklearn_model_name, tags=tags)
```

The logic for registering the model is then added to the `LightGBMModel` class:

```
def register_model(self: "LightGBMModel", model_name: str, tags: Tags) ->
None:
      """Register the model in MLflow Model Registry."""
      client = MlflowClient()
      registered_model = mlflow.register_model(
              model_uri=self.model_info.model_uri,
              name=model_name,
              tags=tags.to_dict(),
          )
      client.set_registered_model_alias(
          name=model_name,
          alias="latest-model",
```

```
        version=registered_model.version,
    )
    return registered_model.version
```

Now that we've explored MLflow experiment tracking, model logging, and model registration in depth, you need to be aware of the resource and API limits that apply, which may influence the way you organize your workspaces and experiments.

Resource and API Limits

As you scale your ML workflows on Databricks, you may encounter platform limits. For example, when you're running extensive hyperparameter tuning across multiple projects within the same workspace, each experiment can generate hundreds or thousands of runs that log metrics, parameters, artifacts, and models to MLflow. Over time, this can lead to throttling or failures if you exceed API rate limits or resource quotas.

For instance, distributed hyperparameter tuning with Ray Tune across several teams in a shared workspace can quickly approach the 500,000 runs per MLflow experiment limit.

In retail demand forecasting scenarios, in which you train a separate model in each product category, you'll commonly log hundreds or even thousands of models within a single experiment. In such cases, you may reach the 500,000 logged models per MLflow experiment limit if you do not clean it in a timely manner.

Understanding these constraints helps you plan more robust, scalable systems. Table 3-1 outlines some of the key resource limits you should be aware of when using MLflow for model development and Unity Catalog for model registration.

Table 3-1. Resource limits for experiment tracking and registered models

Resource	Limit
Experiment tracking metric steps	10 M per MLflow run
Experiment tracking parameters	1,600 per MLflow run/ logged model
Experiment tracking tags	1,600 per MLflow run/ logged model; 1,000 per experiment
Experiment tracking runs	500 k per MLflow experiment
Experiment tracking logged models	500 k per MLflow experiment
Experiment tracking metrics	1,000 per MLflow run/ logged model
Registered model (Unity Catalog)	1,000 per schema; 5000 per metastore
Registered model alias (Unity Catalog)	10 per registered model
Model versions (Unity Catalog)	10 k per registered model; 100k per metastore

Table 3-2 outlines some of the key API limits you should be aware of when using MLflow.

Table 3-2. MLflow API limits

API	Limit
*/mlflow/experiments/** (except list and search) *mlflow/runs/* (except search) */mlflow/metrics/get-history* */mlflow/artifacts/list* */mlflow/logged-models/* (except search and artifacts)	120 rps per workspace
/mlflow/experiments/list */mlflow/experiments/search* */mlflow/runs/search* */mlflow/logged-models/search*	7 rps per workspace
/mlflow/logged-models/{model_id}/artifacts/directories	2 rps per workspace

For the full, up-to-date overview of the resource and API limits, consult the Databricks documentation.

Conclusion

In this chapter, we went through the whole process of experimentation, choosing the right way to split the data for training and validation, hyperparameter tuning, logging, evaluating, and registering models.

By the end of the chapter, we created three experiments:

1. hotel-booking-finetuning
2. hotel-booking-training
3. hotel-booking-wrapper

Under each experiment, we created experiment runs with descriptive names, to make runs easy to find:

1. optuna-finetuning-yyyy-mm-dd
2. lightgbm-training-yyyy-mm-dd
3. wrapper-lightgbm-yyyy-mm-dd

We tagged all the experiment runs and registered models with `git_sha` (for now, a dummy value) and branch, so we can find the exact code version that's responsible for creating the artifacts. We logged the reference to the exact version of training and testing data used for model training and evaluation.

All this allows us to achieve full traceability and reproducibility for our deployments. Chapter 5 will show you how to pass actual `git_sha`, `job_id`, and `run_id` as parameters to the scripts running in a production setting.

But before we can go into deployment, we need to talk about Model Serving architectures and implement a Model Serving endpoint for the use case we just went through.

---

1 For more details, check out the Ray documentation.

2 You can find the full list in the `mlflow.models` documentation.

3 To explore more detailed information about model signatures in MLflow as well as additional methods to define them, refer to the documentation.

Chapter 4. Model Serving: Architectures and Implementation

After we register the model, it will be ready for use by downstream systems. This raises the following question: how do we generate predictions and serve them to end users?

Many ML models are deployed as batch processes, where predictions are generated on a schedule (e.g., daily or hourly) and written to a database—either directly or transferred via mechanisms such as Secure File Transfer Protocol (SFTP). This database typically serves as the backend for an internal application, such as a demand forecasting or pricing system, that consumes the predictions for downstream decision making.

While implementing a full batch inference pipeline is one of the most common deployment patterns in production environments, we will not focus on it in this book. In practice, this setup would follow the same approach we demonstrated earlier when running the `predict` function in Chapter 3. However, instead of running inference on observations from the test set, we would generate predictions on fresh, unseen, and unlabeled data, with the results persisted to a table for downstream consumption by other systems or applications.

In this chapter, I'll cover situations in which ML applications need to deliver predictions in real time.

There are three main scenarios we'll cover:

Model Serving

> All the features are available to the model via the payload, and predictions are computed at the moment of request.

Serving precomputed batch predictions

> Predictions are precomputed in advance and stored in a database, and they only get retrieved at the moment of request.

Model Serving with feature lookup

> Predictions are computed at the moment of request. Some features are available via payload, and some features get retrieved at the moment of request. These features can be precomputed in a batch fashion, or they can be streaming features that are computed in real time and written to a feature store.

While I'll be covering all core deployment scenarios, I'll leave streaming features out of scope. In practice, true streaming features are rarely required for most AI applications. The majority of production systems work perfectly well with precomputed (batch) features that are refreshed on a schedule.

From an endpoint deployment perspective on Databricks, serving streaming features is almost identical to serving precomputed (batch) features. The model endpoint retrieves features in the same way, and the serving infrastructure does not fundamentally change based on how those features were produced.

The primary difference lies upstream: synchronization between the Databricks online feature store and the Delta table is continuous, rather than triggered by a batch job or scheduled process. This affects how feature freshness is maintained, but it does not change the deployment mechanics of the endpoint itself. We'll talk about these details in this chapter.

There's another important question in model deployment: who is responsible for it?

In some organizations, the same team that develops the model is also responsible for deploying and operating it in production. This can create tight feedback loops, reduce handoffs, and speed up iteration.

In other organizations, particularly larger enterprises and regulated industries, deployment is handled by dedicated ML engineering, platform, and infrastructure teams. Sometimes, this separation is mandated by risk management policies or compliance requirements. In these environments, having clearly defined responsibilities and controls is essential, and deployment ownership may intentionally sit outside the development team.

I recommend that you keep model development and production operations as close as you can. Tight feedback loops are what you are optimizing for, and the fewer handoffs there are between building a model and running it, the faster you will iterate and the easier debugging will become. Split ownership can still work, but it does not work for free. When responsibilities sit with multiple teams, you need strong tooling, shared visibility, and standardized processes to hold them together. Without these protocols, deployment slows down, communication overhead grows, and iteration cycles lengthen.

Model Serving is often challenging for data science teams, especially when they are expected to manage the deployment. This is where Databricks Model Serving can help.

With *Serverless Model and Feature Serving*, Databricks simplifies deployment infrastructure. You can create a model endpoint with just one Python command by using the Databricks SDK. This enables the following:

- Direct deployment by data science teams (when the organizational structure allows it)
- Standardization and governance of deployments by ML engineering teams at scale
- Clear ownership with less operational burden

Rather than prescribing a single “ideal” structure, the goal is to enable whichever team owns production deployment—data scientists, ML engineers, or a centralized platform team—to move quickly without compromising governance or reliability.

Let’s begin by covering how to deploy an endpoint in the simplest scenario.

Model Serving

In the first scenario, we’ll explore *Model Serving*, assuming that all required features are included in the request payload. Figure 4-1 shows a typical automated pipeline for retraining and deploying a model in Databricks. This example references two components: *Lakeflow jobs*, which handle orchestration, and *Unity Catalog* (UC), which manages governance and access. We first introduced these in Chapter 1, and we will examine them in detail in Chapter 5.

For Model Serving, we can perform both *model retraining* and *endpoint updates* within the same pipeline. A scheduled Lakeflow job runs data preprocessing and model retraining steps, and after that, the new model is registered in Unity Catalog. The pipeline then programmatically updates the real-time serving endpoint so applications always use the latest version of the model.

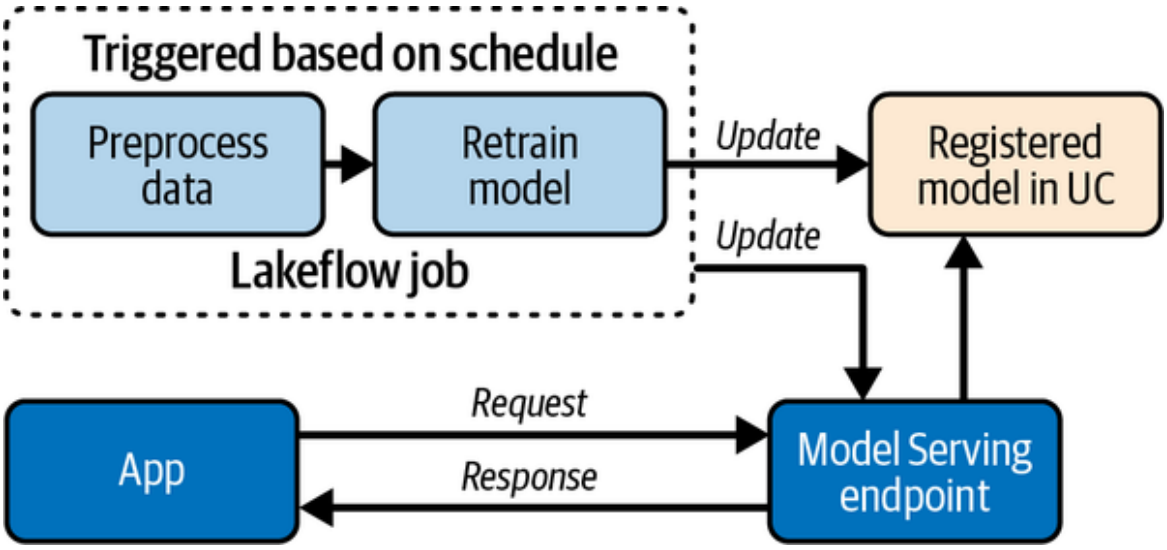


Figure 4-1. Simplified architecture for Model Serving

We will discuss this pipeline in depth in Chapter 5, but for now, let's focus on the endpoint deployment itself. Our goal is to predict the nightly price of a hotel reservation using information the user provides, like the number of children and adults, the number of weekday and weekend nights, parking spaces available, special requests, and so on.

To deploy an endpoint for this use case, we first need a registered model. So let's grab the latest version of our wrapped model:

```
import mlflow
from mlflow import MlflowClient

from hotel_booking.config import ProjectConfig
from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()

cfg = ProjectConfig.from_yaml("../project_config.yml")

catalog = cfg.catalog
schema = cfg.schema

model_name = "hotel_booking_pyfunc"

client = MlflowClient()
model_version = client.get_model_version_by_alias(
```

```
    alias="latest-model",
    name=f"{catalog}.{schema}.{model_name}")
```

Then, we'll need to define a list that consists of instances of `ServedEntityInput`. If we want to serve a model, we'll need to provide the following:

`entity_name`

: This is the full name of the model in the following format: {*catalog_name*}.{*schema_name*}.{*model_name*}.

`entity_version`

: This is the specific version of the model to serve.

`environment_vars`

: This is optional, but it's useful, for example, if we want to authenticate to another service and refer to Databricks secrets. We will use this feature in Chapter 9.

`workload_type`

: The default is a CPU, but various graphics processing unit (GPU) workload sizes are also available.

`scale_to_zero_enabled`

: This controls whether the compute resources should scale down to zero when not in use. I recommend that you set this parameter to `True` for development and `False` for production, due to a cold-start problem.

Concurrency settings

: You must define autoscaling ranges by specifying either a `workload_size` (e.g., Small, Medium, Large) or by setting

`min_provisioned_concurrency` and `max_provisioned_concurrency`.

In the following example, we'll use some of these parameters:

```
from databricks.sdk.service.serving import ServedEntityInput

served_entities = [
   ServedEntityInput(
       entity_name=f"{catalog}.{schema}.{model_name}",
       scale_to_zero_enabled=True,
       workload_size="Small",
       entity_version=model_version.version,
   )
]
```

Now, let's talk about autoscaling:

- A single unit of provisioned concurrency can process one request at a time.
- The workload size corresponds to a range of provisioned concurrency units. The valid values are: Small (4 units), Medium (8–16 units), and Large (16–64 units). We can also use additional custom workload sizes when they're available in the workspace. They can be requested on demand.
- Autoscaling is based on the number of required units, which is calculated as *queries per second* × *model processing time*.

Let's look at an example. Imagine your service needs to handle 1,000 queries per second, and the model takes 0.02 seconds to process each query. To handle this load, the serving endpoint needs to be able to process 20 queries simultaneously. So in this case, you would provision 20 units of concurrency.

Endpoint Deployment

Let's deploy an endpoint using Databricks SDK. When creating an endpoint, we can customize it with a few optional settings:

Cost monitoring

We can add tags or a usage policy ID to monitor the costs associated with the endpoint.

Route optimization

When this is enabled, a unique, optimized URL is created for the endpoint, which can lower latency. A key detail to remember is that this feature only supports OAuth authentication, so you cannot use a personal access token (one that is generated via the workspace client) for testing. While we won't enable this in our example, it's often a good idea to enable it for production use.

Databricks Unity AI Gateway configuration

AI Gateway allows you to enable the following:

- Fallback to another served entity if a served entity fails with certain error codes
- AI guardrails for foundation model APIs, to prevent unsafe or unwanted data in requests and responses (which we'll cover in Chapter 9)
- An inference table for logging all requests and responses for monitoring and audit purposes
- Rate limits that you can set to limit endpoint traffic
- Usage tracking in system tables, to monitor operational usage on endpoints

Here, we'll enable usage tracking and the inference table for our endpoint to use it for monitoring:

```
from databricks.sdk.service.serving import (
   AiGatewayConfig,
   AiGatewayInferenceTableConfig,
   AiGatewayUsageTrackingConfig,
)
ai_gateway_cfg = AiGatewayConfig(
       inference_table_config=AiGatewayInferenceTableConfig(
           enabled=True,
           catalog_name=catalog,
           schema_name=schema,
           table_name_prefix="hotel_booking_monitoring",
       ),
       usage_tracking_config=AiGatewayUsageTrackingConfig(enabled=True)
   )
```

Next, we either create a new endpoint or update an existing one. In the configuration, we can optionally define traffic routing using `TrafficConfig`. In this example, we route 100% of the traffic to a single model. Providing a traffic configuration is optional; however, we include it here for clarity and demonstration purposes. In practice, you could serve multiple entities and define multiple traffic routes, for example, by splitting traffic 50/50 between two model versions. We'll return to traffic-splitting considerations when we cover A/B testing in the next section.

Finally, we specify the usage policy ID defined in the *project_config.yml* file, which allows us to track and attribute project-related costs:

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.serving import (
   EndpointCoreConfigInput,
   Route,
   TrafficConfig,
)

w = WorkspaceClient()
endpoint_name = "hotel-booking-pyfunc"
endpoint_exists = any(
```

```
    item.name == endpoint_name for item in w.serving_endpoints.list()
)

traffic_config=TrafficConfig(
            routes=[
                Route(
                    served_model_name=f"{model_name}-{model_version.version}",
                    traffic_percentage=100)
            ]
        )

if not endpoint_exists:
    w.serving_endpoints.create(
        name=endpoint_name,
        config=EndpointCoreConfigInput(
            name=endpoint_name,
            served_entities=served_entities,
            traffic_config=traffic_config),
        ai_gateway=ai_gateway_cfg,
        budget_policy_id=cfg.usage_policy_id,
    )
else:
    w.serving_endpoints.update_config(
        name=endpoint_name, served_entities=served_entities
    )
```

Make sure the endpoint is ready before sending requests. You can check its status by using `workspace.serving_endpoints.get(name=endpoint_name)`. Alternatively, instead of the `create()` and `update_config()` methods, you can use `create_and_wait()` and `update_config_and_wait()`, which block until the operation completes and the endpoint is ready.

Once the endpoint is deployed and ready, you can start sending inference requests. Databricks Model Serving uses MLflow Model Serving under the hood, and it therefore follows the same request payload format. For models that expect a Pandas DataFrame as input rather than Tensors, you must send requests by using either the `dataframe_split` or the `dataframe_records` format, with `dataframe_split` generally being the recommended option because it guarantees that the column ordering is preserved (see Table 4-1).

Table 4-1. Payload structure for Model Serving (from the MLflow documentation)

Field	Description	Example
`dataframe_split`	A Pandas DataFrame in the split orientation	`{"dataframe_split": pandas_df.to_dict(orient="split")}`
`dataframe_records`	A Pandas DataFrame in the records orientation	`{"dataframe_records": pandas_df.to_dict(orient="records")}`
`instances`	Tensor input formatted according to TensorFlow Serving API	`{"instances": [1.0, 2.0, 5.0]}`
`inputs`	Same as instances but with a different key	`{"inputs": [["Cheese"], ["and", "Crackers"]]}`

Before we can send any requests, we need to authenticate. In the *notebooks/chapter4_1.model_serving.py* file, we can retrieve a personal access token and the host by using the workspace client:

```
host = w.config.host
token = w.tokens.create(lifetime_seconds=1200).token_value
```

You should not use personal access tokens for a production setting; instead, you must create a service principal and use its OAuth token. We'll discuss setting it up at the end of this chapter.

When calling the endpoint, we can provide the payload in two formats: `dataframe_records` and `dataframe_split`:

```
payload = {
   "dataframe_records": [{
```

```
        'number_of_adults': 2,
        'number_of_children': 0,
        'number_of_weekend_nights': 0,
        'number_of_week_nights': 1,
        'car_parking_space': 0,
        'special_requests': 0,
        'lead_time': 103,
        'type_of_meal': 'Not Selected',
        'room_type': 'Room_Type 1',
        'arrival_month': 8,
        'market_segment_type': 'Online'
    }]
}

payload = {
    "dataframe_split": {
        'columns': ['number_of_adults', 'number_of_children',
                    'number_of_weekend_nights', 'number_of_week_nights',
                    'car_parking_space', 'special_requests', 'lead_time',
                    'type_of_meal', 'room_type', 'arrival_month',
                    'market_segment_type'],
        'data': [[2, 0,
           0, 1,
           0, 0, 103,
           'Not Selected', 'Room_Type 1', 8,
           'Online']]
    }
}
```

Then, we can use the `requests` library to call the endpoint:

```
import requests

serving_endpoint = f"{host}/serving-endpoints/hotel-booking-
pyfunc/invocations"

response = requests.post(
    serving_endpoint, headers={"Authorization": f"Bearer {token}"},
json=payload,
)
```

The output contains the "`predictions`" key, and the content is defined by the PyFunc wrapper:

```
{"predictions": {"Total price per night": [104.98]}}
```

Deploying a model without proper validation can lead to costly delays. Spinning up an endpoint can take 10 minutes or more, and it's an expensive interaction cycle. Luckily, Databricks provides mechanisms for debugging the endpoint before deployment.

Databricks also supports express deployments, which significantly speed up the deployment process. It's important to note that this feature is available only within the *serverless environment*. You must log and register the model from a serverless compute, which may not be suitable for certain ML workloads, as we discussed earlier. This is, however, a great feature for AI application deployment, and we'll use it in the LLMOps chapters.

A/B Testing

In the section "Endpoint Deployment", we configured an endpoint with a `TrafficConfig` and routed all of its traffic to a single model. We noted there that you could instead split traffic across two models, say 50/50 between two versions, and that this is the simplest way to run an A/B test. It is not the only way, though, and it is not always the correct one. Before you rely on it, you should learn exactly what the split does and what it quietly fails to do.

The defining property of a traffic split is that model selection is determined independently for each request. As a result, the same customer may be routed to different model versions over the course of an experiment. This independence is what makes traffic splits easy to implement, but it is also what makes them unsuitable for certain types of experiments.

The alternative is to derive the assignment from a stable identifier for the entity you care about, such as a customer ID (in our example, the booking ID). You hash the identifier and map the hash to a bucket, so the same identifier is always assigned to the same model for the duration of the experiment. Unlike a traffic split, the assignment is deterministic: the same booking, scored twice, always gets the same model.

Choosing a routing strategy

The rule for this is simple: randomize at the same level at which you will analyze the experiment.

The experimentation literature distinguishes between the randomization unit and the analysis unit.[1] The *randomization unit* is the entity assigned to a model. The *analysis unit* is the entity on which the outcome is measured. In a well-designed experiment, the two usually match.

Let's discuss what randomization unit to choose in different scenarios.

Randomization by customer

Most experiments do this because most business outcomes are measured by customer. Conversion, retention, repeat bookings, and revenue are all customer-level metrics. If a customer sees multiple models during the experiment, their behavior will no longer be attributable to a single group, and that will make the results harder to interpret. Customer-level assignment avoids this problem and keeps the customer experience consistent throughout the experiment.

Session-level assignment

This is the middle-of-the-road approach. Use it when the outcome is defined within a visit, rather than across visits. For example, if you care about session conversion or behavior during a single browsing session, you should assign each session to one model, but you can reassign a returning customer in a later session. This is also a practical choice when no stable customer identifier exists.

Request-level assignment

This is appropriate only when requests are genuinely independent. This is typically the case for backend model evaluations in which the prediction is invisible to the customer and does not influence their future behavior. Examples include latency measurements, throughput comparisons, offline batch scoring, and accuracy evaluation of independent transactions,

documents, or events. Here, a simple traffic split is sufficient because the request itself is both the randomization unit and the analysis unit.

A useful test involves asking what entity appears in your final metric. If the metric is measured per request, you should split by request. If it is measured per session, split by session. If it is measured per customer, split by customer. The assignment mechanism should follow the unit of analysis, not the implementation convenience of a traffic router.

Moving assignment into the model logic

So far, we have shown you how you could configure traffic splitting programmatically at the endpoint level. That mechanism randomizes at the request level, which is fine when the request is both the unit of assignment and the unit of analysis

Now, consider an experiment in which the booking is the experimental unit. Every prediction associated with the same booking must be handled by the same model throughout the experiment. To assign consistently at the booking level, the assignment decision must move out of the endpoint configuration and into the model logic itself.

We will use the model we trained in the previous chapter as variant A:

```
from hotel_booking.config import ProjectConfig

cfg = ProjectConfig.from_yaml(
   config_path="../project_config.yml")

model_name_a = f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"
model_a_uri = f"models:/{model_name_a}@latest-model"
```

Next, we train a second variant with a different set of hyperparameters. In practice, the difference between variants can be much larger: a different algorithm, a different feature set, or an entirely different preprocessing pipeline. But different hyperparameters are enough to illustrate the mechanism:

```
from pyspark.sql import SparkSession

from hotel_booking.data.data_loader import DataLoader
from hotel_booking.models.lightgbm_model import LightGBMModel

spark = SparkSession.builder.getOrCreate()
data_loader = DataLoader(spark=spark, config=cfg)
X_train, y_train, X_test, y_test = data_loader.split()

model_b = LightGBMModel(config=cfg)

model_b.train(
   X_train=X_train,
   y_train=y_train,
   parameters={"learning_rate": 0.01,
               "n_estimators": 1000,
               "max_depth": 6},
)
```

Then, we log and register variant B:

```
from hotel_booking.config import Tags
from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()
tags = Tags(**{"git_sha": "1234567890abcd", "branch": "main"})

model_b.log_model(
   experiment_name="/Shared/hotel-booking-ab-testing",
   tags=tags,
   X_test=X_test,
   y_test=y_test,
   train_set_spark=data_loader.train_set_spark,
   train_query=data_loader.train_query,
   test_set_spark=data_loader.test_set_spark,
   test_query=data_loader.test_query,
)
model_name_b = f"{cfg.catalog}.{cfg.schema}.hotel_booking_ab_b"

model_b.register_model(model_name=model_name_b, tags=tags)
model_b_uri = f"models:/{model_name_b}@latest-model"
```

Now, we create a wrapper model that contains the assignment logic. The wrapper receives the booking identifier as a parameter in the request payload, hashes it using `hashlib`, converts the hex digest to an integer, and assigns the request to variant A or B based on whether the result is even or odd. The hex digest is the MD5 hash written as a 32-character base-16 string, and `int(hashed_id, 16)` reads that string back into a single large integer, where the 16 tells Python that the digits are hexadecimal so the characters a to f, which are worth 10 to 15, are valid.

Hashing is what makes the split well behaved. Raw booking identifiers often follow patterns, such as sequential numbering, whereas an MD5 hash maps them to values spread evenly across its output range. This means the odd/even outcome lands close to 50/50 and stays uncorrelated with any property of the booking. This is also deterministic, so the same booking identifier always maps to the same variant, regardless of how many requests it generates:

```
import hashlib

import mlflow
import numpy as np
import pandas as pd
from mlflow.pyfunc import PythonModelContext

class HotelBookingABTestWrapper(mlflow.pyfunc.PythonModel):
    def load_context(self, context: PythonModelContext) -> None:
        self.model_a = mlflow.sklearn.load_model(
            context.artifacts["model-a"])
        self.model_b = mlflow.sklearn.load_model(
            context.artifacts["model-b"])

    def predict(
        self,
        context: PythonModelContext,
        model_input: pd.DataFrame | np.ndarray,
        params: dict | None = None,
    ) -> dict:
        booking_id = str((params or {}).get("booking_id", ""))
        hashed_id = hashlib.md5(booking_id.encode("UTF-8")).hexdigest()
        # Hex digest to int; even/odd gives a stable 50/50 split.
        if int(hashed_id, 16) % 2:
```

```
        predictions = self.model_a.predict(model_input)
        variant = "A"
    else:
        predictions = self.model_b.predict(model_input)
        variant = "B"
    prices = [round(float(pred) * 1.05, 2) for pred in predictions]
    return {
        "Total price per night": prices,
        "model": variant,
        "booking_id": booking_id,
    }
```

Table 4-2 lists some examples of the assignment.

Table 4-2. Conversion of booking ID to hex digest and to model variant assignment

Booking ID	MD5 hex digest	int % 2	Variant
INN12345	5c8f093e729b3cccbed92a48848c3fbd	1	A
INN12346	47ee65b0ba45f45b68a093dcbfd5f6b3	1	A
INN12347	b52126aac65831a322a21c43d1e8d58a	0	B
INN12348	1047b9493a5f98b4cd8a1e7d27059aea	0	B

We log, register, and deploy the wrapper model in the same way as the custom model from the previous section. The only difference at inference time is that the request payload must include the booking identifier so the wrapper can perform the assignment. The full implementation is available in the *chapter4_1a.ab_testing_demo.py* notebook.

In practice, organizations rarely implement model assignment this way. Instead, they integrate with an experimentation platform, either built in house or provided by a third-party service, that handles assignment through an API call and supports experiment tracking and statistical analysis.

Now that we've covered deployment and A/B testing, our next practical concern is debugging. Model Serving endpoints can fail for reasons that are not always obvious, and having a systematic approach to diagnosis will save us considerable time.

Debugging

A common check involves loading the model using `mlflow.pyfunc.load_model()` and running test data through it. While this works in Visual Studio Code (VS Code) or a Databricks notebook, it doesn't fully simulate the serving environment and may miss issues related to packaging or dependencies.

A more reliable approach is to use `mlflow.models.predict()`:

```
import pandas as pd

input_example = pd.DataFrame(
   data=payload["dataframe_split"]["data"],
   columns=payload["dataframe_split"]["columns"]
)
model_uri = f"models:/{catalog}.{schema}.{model_name}@latest-model"
mlflow.models.predict(model_uri, input_example)
```

Unlike `load_model()`, it builds a lightweight virtual environment based on the `conda_env` defined in your model's configuration. This makes it a much closer proxy to the actual serving container. Although `predict()` runs 5 to 10 times slower than `load_model()`, it's worth it. Catching environment issues early can save you hours of troubleshooting failed endpoint deployments.

MLflow also includes handy functions to convert an input example to a serving input and to validate that input:

```
from mlflow.models import validate_serving_input
from mlflow.models import convert_input_example_to_serving_input

serving_payload = convert_input_example_to_serving_input(input_example)
validate_serving_input(model_uri, serving_payload)
```

Now that we have walked through the Model Serving implementation, let's check another common scenario: serving batch features.

Feature Serving

Serving batch predictions is probably one of the most popular and underestimated types of ML model deployment. Predictions are computed in advance with a batch process, stored in a transactional database, and retrieved upon request.

This architecture is very popular in the case of personal recommendations with low latency requirements. For example, an e-commerce store may recommend products to customers on various pages of its website.

Databricks Feature Serving is a great feature for use in this scenario (see Figure 4-2). A scheduled Lakeflow job preprocesses data, retrains the model, and generates predictions, which are stored in a feature table registered in the Unity Catalog. These features are then synced to an online store and served via a Feature Serving endpoint that's defined by a `FeatureSpec`. The application can query this endpoint in real time to retrieve the most up-to-date features for inference.

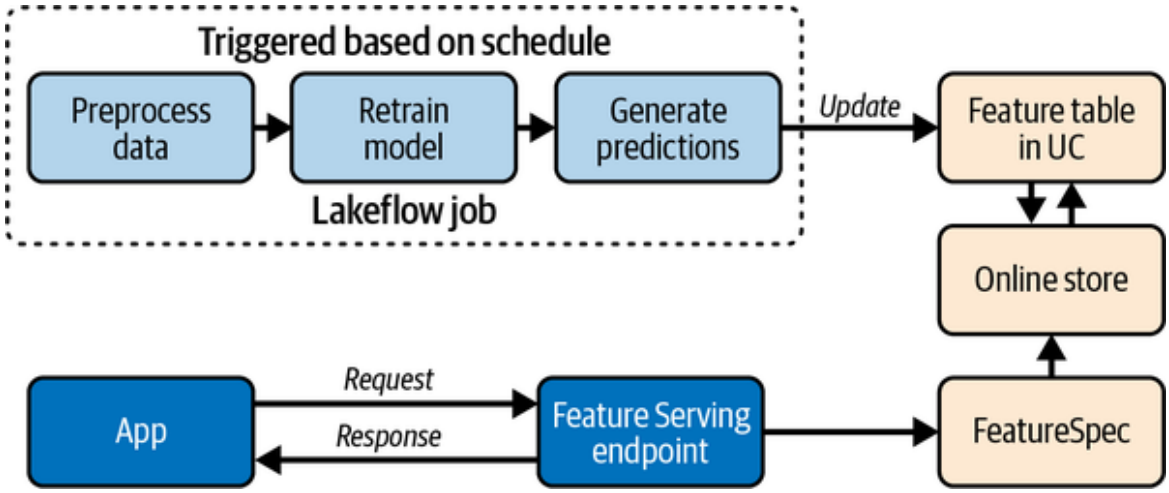


Figure 4-2. Simplified architecture for Feature Serving

When you're using Feature Serving, the payload doesn't need to include every feature the model requires. Values or predictions are precomputed in advance and stored in the online store, and you can retrieve them on demand by using just the `Booking_ID`.

Now, let's go through the *notebooks/chapter4_2.feature_serving.py* notebook, where we cover all the steps required for Feature Serving, starting with the generating predictions step. We'll be using the `hotel_booking_basic` model, which we trained and registered earlier. We can easily access the latest registered version of this model by using the `@latest-model` alias.

> **NOTE**
>
> This notebook must be executed in a Databricks environment as it uses the databricks-feature-engineering package, which does not work from VS Code.

Creating a Feature Table

Let's go through the steps required to create a feature table. First of all, we load the input data into a `pyspark` DataFrame and only select the features required to generate model predictions. Then, we use `mlflow.pyfunc.spark_udf` to generate predictions at scale. The final

output is a new DataFrame that contains just the `Booking_ID` and the predicted booking price, which is labeled `Predicted_BookingPrice`:

```
import mlflow
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, struct

from hotel_booking.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")
spark = SparkSession.builder.getOrCreate()

model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"
model_uri = f"models:/{model_name}@latest-model"
columns = cfg.cat_features + cfg.num_features

input_df = spark.table(f"{cfg.catalog}.{cfg.schema}.hotel_booking")
model_udf = mlflow.pyfunc.spark_udf(spark, model_uri)

preds_df = input_df.select(
   col("Booking_ID"),
   model_udf(struct(*[col(c) for c in
columns])).alias("Predicted_BookingPrice")
)
```

After we define a pyspark DataFrame, we can create a feature table. Here, we use the `databricks-feature-engineering` functionality, but we could also create a feature table by using SQL commands. In fact, we could use any Delta table with a primary key constraint as a feature table. In our example, we'll use the `hotel_booking_price_preds` feature table to update the online store, and we must enable `ChangeDataFeed` for that purpose.

```
from databricks.feature_engineering import FeatureEngineeringClient

fe = FeatureEngineeringClient()

feature_table_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_price_preds"
fe.create_table(
   name=feature_table_name,
```

```
    primary_keys=["Booking_ID"],
    df=preds_df,
    description="Hotel Booking Prices predictions feature table"
)

spark.sql(f"""
          ALTER TABLE {feature_table_name}
          SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
        """)
```

Now that the feature table is available in Unity Catalog, we can create a *FeatureSpec*, which is a user-defined set of features and functions.

Creating a FeatureSpec

To create a `FeatureSpec`, we need a list that consists of instances of `FeatureLookup` and `FeatureFunction`, where the order of elements defines the order of execution. `FeatureSpecs` are stored in and managed by Unity Catalog, and we can view them via the UI in the Databricks Catalog Explorer (just like the tables and models we created earlier).

In our example, we only use `FeatureLookup` to retrieve values based on the key:

```
from databricks.feature_engineering import FeatureLookup

feature_spec_name = f"{cfg.catalog}.{cfg.schema}.return_hotel_booking_prices"
features = [
            FeatureLookup(
                table_name=feature_table_name,
                lookup_key="Booking_ID",
                feature_names=["Predicted_BookingPrice"],
            )
        ]
fe.create_feature_spec(name=feature_spec_name,
                       features=features,
                       exclude_columns=None)
```

Note that I will be covering `FeatureFunction` in the next section of this chapter, when we discuss Model Serving with feature lookups.

Feature Serving would not work efficiently without an Online store, which we can initialize now, now that we've stored the `FeatureSpec` in Unity Catalog.

Publishing a Table to a Databricks Online Feature Store

We can create a Databricks Online Feature Store by using the `create_online_store` method from the `FeatureEngineering` client. We can also define the size of the Feature Store. The capacity options correspond to the number of capacity units (each of which is about 2 GB of RAM): `"CU_1"`, `"CU_2"`, `"CU_4"`, or `"CU_8"`.[2] Notice that we pass the usage policy ID here to ensure that the costs are properly associated with the project:

```
instance_name = "hotel-booking-price-preds"

fe.create_online_store(
   name=instance_name,
   capacity="CU_1",
   usage_policy_id=cfg.usage_policy_id)
```

Since the Databricks Online Feature Store is built on Lakebase, you can also create and manage it by using the Databricks SDK. Make sure to configure the compute used by the branch in your Lakebase project to scale down to zero when idle to avoid unnecessary costs. We'll demonstrate this in Chapter 7, where we use Lakebase to store agent session information.

> **NOTE**
>
> At the time of writing this book, Lakebase is not supported on Databricks Free Edition. You'll therefore need a premium subscription to execute this code.

After we create the online store and verify that its status is `AVAILABLE`, we can synchronize the data from the feature table to the online store by using

the `publish_table()` method:

```
online_store = fe.get_online_store(name="hotel-booking-price-preds")
fe.publish_table(
   online_store=online_store,
   source_table_name=feature_table_name,
   online_table_name=f"{feature_table_name}_online"
)
```

Although the `FeatureSpec` definition doesn't explicitly reference an online table, it's implicitly associated with the corresponding feature table. Figure 4-3 shows the catalog view of the online table, where you can find the reference to the Lakeflow Spark Declarative Pipeline that updates the online table based on the changes in the feature table. Because `ChangeDataFeed` is enabled in the feature table, only incremental updates will be processed.

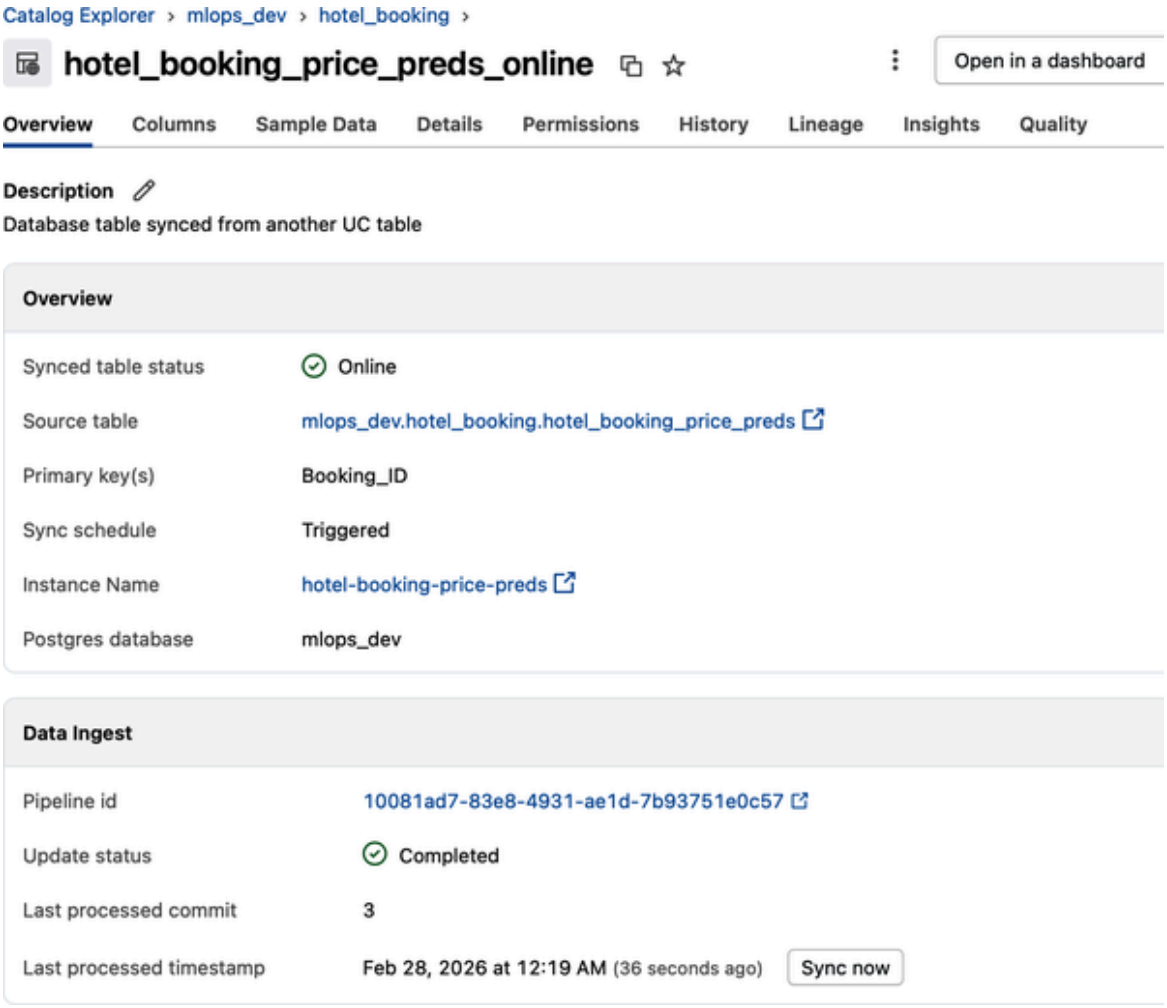


Figure 4-3. Catalog view of an online table

Note that the synchronization schedule is set to Triggered and the pipeline gets triggered when you run `publish_table`. If `publish_table` is called with `publish_mode='CONTINUOUS'`, the sync schedule will be Continuous, which means that the online table gets updated once the feature table gets updated.

Triggered pipelines are more cost efficient in most cases because the cluster runs only for the update window but new data isn't processed until the next trigger. Continuous pipelines keep a cluster running at all times, which costs more but delivers lower latency.

It's generally recommended to use triggered synchronization when updates are needed every 10 minutes or less frequently. Choose continuous mode

when you need fresh data within 10 seconds to a few minutes.[3]

Endpoint Deployment

We can create a Feature Serving endpoint in much the same way we created a model-serving endpoint. The main difference lies in what we provide as the `ServedEntityInput`.

For a model, we specify a particular version of a registered model. However, for a Feature Serving endpoint, we use a `FeatureSpec`. A key difference here is that a `FeatureSpec` doesn't have a version number, unlike a registered model. Here again, we provide a serverless usage policy ID:

```
from databricks.sdk.service.serving import (
   EndpointCoreConfigInput,
   ServedEntityInput,
)

served_entities = [
   ServedEntityInput(
       entity_name=feature_spec_name,
       scale_to_zero_enabled=True,
       workload_size="Small",
   )
]

w = WorkspaceClient()
endpoint_name = "hotel-booking-feature-serving"

w.serving_endpoints.create(
   name=endpoint_name,
   config=EndpointCoreConfigInput(
       served_entities=served_entities),
   budget_policy_id=cfg.usage_policy_id
   )
```

Finally, we can call the endpoint. You'll notice that the payload in this case is much simpler, while the rest stays the same.

```
import requests

host = w.config.host
token = w.tokens.create(lifetime_seconds=1200).token_value

serving_endpoint = f"{host}/serving-endpoints/{endpoint_name}/invocations"

response = requests.post(
   serving_endpoint,
   headers={"Authorization": f"Bearer {token}"},
   json={"dataframe_split": {"columns": ["Booking_ID"],
                             "data": [["INN36285"]]}},
)
```

Now that we have discussed Model Serving and Feature Serving, we can move on to Model Serving with feature lookup, which combines the functionality of the previously discussed architectures.

Model Serving with Feature Lookup

Now, let's explore our most complex scenario, which combines the two approaches we've already covered. In this setup, a model request doesn't need to contain all the features. Instead, some input features are provided with the request itself, while others are automatically retrieved through a *feature lookup* from an online store. The great thing about this is that those features in the online store can be computed in either a batch or streaming fashion, which gives you a lot of flexibility, depending on your needs.

Figure 4-4 shows a simplified architecture for Model Serving with feature lookup. A scheduled Lakeflow job preprocesses data and updates a feature table in Databricks Unity Catalog, which is then synced to the online store. Another scheduled job retrains the model using the latest features and registers it in Unity Catalog. During inference, the Model Serving endpoint fetches features from the online store using a primary key (in our case, `Booking_ID`), combines them with any features provided in the request payload, runs the model's prediction function, and returns the result to the application.

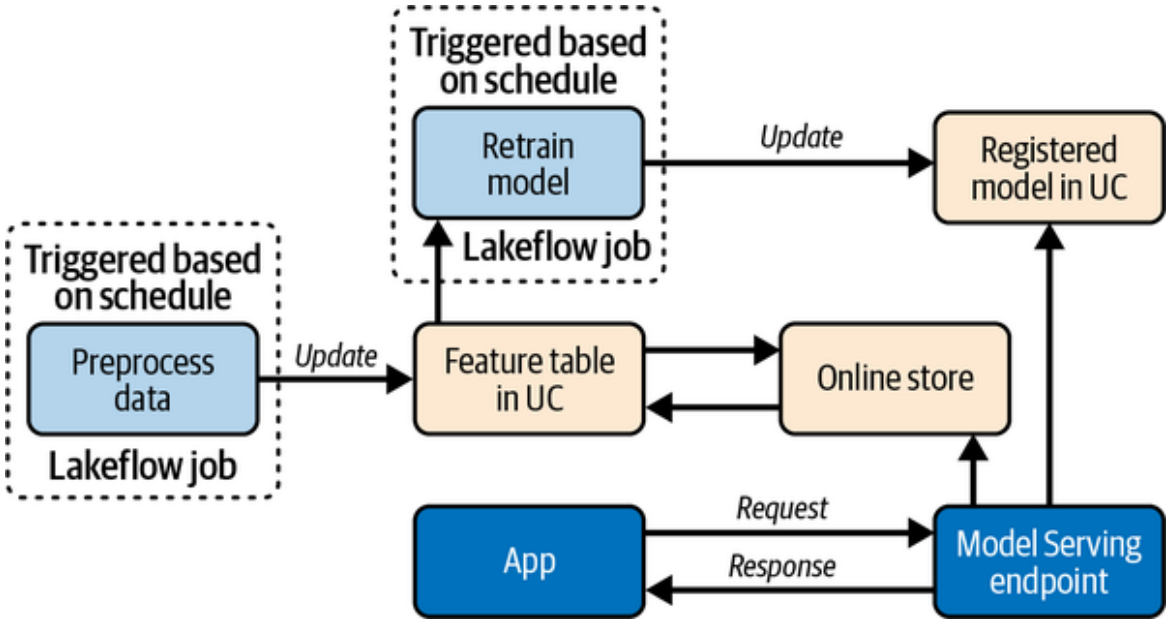


Figure 4-4. Simplified architecture for Model Serving with feature lookup

To implement this architecture, we must log the model using the `databricks-feature-engineering` package. This requires us to define a training set object, where we specify a sequence of feature lookups and feature functions.

I won't cover the model endpoint deployment here, as it is nearly identical to the standard Model Serving scenario. The key difference appears at inference time: you don't need to pass all features in the request. Instead, you must provide a `Booking_ID` as an additional input. This is because some features are either computed dynamically or retrieved from the online feature table, and `Booking_ID` is required to perform those lookups.

Let's walk through the *chapter4_3.model_serving_feature_lookup.py* notebook, where we'll implement all these steps.

We'll start by loading the data. Notice that we drop several features from the training set—specifically, those that will later be computed or looked up: `lead_time`, `arrival_month`, and `repeated`. Later in the notebook, we define feature functions to calculate `lead_time` and `arrival_month` based on the reservation and arrival dates, and we retrieve the `repeated` value from the feature table.

```
from datetime import datetime

from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig
from hotel_booking.data.data_loader import DataLoader
from hotel_booking.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()
cfg = ProjectConfig.from_yaml(config_path="../project_config.yml")

spark = SparkSession.builder.getOrCreate()

data_loader = DataLoader(spark=spark, config=cfg)
train_query, test_query = data_loader.generate_queries()
train_set = spark.sql(train_query).drop(
   "lead_time", "arrival_month", # computed
   "repeated", # looked up
   # present in the original table but not used:
   "P_C", "P_not_C", "booking_status")
test_set = spark.sql(test_query).toPandas()
```

Notice that we also drop features that are present in the DataFrame returned by the `train_query` execution. While this may seem counterintuitive, it prevents those columns from becoming required inputs at serving time. When you create a training set with the model and later log the model using the feature-engineering client, Databricks uses the training set schema (excluding the label) as the model's input contract. As a result, every column present during training is expected in inference requests, even if the model does not actually use it.

Let's start with defining feature functions.

Creating Feature Functions and a Feature Table

A *feature function* is a construct we use to create a training set with the feature-engineering client. It allows us to define how certain features should be computed dynamically as part of the feature pipeline.

A feature function relies on a user-defined function (UDF) that's registered in Databricks Unity Catalog. This ensures that the function will be

governed, reusable, and accessible during both training and inference.

Let's start by defining UDFs for calculating the `lead_time` and `arrival_month` features:

```
lead_time_function = f"{cfg.catalog}.{cfg.schema}.calculate_lead_time"

spark.sql(f"""
CREATE OR REPLACE FUNCTION {lead_time_function}
(arrival_date TIMESTAMP, reservation_date TIMESTAMP)
RETURNS BIGINT
LANGUAGE PYTHON AS
$$
return (arrival_date-reservation_date).days
$$""")

arrival_month_function = f"{cfg.catalog}.{cfg.schema}.get_arrival_month"

spark.sql(f"""
CREATE OR REPLACE FUNCTION {arrival_month_function}
(arrival_date TIMESTAMP)
RETURNS BIGINT
LANGUAGE PYTHON AS
$$
return arrival_date.month
$$""")
```

We can define feature functions by using either SQL or Python. However, for feature-engineering workflows, only Python-based feature functions are supported.

Next, let's create a feature table that contains historical booking features. We'll assume that returning customers may receive a discount on new bookings, which can affect the price per night. Previously, we created a feature table by using the feature engineering client. This time, we'll create the feature table directly by using SQL, with the `Booking_ID` defined as the primary key.

But before assigning `Booking_ID` as the primary key, we must first apply a `NOT NULL` constraint. In Databricks, the `NOT NULL` constraint is *enforced*, meaning that any attempt to insert or update a row with a null value in that

column will fail. In contrast, the primary key constraint is *informational*, which means that Databricks records the metadata and can use it for query optimization but does not validate or enforce uniqueness. As a result, we can still insert duplicate primary key values unless uniqueness is enforced through other mechanisms in our data pipeline.[4]

```
feature_table_name = f"{cfg.catalog}.{cfg.schema}.historical_booking_features"
spark.sql(f"""
   CREATE OR REPLACE TABLE {feature_table_name}
   AS SELECT Booking_ID, repeated FROM
  {cfg.catalog}.{cfg.schema}.hotel_booking
""")
spark.sql(f"""
   ALTER TABLE {feature_table_name}
   ALTER COLUMN Booking_ID SET NOT NULL
""")
spark.sql(f"""
   ALTER TABLE {feature_table_name}
   ADD CONSTRAINT pk_Booking_ID PRIMARY KEY (Booking_ID)
""")
```

We must also enable the `ChangeDataFeed` to synchronize the table with the online store, which is required for the Model Serving with feature lookup:

```
spark.sql(f"""
   ALTER TABLE {feature_table_name}
   SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
""")
```

With our features and UDFs in place, the next step is to define a training set. This is an important step for reproducibility, as it captures the input PySpark DataFrame along with the features, transformations, and labels used. By doing this, we ensure lineage and consistency between our training and inference processes.

Defining the Training Set

We use the `create_training_set()` function to define a training set. We pass a list that consists of instances of `FeatureLookup` and `FeatureFunction`, in which the order of the list determines the order of execution. `FeatureFunction` takes the `udf_name` as an argument, and it also requires the *output name* (what the feature is called in the training set) and the input bindings.

The keys in the `input_bindings` dictionary refer to the names of inputs specified in the UDF, while the values refer to the feature names in the input DataFrame:

```
from databricks.feature_engineering import (
   FeatureEngineeringClient,
   FeatureFunction,
   FeatureLookup,
)

fe = FeatureEngineeringClient()

training_set = fe.create_training_set(
           df=train_set,
           label=cfg.target,
           feature_lookups=[
               FeatureLookup(
                   table_name=feature_table_name,
                   feature_names=["repeated"],
                   lookup_key="Booking_ID",
               ),
               FeatureFunction(
                   udf_name=lead_time_function,
                   output_name="lead_time",
                   input_bindings={"arrival_date": "arrival_date",
                                   "reservation_date": "date_of_reservation"},
               ),
               FeatureFunction(
                   udf_name=arrival_month_function,
                   output_name="arrival_month",
                   input_bindings={"arrival_date": "arrival_date"},
               ),
```

```
        ],
    )
```

Since we are training a LightGBM model, we use the `load_df()` method of the training set to obtain a PySpark DataFrame, and then, we call `toPandas()` to convert it into a Pandas DataFrame. Notice that we include an additional feature, `repeated`, when defining `X_train` and `X_test`. We have not used this feature previously, so we introduce it here to showcase how we can retrieve it via feature lookup from the feature table:

```
training_df = training_set.load_df().toPandas()
X_train = training_df[cfg.num_features + cfg.cat_features + ["repeated"]]
y_train = training_df[cfg.target]
X_test = test_set[cfg.num_features + cfg.cat_features + ["repeated"]]
y_test = test_set[cfg.target]
```

Once we have loaded the data, we can train the model as usual:

```
from hotel_booking.models.lightgbm_model import LightGBMModel

model = LightGBMModel(config=cfg)
model.train(X_train=X_train, y_train=y_train)
```

The only difference is in the model logging step. Previously, we used standard MLflow functions to log the model. Now, we log the model using the `FeatureEngineeringClient` and pass the training set as an additional input.

In this example, we provide dummy values for `branch` and `git_sha`:

```
import mlflow
import numpy as np
from mlflow.models import infer_signature
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

from hotel_booking.config import Tags
```

```
mlflow.set_experiment("/Shared/hotel-booking-training-fe")
tags = Tags(**{"git_sha": "1234567890abcd", "branch": "main"})

run = mlflow.start_run(
   run_name=f"lightgbm-training-{datetime.now().strftime('%Y-%m-%d')}",
   description="LightGBM model training",
   tags=tags.to_dict(),
)
run_id = run.info.run_id
mlflow.log_params(model.parameters)
signature = infer_signature(
   model_input=X_train,
   model_output=model.pipeline.predict(X_train[0:1])
)
y_pred = model.pipeline.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
metrics = {
   "mse": mse,
   "rmse": rmse,
   "mae": mean_absolute_error(y_test, y_pred),
   "r2_score": r2_score(y_test, y_pred),
}
mlflow.log_metrics(metrics)

fe.log_model(
   model=model.pipeline,
   flavor=mlflow.sklearn,
   artifact_path="lightgbm-pipeline-model-fe",
   training_set=training_set,
   signature=signature,
)
```

In a production deployment, however, these should be real values that are automatically passed from the GitHub Actions workflow via the DAB. We will discuss this setup in the next chapter.

It's interesting to examine how the model is logged (see Figure 4-5). The model is stored with the `python_function` flavor, and it uses the `databricks.feature_store.mlflow_model` loader.

In addition to the standard *MLmodel* file and environment definition, the logged artifacts include a *metadata/* folder (which duplicates some of the top-level information) and a *data/* folder. The *data/* folder contains the

serialized raw model as well as a *feature_spec.yaml* file that defines the feature lookups and feature functions required at inference time.

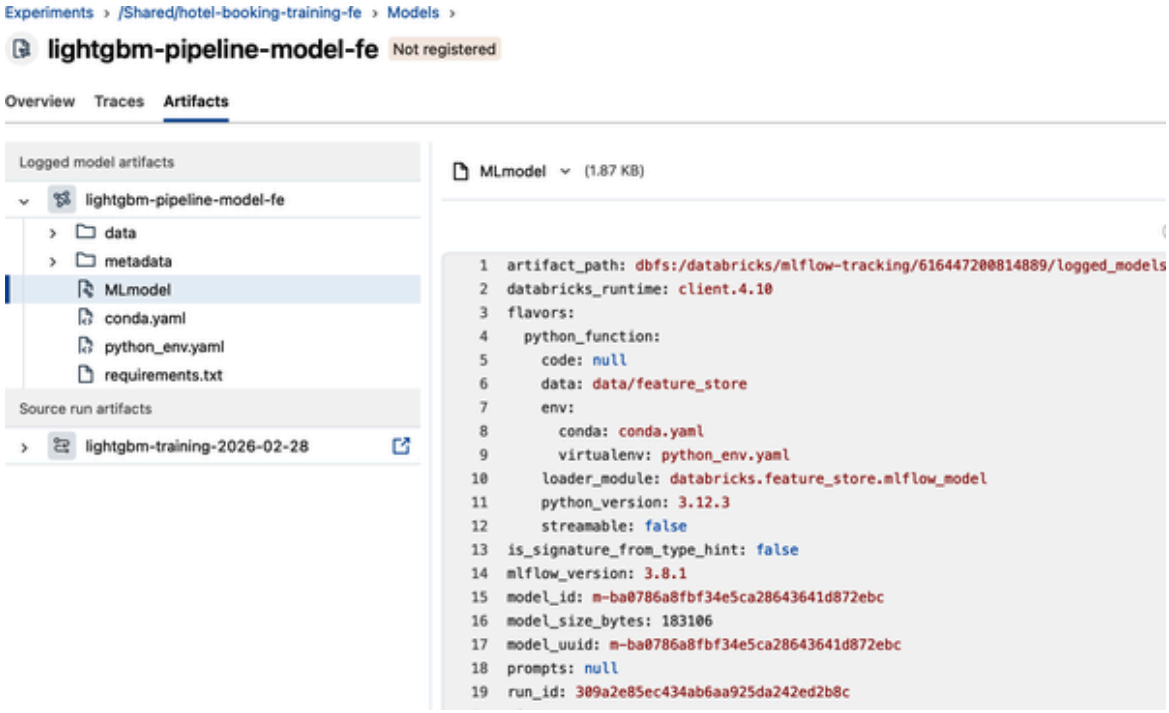


Figure 4-5. UI view of the model logged with feature-engineering package

The *feature_spec.yaml* file captures each input feature's data type, topological ordering, and source. For example, the `number_of_adults` feature exists in the training data, so its source is `training_data`:

```
- number_of_adults:
    data_type: bigint
    topological_ordering: 5
    source: training_data
```

The `repeated` feature is looked up from a feature table, so the source is `feature_store`, with the table and lookup key specified:

```
- repeated:
    table_name: mlops_dev.hotel_booking.historical_booking_features
    feature_name: repeated
    lookup_key:
```

```
    - Booking_ID
    data_type: bigint
    topological_ordering: 14
    source: feature_store
```

The `lead_time` feature is produced by an on-demand feature function, so the source is `on_demand_feature`, with the UDF and its input bindings defined:

```
- lead_time:
    udf_name: mlops_dev.hotel_booking.calculate_lead_time
    input_bindings:
      arrival_date: arrival_date
      reservation_date: date_of_reservation
    data_type: bigint
    topological_ordering: 13
    source: on_demand_feature
```

Registering the Model and Publishing an Online Table

After we log the model, we register it so that we can use the model version behind a Model Serving endpoint:

```
from mlflow import MlflowClient

model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_model_fe"
registered_model = mlflow.register_model(
   model_uri=f"runs:/{run_id}/lightgbm-pipeline-model-fe",
   name=model_name,
   tags=tags.to_dict(),
)

client = MlflowClient()
client.set_registered_model_alias(
   name=model_name,
   alias="latest-model",
   version=registered_model.version,
)
```

Lastly, we must publish an online table to make it reachable for the feature lookup at the serving stage. In this example, we reuse the same online store we created earlier:[5]

```
online_store = fe.get_online_store(name="hotel-booking-price-preds")

fe.publish_table(
   online_store=online_store,
   source_table_name=feature_table_name,
   online_table_name=f"{feature_table_name}_online")
```

When it comes to Model Serving, the deployment process remains the same as what we covered earlier in the Model Serving section. The only difference lies in the input payload, so let's explore it in detail.

Payload Structure

For Model Serving with feature lookup, we do not need to provide values for `lead_time, arrival_month`, or `repeated`, as these features are computed or retrieved automatically at inference time. Instead, we supply the raw inputs required for those transformations: specifically, `arrival_date` and `date_of_reservation` for the feature functions and `Booking_ID` for the feature lookup. Table 4-3 highlights the differences between the input and output payloads across the three serving scenarios we've covered.

Table 4-3. Payloads for Model Serving, Feature Serving, and Model Serving with feature lookup

Scenario	Input payload	Output payload
Model Serving (PyFunc)	`{ "dataframe_split": { 'columns': ['number_of_adults', 'number_of_children', 'number_of_weekend_nights', 'number_of_week_nights', 'car_parking_space', 'special_requests', 'lead_time', 'type_of_meal', 'room_type', 'arrival_month', 'market_segment_type'], 'data': [[2, 0, 0, 1, 0, 0, 103, 'Not Selected', 'Room_Type 1', 8, 'Online']] } }`	`{ "predictions": { "Total price per night": [104.98] } }`
Feature Serving	`{ "dataframe_split": { 'columns': [ 'Booking_ID' ], 'data': [['INN36285']] } }`	`{ "outputs": [ {"Booking_ID": “INN36285”, "Predicted_BookingPrice": 104.98 } ] }`
Model Serving with feature lookup	`{ "dataframe_split": { 'columns': ['Booking_ID', 'number_of_adults', 'number_of_children', 'number_of_weekend_nights', 'number_of_week_nights', 'car_parking_space', 'special_requests', 'type_of_meal', 'room_type', 'market_segment_type', 'arrival_date', 'date_of_reservation’'], 'data': [['INN36285', 2, 0, 0, 1, 0, 0, 'Not Selected', 'Room_Type 1', 'Online', '2018-08-01 00:00:00', '2018-03-01 00:00:00']] } }`	`{ "predictions": { [104.98] } }`

It’s important to note that both the Feature Serving endpoint and the Model Serving endpoint with feature lookup will fail to return predictions if the `Booking_ID` is missing from the feature table. To handle this, you can implement a `FeatureFunction` to substitute default values for missing entries. Alternatively, you can create a custom Python function (PyFunc) model that interacts directly with Lakebase, which is the underlying system used for feature lookups.

So far, we’ve been using a personal access token to call the endpoint. This is great for testing, but it’s not recommended for production environments. Furthermore, route-optimized endpoints can only be queried with an OAuth token. Let’s walk through how to set this up.

OAuth Authentication

Personal access tokens are powerful because they grant the holder all the permissions of the associated user. A more secure approach is to use an OAuth token associated with the service principal, which has no permissions by default. You can then grant it permissions to query a specific endpoint, just as you would a user.

Workspace admins have the required permissions to create a Databricks service principal.[6] Once they create the service principal, they must update the endpoint permissions. They can accomplish both actions via Databricks API, Databricks SDK for Python, or the Databricks Terraform module, or through the workspace settings and the endpoint’s permission settings in the UI. Figure 4-6 depicts the “Service principals” view in the UI.

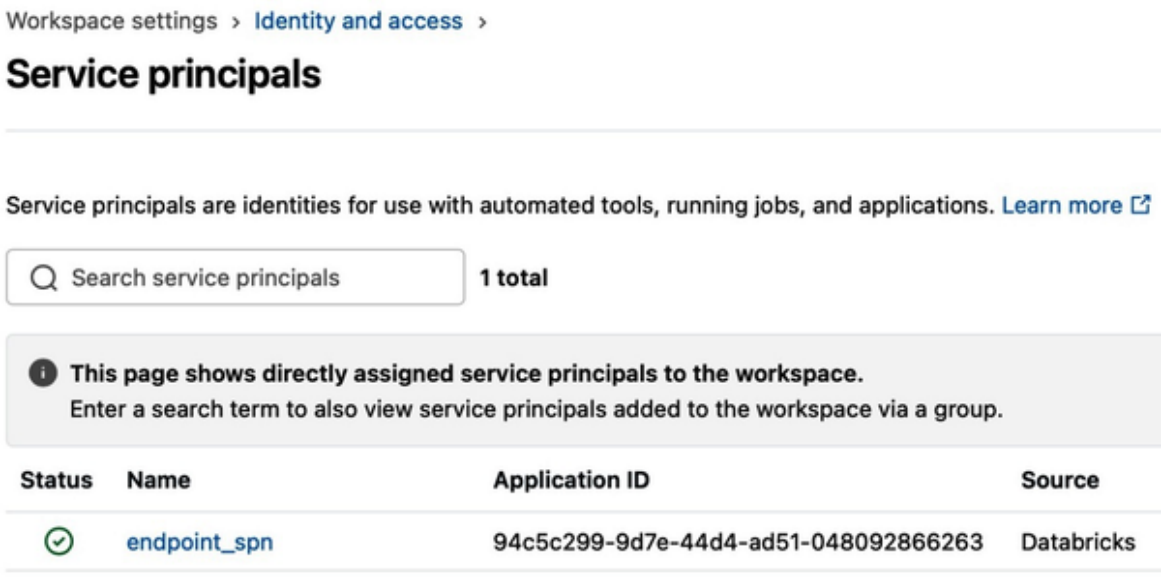


Figure 4-6. "Service principals" view in the UI

Figure 4-7 depicts the Permission Settings view in the UI for an individual endpoint (`hotel-booking-pyfunc`).

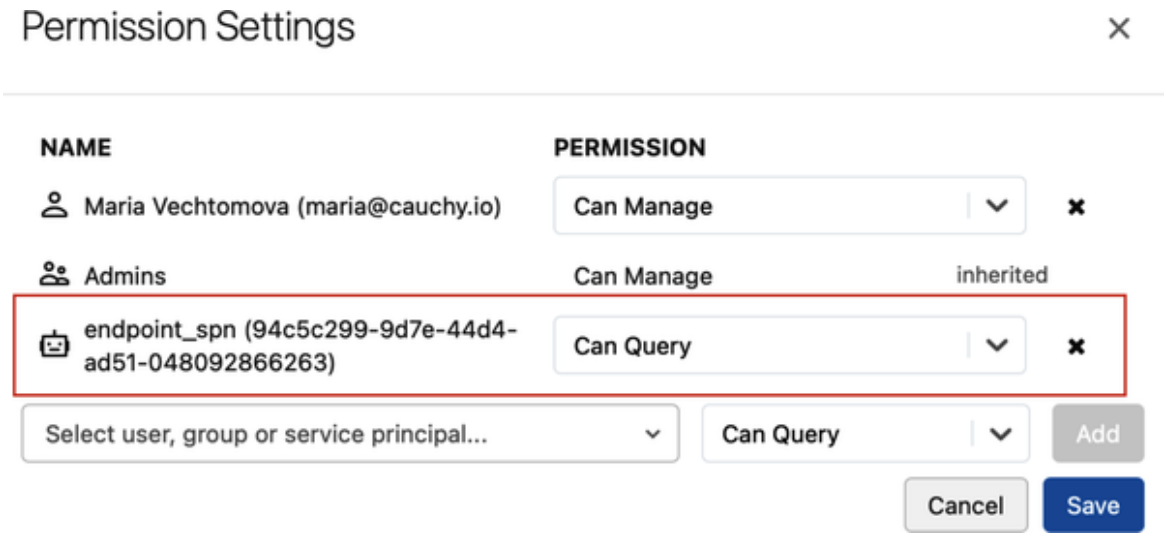


Figure 4-7. Object's "Permission Settings" view in the UI

Once we've created the service principal and set the permissions, we need to create a client secret, which is required for authentication. We can do this programmatically or via the "Service principal details" page in the UI (see Figure 4-8). The secret can have a maximum lifetime of 730 days. Also remember that one of the most common system failures is due to expired

secrets, and there must be a process in place to refresh the secrets periodically before they expire.

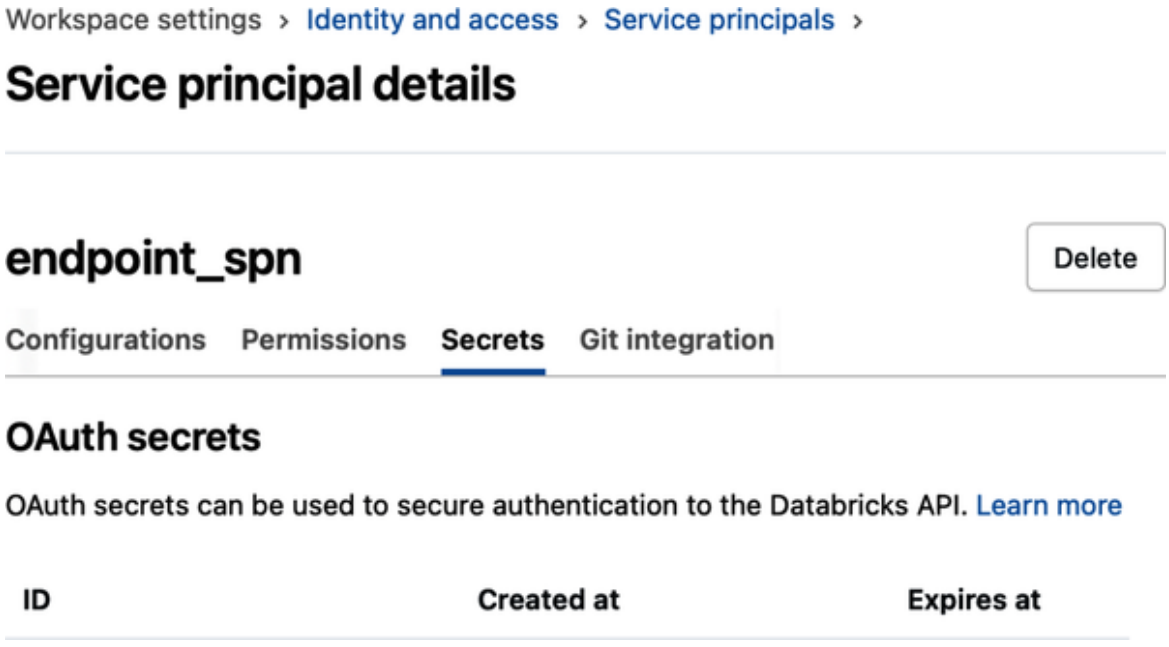


Figure 4-8. "Service principal details: Secrets" view in the UI

Let's store the values of `DATABRICKS_CLIENT_ID` (the same as in `Application ID`) and `DATABRICKS_CLIENT_SECRET` in the *.env* file, so that we can run the following code from the VS Code to get an access token:[7]

```
import os

import requests
from dotenv import load_dotenv
from requests.auth import HTTPBasicAuth

load_dotenv()

def get_token():
   response = requests.post(
       f"https://{os.environ['DATABRICKS_HOST']}/oidc/v1/token",
       auth=HTTPBasicAuth(
           os.environ["DATABRICKS_CLIENT_ID"],
           os.environ["DATABRICKS_CLIENT_SECRET"]
       ),
       data={
           'grant_type': 'client_credentials',
```

```
            'scope': 'all-apis'
        }
    )
    return response.json()["access_token"]

token = get_token()
```

The token we fetched will be valid for one hour. For more details on OAuth authentication, you can refer to the Databricks OAuth documentation.

Now, before we move on, let's discuss some important practicalities. You need to be aware of Databricks' serving limitations when designing your applications, so let's cover those next.

Serving Limitations

Databricks Model Serving offers a scalable way to deploy ML models, but it comes with constraints that are worth understanding up front, particularly when migrating from existing infrastructure.

A few are likely to surface during a migration:

Request format

> Endpoints expect a predefined request payload schema. As we saw earlier, this can require adaptation if your application already uses a custom or incompatible format.

Networking

> Private connectivity is not always available out of the box and may need additional configuration, depending on your cloud and workspace setup.

CPU memory

> By default, each CPU model endpoint is allocated 4 GB of memory per worker. The `CPU_MEDIUM` and `CPU_LARGE` workload types let you trade concurrency for more memory on the

same CPU hardware, which matters when a model does not fit within the standard allocation—a common consideration in multimodel deployments.

Payload and logging size

Requests are limited to 16 MB each. Separately, requests or responses larger than 1 MB are not captured in inference tables, so very large payloads can serve successfully but fall outside your logged data.

These are mostly design time decisions rather than blockers, but they are easier to account for early than to retrofit later. The Databricks documentation lists the full set of current limits.

None of this should suggest that the platform is constrained at the top end. Concurrency is managed at the workspace level, with a default provisioned concurrency of 4,096 units per workspace and 1,024 per model, and a single endpoint can reach up to 300,000 queries per second (QPS) with route optimization. Most of these limits can be raised on request.

GPU serving, though outside the scope of this book, has matured along the same lines, and it now backs some genuinely demanding production workloads. Superhuman is a good example.[8] It migrated its grammar correction model from a self-managed vLLM stack to Databricks Model Serving, and now, it serves it at more than 200,000 QPS, with subsecond P99 latency and four nines of availability.

Conclusion

In this chapter, we explored the various Model Serving architectures that are available on Databricks, starting with basic approaches such as Model Serving and Feature Serving and progressing to more advanced patterns like Model Serving with feature lookup. We also discussed key topics including payload structure, authentication, and serving limitations.

Let’s now summarize when each serving approach is appropriate:

Feature Serving

> This is useful when predictions are precomputed and simply need to be retrieved at request time, or when a lightweight transformation is required that doesn’t involve a model.

Model Serving

> This is the right choice when predictions need to be computed in real time and all required features are provided directly in the request payload.

Model Serving with feature lookup

> This is appropriate when predictions are computed on demand but some features must be fetched from an online feature store during inference.

The different options fit different operational requirements depending on latency, infrastructure, and the availability of features at the time of prediction.

Now, with Model Serving in place, our next step is to bring everything together into a robust, automated deployment workflow.

---

1 For a detailed discussion, see Alex Deng, *Causal Inference and Its Applications in Online Industry* (self-pub online, 2023), ch. 8, “Statistical Analysis of A/B Tests,” *https://oreil.ly/XlMkP*.

2 The Lakebase Autoscaling model is the default for the Databricks Online Feature Store. For information about autoscaling, see the Databricks documentation.

3 This is a generic recommendation for Lakeflow Spark Declarative Pipelines, per the Databricks documentation.

4 Consult the Databricks documentation on constraints for more details.

[5] For cost efficiency reasons during development, you should make sure to enable scale to 0 on Lakebase compute. See he Databricks documentation for details.

[6] For Databricks on Azure, it is common to use Microsoft Entra ID service principals instead and sync them automatically to the Databricks account using automatic identity management. Consult the Databricks documentation for more details.

[7] Make sure to add the *.env* file to *.gitignore* to avoid accidentally committing secrets to your source code repository. While *.env* files are convenient for local development, production environments should use a dedicated secret management solution, such as a Key Vault, to securely store and manage credentials, API keys, and other sensitive information.

[8] Read the blog "How Superhuman and Databricks built a 200K QPS inference platform together".

Chapter 5. Machine Learning Model Deployment

So far, we've explored the individual stages of building and deploying a ML model: data preprocessing, feature engineering, model training, evaluation, and serving. In this chapter, we'll shift our focus to orchestration and look at how to connect these steps into an automated pipeline that runs reliably in production.

We'll walk through how to structure end-to-end ML workflows on Databricks, including platform components, workspace design, permission management, and deployment strategies. By the end of the chapter, you'll understand how to design your ML pipeline using best practices that are suited to the Databricks environment.

Let's begin with the orchestration of our machine learning pipelines. In Chapter 2, we introduced DABs and examined the structure of the *databricks.yml* file. What we have not yet looked at are the resources that get deployed when the `databricks bundle deploy` command runs.

The *resources* folder is referenced in the bundle definition and includes the following files:

ml_pipeline.yml

: This is a Lakeflow job that defines the whole ML pipeline.

ml_monitoring.yml

: This is a Lakeflow job that updates the monitoring table.

alert.yml

: This file contains alert-related resources.

In this chapter, we focus on the ML pipeline, and in Chapter 6, we'll turn to monitoring and alerting resources.

Machine Learning Pipeline

Before we examine the structure of the *ml_pipeline.yml* file, let's clarify the objective. The file contains a Lakeflow job that brings together data preprocessing, model retraining and evaluation, and model deployment. Figure 5-1 illustrates the pipeline's architecture, highlighting how it supports retraining and deploying models with built-in conditional logic.

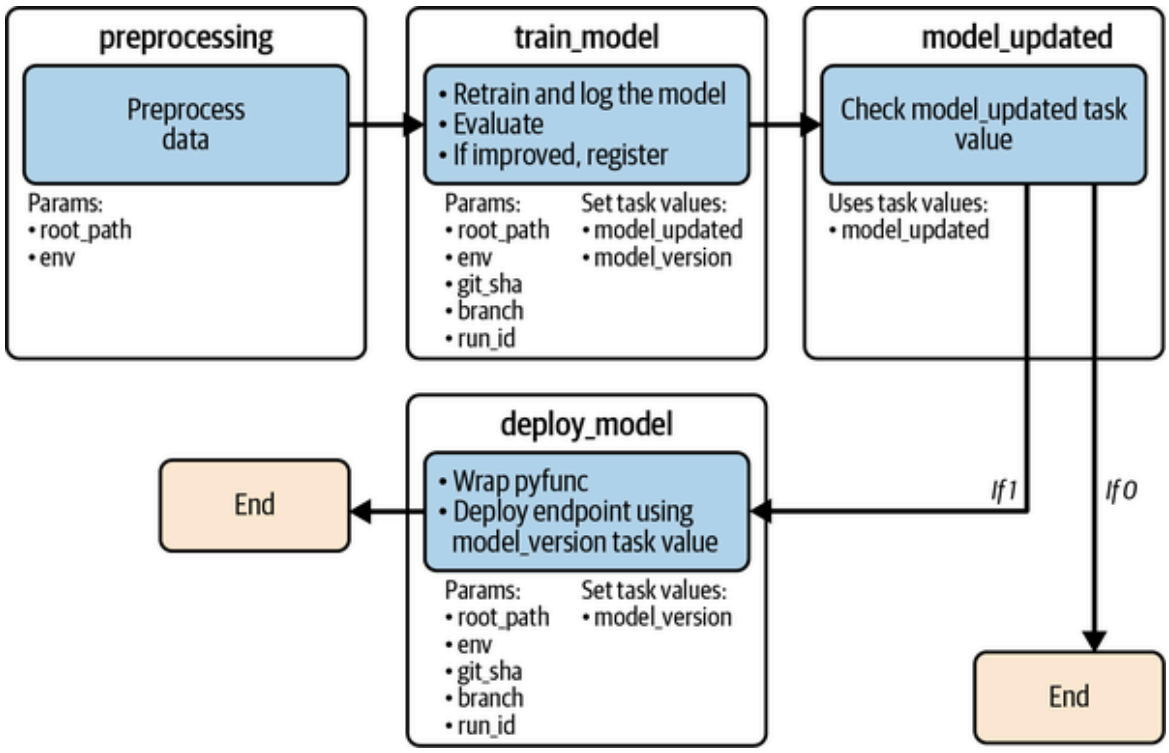


Figure 5-1. Lakeflow job structure

The workflow begins with a preprocessing step that prepares the data. Next, the `train_model` task retrains the model, evaluates its performance against the latest registered model on the new test set, and registers it if there is an improvement. It sets two task values: `model_updated` and `model_version`.

The `model_updated` task is an `IF` task in the Lakeflow job. It checks whether a new model was actually registered. If not (i.e., if the task

`returns model_updated = 0`), the job ends early. If a new model was registered (i.e., if the task returns `model_updated = 1`), the workflow continues to the `deploy_model` task, which wraps the model as a PyFunc and deploys it to an endpoint using the latest `model_version`. This setup ensures that only evaluated and improved models are deployed to production.

Each task in the workflow (except the `IF` task) runs a Python script that requires specific parameters. Some parameters are set by the bundle:

env

: This parameter matches the bundle target (e.g., `dev`, `prd`).

root_path

: This is the root directory of the bundle.

git_sha and branch

: These are variables that are defined in the bundle and overwritten by the CI/CD pipeline.

run_id, job_id, and task_name

: These are set dynamically, using references like `{{job.run_id}}` and `{{task.name}}`, when the job starts.

We need these parameters for a few reasons:

- They allow the scripts to access the correct configuration files.
- We can use them to tag experiments, models, and deployments, so we can always trace a result back to the corresponding code and run.

To simplify parameter parsing, we'll define a `create_parser()` function in the `utils.common` module of our package.

First, we define the common arguments among all steps: `env` and `root_path`. We then create a `common` parser that accepts these two arguments, which is used for preprocessing and later for monitoring.

Next, the training and deployment steps require the same arguments, so we create a `train_deploy_model` parser for them:

```
def create_parser(args: Sequence[str] = None) -> argparse.Namespace:
    """Create and configure an argument parser for MLOps on Databricks.

    This function sets up a parser with subparsers for different MLOps
operations.

    :param args: Optional sequence of command-line argument strings
    :return: Parsed argument namespace
    """
    parser = argparse.ArgumentParser(description="Parser for MLOps on
Databricks")
    subparsers = parser.add_subparsers(dest="command", required=True)

    # Common arguments
    common_args = argparse.ArgumentParser(add_help=False)
    common_args.add_argument(
        "--root_path", type=str, required=True,
        help="Path of root on DAB")
    common_args.add_argument(
        "--env", type=str, required=True,
        help="Path of env file on DAB")

    # Data ingestion subparser
    subparsers.add_parser("common", parents=[common_args], help="Common
processor")

    # Model training & registering subparser
    model_parser = subparsers.add_parser(
        "train_deploy_model",
        parents=[common_args],
        help="Model training and registering")
    model_parser.add_argument(
        "--branch", type=str, required=True,
        help="branch of the project")
    model_parser.add_argument(
        "--git_sha", type=str, required=True,
        help="git sha of the commit")
    model_parser.add_argument(
```

```
        "--run_id", type=str, required=True,
        help="run id of the run of the Lakeflow job")
    model_parser.add_argument(
        "--job_id", type=str, required=True,
        help="Lakeflow job id")
    return parser.parse_args(args)
```

Using this approach means we don't have to define the argument parsing logic in each script, which simplifies the project code. Let's now take a deep dive into each of the steps.

Data Preprocessing

Let's take a look at the *ml_pipeline.yml* file. It has a resources section where one job, `ml-pipeline`, is defined. Once deployed, the job will be called `hotel-booking-ml-pipeline`. It runs based on a schedule (which we can set using Quartz Cron syntax): every Monday at 6:00 a.m. CET. We can set the pause status to `PAUSED` to nonproduction environments by using the `schedule_pause_status` bundle variable.

The schedule is illustrative: in practice, to choose the right cadence, you must think carefully about your specific use case. Retraining frequency depends on several factors. You should ask yourself how quickly your data distribution shifts. A recommendation system for a news platform might need daily updates as trending topics change rapidly, while an email classification model for internal processes might remain stable for months. Consider also the cost of retraining versus the cost of degraded performance: if a stale model means lost revenue or poor user experience, more frequent retraining pays for itself.

Triggering retraining only when monitoring detects performance degradation sounds appealing in theory. But in practice, this approach is often impractical. By the time your metrics have degraded enough to trigger retraining, you're already serving poor predictions to users. Moreover, the degradation itself may signal a deeper problem that retraining won't fix. If your input data quality has dropped, an upstream system changed its output format, or a key feature has stopped updating, throwing fresh training at the

problem won't help. You'll just train a new model on the same broken data. But scheduled retraining with robust monitoring catches these issues earlier and keeps the system in a known, predictable state. We'll talk about monitoring in detail in Chapter 6.

Here is how the schedule for the pipeline is defined in the corresponding *resources/ml_pipeline.yml* file:

```
resources:
 jobs:
   ml-pipeline:
     name: hotel-booking-ml-pipeline
     schedule:
       quartz_cron_expression: "0 0 6 ? * MON"
       timezone_id: "Europe/Amsterdam"
       pause_status: "${var.schedule_pause_status}"
```

We pass a usage policy ID (formerly known as a budget policy ID) for cost monitoring purposes:

```
budget_policy_id: "${var.usage_policy_id}"
```

We'll discuss cost allocation in Chapter 9.

Here, we will use a serverless cluster to run the job. We'll define an environment called `default` and select environment version 4, which matches Python 3.12 (which we are using), as we discussed in Chapter 2. Here, we'll also specify the project dependencies: all *.whl* files in the *dist* folder, where the project package built by the bundle is stored. This environment is then referenced in all the tasks:

```
environments:
  - environment_key: default
    spec:
      environment_version: "4"
      dependencies:
        - ../dist/*.whl
```

The first task of the job is called `preprocessing`. It is a `spark_python` task that runs the `preprocess_data.py` Python script. Notice that the location of the script is relative to the location of the *ml_pipeline.yml* file. (It is located in the *resources* folder.):

```
tasks:
  - task_key: preprocessing
    environment_key: default
    spark_python_task:
      python_file: ../scripts/preprocess_data.py
      parameters:
        - "common"
        - "--root_path"
        - "${workspace.root_path}"
        - "--env"
        - "${bundle.target}"
```

You'll notice that the *preprocess_data.py* script takes two arguments: `env` and `root_path`. The first parameter, `common`, means that we will use the common parser to parse the arguments. The `databricks bundle deploy` command uploads all the files that belong to the bundle to a specific location (defined as the `root_path`). Our data preprocessing code depends on the *project_config.yml* file, which is stored in the files folder under the root path. Because the root path changes depending on the target, it is convenient to provide it as an argument. In the *project_config.yml* file, we define the environment-specific values for the catalog and schema. These values match the targets specified in the *databricks.yml* file, so we can pass the target as an argument.

Inside the `preprocess_data.py` script, we use the `create_parser` function we defined earlier to parse the arguments that were passed to the script. We need these arguments to identify the location of the configuration file and load it:

```
import pandas as pd
import yaml
from loguru import logger
from pyspark.sql import SparkSession
```

```
from hotel_booking.config import ProjectConfig
from hotel_booking.data.data_processor import DataProcessor
from hotel_booking.utils.common import create_parser

args = create_parser()

cfg = ProjectConfig.from_yaml(
   config_path=f"{args.root_path}/files/project_config.yml", env=args.env
)
logger.info("Configuration loaded:")
logger.info(yaml.dump(cfg, default_flow_style=False))
```

After that, the code gets really simple. We initialize the `DataProcessor` class we defined earlier by using the Pandas DataFrame, project configuration, and `SparkSession`. We preprocess the data, generate 1,000 synthetic observations, and store them in Unity Catalog under the catalog and schema defined in the *project_config.yml* file:

```
spark = SparkSession.builder.getOrCreate()

# Load the dataset
df = pd.read_csv(f"{args.root_path}/files/data/booking.csv")

# Initialize DataProcessor
data_processor = DataProcessor(df=df, config=cfg, spark=spark)
data_processor.preprocess()
if spark.catalog.tableExists(data_processor.target_table):
   data_processor.generate_synthetic_df(n=1000, max_date=None)

data_processor.save_to_catalog()
```

And with that, we're done with the data preprocessing step! Now, let's move to the next part: model training.

Training, Evaluating, and Registering a Model

In the *databricks.yml* file, we've defined the `branch` and `git_sha` variables, which are passed as parameters in the `train_model` task, along with other parameters. Here, we'll use the `train_deploy_model` parser we defined earlier:

```
- task_key: "train_model"
  environment_key: default
  depends_on:
    - task_key: "preprocessing"
  spark_python_task:
    python_file: "../scripts/train_register_model.py"
    parameters:
      - "train_deploy_model"
      - "--root_path"
      - "${workspace.root_path}"
      - "--env"
      - "${bundle.target}"
      - "--branch"
      - "${var.branch}"
      - "--git_sha"
      - "${var.git_sha}"
      - "--run_id"
      - "{{job.run_id}}"
      - "--job_id"
      - "{{job.id}}"
```

This task runs the `train_register_model.py` Python script. Here, we are mostly going through the same steps as in the "MLflow Evaluation" section of Chapter 3. Just as in the data preprocessing step, we load the project configuration, initialize the `SparkSession`, and parse the arguments, and then, we use the `DataLoader` class to load and split the data into the train and the test set:

```
import mlflow
import yaml
from loguru import logger
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig, Tags
from hotel_booking.data.data_loader import DataLoader
from hotel_booking.models.lightgbm_model import LightGBMModel
from hotel_booking.utils.common import create_parser

args = create_parser()

cfg = ProjectConfig.from_yaml(
   config_path=f"{args.root_path}/files/project_config.yml", env=args.env)
```

```
logger.info("Configuration loaded:")
logger.info(yaml.dump(cfg, default_flow_style=False))

spark = SparkSession.builder.getOrCreate()

data_loader = DataLoader(spark=spark, config=cfg)
X_train, y_train, X_test, y_test = data_loader.split()
```

After we load the data, we train and log the model. The model metrics dictionary is stored as `metrics_new`. Notice that we pass actual values of the `run_id`, `git_sha`, and `branch` as tags:

```
model = LightGBMModel(config=cfg)
tags = Tags(**{"git_sha": args.git_sha, "branch": args.branch, "run_id":
args.run_id})
model.train(X_train=X_train, y_train=y_train)
model_info = model.log_model(
   experiment_name="/Shared/hotel-booking-training",
   tags=tags,
   X_test=X_test,
   y_test=y_test,
   train_set_spark=data_loader.train_set_spark,
   train_query=data_loader.train_query,
   test_set_spark=data_loader.test_set_spark,
   test_query=data_loader.test_query,
)

metrics_new = model.metrics
```

Now, we'll evaluate the latest registered model (if it exists) on the new test set. If no older registered model with the `latest-model` alias exists, or if the newly trained model improves on the metric (here, we use root mean squared error as an example), we register it and set the `model_updated` and `model_version` task values, which we will use in the follow-up tasks:

```
sklearn_model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"
eval_data = X_test.copy()
eval_data[cfg.target] = y_test

should_register = True
try:
   mlflow.MlflowClient().get_model_version_by_alias(sklearn_model_name,
```

```
"latest-model")
    model_exists = True
except mlflow.exceptions.RestException:
    model_exists = False
    logger.info(f"No {sklearn_model_name}@latest-model yet. Registering new
model.")

if model_exists:
    result = mlflow.models.evaluate(
        model=f"models:/{sklearn_model_name}@latest-model",
        data=eval_data,
        targets=cfg.target,
        model_type="regressor",
        evaluators=["default"],
    )
    rmse_old = result.metrics["root_mean_squared_error"]
    rmse_new = metrics_new["root_mean_squared_error"]
    should_register = rmse_new < rmse_old
    logger.info(
        f"Current RMSE {rmse_old:.4f}, new RMSE {rmse_new:.4f} -> "
        + ("registering new model" if should_register else "keeping current
model")
    )

if should_register:
    model_version = model.register_model(model_name=sklearn_model_name,
tags=tags)
    dbutils.jobs.taskValues.set(key="model_version", value=model_version)
    dbutils.jobs.taskValues.set(key="model_updated", value=1)
else:
    dbutils.jobs.taskValues.set(key="model_updated", value=0)
```

Deploying a Model Serving Endpoint

The `model_updated` task defines whether the pipeline will run further. It checks whether the `model_updated` task value is equal to 1: if the condition evaluates to true, the job proceeds to the final step, `deploy_model` task. It takes the same arguments as the `train_model` task:

```
- task_key: model_updated
  condition_task:
    op: "EQUAL_TO"
    left: "{{tasks.train_model.values.model_updated}}"
```

```
      right: "1"
    depends_on:
      - task_key: "train_model"

  - task_key: "deploy_model"
    environment_key: default
    depends_on:
      - task_key: "model_updated"
        outcome: "true"
    spark_python_task:
      python_file: "../scripts/deploy_model.py"
      parameters:
        - "train_deploy_model"
        - "--root_path"
        - "${workspace.root_path}"
        - "--env"
        - "${bundle.target}"
        - "--branch"
        - "${var.branch}"
        - "--git_sha"
        - "${var.git_sha}"
        - "--run_id"
        - "{{job.run_id}}"
        - "--job_id"
        - "{{job.id}}"
```

The task runs the `deploy_model.py` script. After loading the project configuration and parsing the arguments, we retrieve the `model_version` task value from the `train_model` task. We then use it to look up the corresponding `ModelInfo` instance:

```
import mlflow
from loguru import logger

from hotel_booking.config import ProjectConfig, Tags
from hotel_booking.models.pyfunc_model_wrapper import HotelBookingModelWrapper
from hotel_booking.models.serving import serve_model
from hotel_booking.utils.common import create_parser

args = create_parser()

model_version = dbutils.jobs.taskValues.get(
   taskKey="train_model",
   key="model_version")
logger.info(f"Retrieved wrapped model version: {model_version}")
```

```
cfg = ProjectConfig.from_yaml(
   config_path=f"{args.root_path}/files/project_config.yml", env=args.env)

model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_pyfunc"
wrapped_model_name = f"{cfg.catalog}.{cfg.schema}.hotel_booking_basic"

wrapped_model_info = mlflow.models.get_model_info(
    model_uri=f"models:/{wrapped_model_name}/{model_version}"
)
```

We then use that to log and register the wrapper with its dependencies:

```
wrapper = HotelBookingModelWrapper()

with open(f"{args.root_path}/files/version.txt") as f:
   version = f.read().strip()
whl_name = f"hotel_booking-{version}-py3-none-any.whl"
code_paths = [f"{args.root_path}/artifacts/.internal/{whl_name}"]

tags = Tags(**{"git_sha": args.git_sha,
               "branch": args.branch,
               "run_id": args.run_id})

model_version = wrapper.log_register_model(
   wrapped_model_info=wrapped_model_info,
   pyfunc_model_name=model_name,
   experiment_name="/Shared/hotel-booking-pyfunc",
   code_paths=code_paths,
   tags=tags,
)
```

After we have registered the model, we can finally deploy it using the `serve_model` function that we defined earlier (and that you can find in the `models.serving` module) and finish our pipeline:

```
serve_model(
   entity_name=model_name,
   entity_version=model_version,
   budget_policy_id=cfg.usage_policy_id,
   endpoint_name="hotel-booking-pyfunc",
   schema_name=cfg.schema,
   catalog_name=cfg.catalog,
```

```
    table_name_prefix="hotel_booking_monitoring",
)
```

As mentioned earlier, we should view PyFunc more as a FastAPI model wrapper and should keep it separate from the model training logic. That way, we won't necessarily have to retrain it if the payload logic needs to change.

So far, we've been using *serverless compute* to run our jobs. Now, let's look at another Lakeflow job that explicitly defines a *classic cluster configuration* instead. In this example, we use *classic compute* purely for demonstration purposes, to show how cluster specifications work within a job definition. However, if you are working in the Databricks Free Edition, you should update the configuration to use serverless compute, as it is the only option available in that environment.

Serverless versus Classic Compute for Lakeflow Jobs

In the *resources* folder, you can find another *yml* file called *ml_monitoring.yml*, which defines the `hotel-booking-monitor-update` job.

This job is configured to run daily at 9:00 CET, and its schedule can be paused depending on the bundle deployment target (e.g., development, production). The job cluster, where we can set many different configurations, is defined, and we can use this cluster to run the `scripts/refresh_monitor.py` Python script:

```
resources:
 jobs:
   house-price-monitor-update:
     name: hotel-booking-monitor-update-workflow
     schedule:
       quartz_cron_expression: "0 0 9 * * ?"
       timezone_id: "Europe/Amsterdam"
       pause_status: "${var.schedule_pause_status}"
     tags:
       project_name: "hotel-booking"
```

```
    job_clusters:
      - job_cluster_key: "hotel-booking-cluster"
        new_cluster:
          spark_version: "17.3.x-scala2.13"
          data_security_mode: "SINGLE_USER"
          node_type_id: "r3.xlarge"
          driver_node_type_id: "r3.xlarge"
          num_workers: 1

    tasks:
      - task_key: "refresh_monitor_table"
        job_cluster_key: "hotel-booking-cluster"
        spark_python_task:
          python_file: "../scripts/refresh_monitor.py"
          parameters:
            - "common"
            - "--root_path"
            - "${workspace.root_path}"
            - "--env"
            - "${bundle.target}"
        libraries:
         - whl: ../dist/*.whl
```

Notice that we do not provide a usage policy ID in the job definition, as it can only be applied to serverless resources. Instead, we pass a tag for cost allocation.

We can also override the job's default cluster with an interactive cluster by specifying the `cluster_id` argument for a given target. For example, it's common practice to do this in the `dev` target to speed up iteration and debugging. In this setup, jobs run directly on an existing interactive cluster, rather than spinning up a new one. Additionally, we set the `dynamic_version` setting for `artifacts` to `true`, to ensure that the latest version of the package is always installed whenever the code changes, which is helpful during active development:

```
targets:
 dev:
  default: true
  mode: development
  cluster_id: 0513-112814-kzixtb9s
  workspace:
    host: <your_host>
```

```
    root_path: "/Workspace/Users/${workspace.current_user.userName}/\
               .bundle/${bundle.target}/${bundle.name}"
artifacts:
  default:
    type: whl
    build: uv build
    path: .
    dynamic_version: true
```

In Chapter 6, we'll cover the monitoring job in detail, including its role in tracking model performance. For now, our focus is on how serverless jobs differ from jobs that utilize classic jobs compute, which can be useful if more customization is required.

So far, we've worked entirely in a development environment, without worrying about how code and models move across environments. But to address promotion strategies, we'll first need to understand the basics of Databricks platform design, including workspace structure, environment isolation, and compute options. That foundation is key for managing ML workflows in production.

Databricks Platform Design

We have already interacted with various components of the Databricks platform and experienced the Databricks workspace—a user interface that allows developers to create, access, and manage various Databricks assets, such as experiments, models, and jobs. It is common for organizations to have more than one workspace and use separate workspaces to isolate Databricks assets from other environments, teams, and even data and AI products.

We've also interacted with Unity Catalog: we used it to create Delta tables and register models. For a workspace to use Unity Catalog, it must be attached to a Unity Catalog metastore, which is the top-level container for metadata of all data and AI assets. You can have one metastore per cloud region, and you can only attach a workspace to one metastore in the same

cloud region. It is not uncommon for large organizations with international operations to have multiple Unity Catalog metastores.

In Unity Catalog, assets are organized using the following a three-tier hierarchy within the metastore:

1. Catalogs, which are the highest level of organization.
2. Schemas, which are contained within catalogs.
3. Assets, which include the tables, views, volumes, models, and functions within schemas. We refer to these assets with this three-part naming convention: *catalog.schema.asset*.

Figure 5-2 depicts this hierarchy.

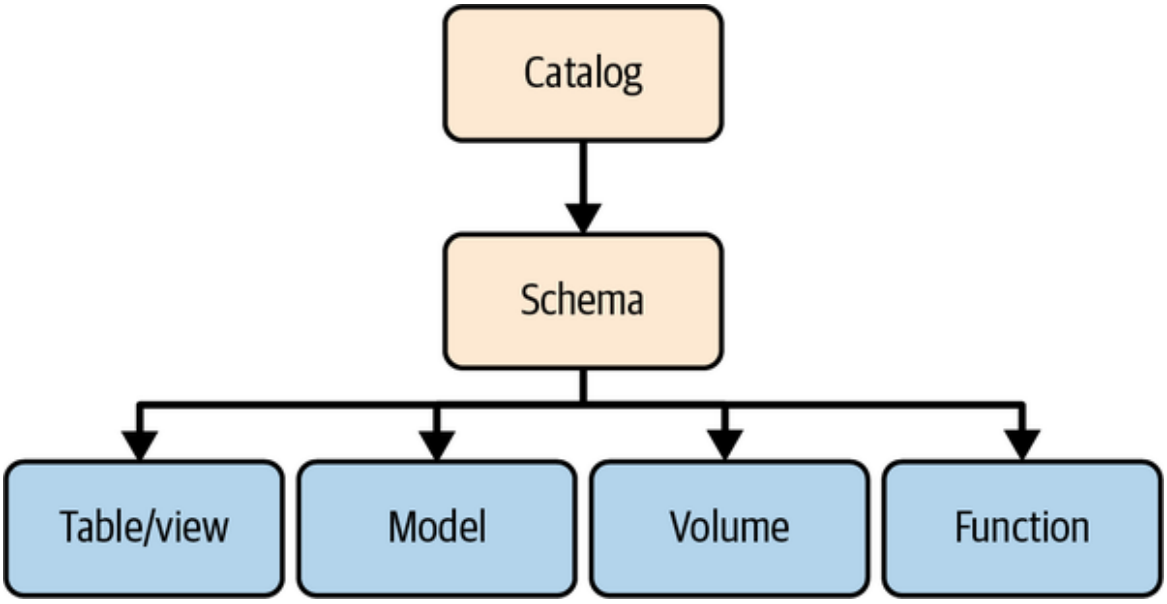


Figure 5-2. Unity Catalog metastore object hierarchy

All the workspaces belong to a Databricks account. Databricks account admin permissions are needed to create and manage workspaces and Unity Catalog metastores, and they're also needed to configure settings on the account level (such as by enabling certain features). Figure 5-3 depicts the relationships among accounts, metastores and workspaces.

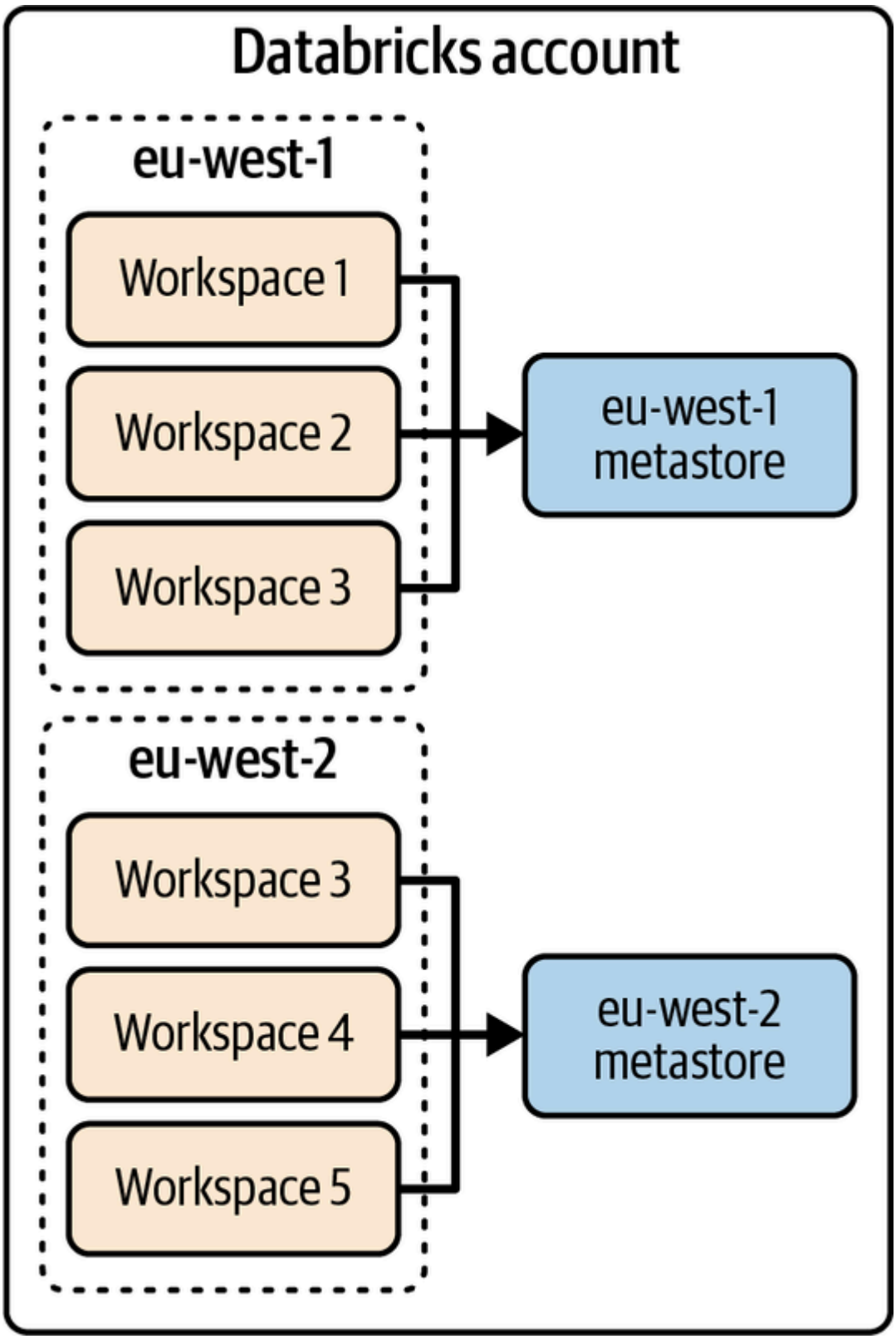


Figure 5-3. Relationships among accounts, workspaces and metastores

I've mentioned that you may want to have different Databricks workspaces for different environments, but how many environments do you actually need? Let's look at that next.

How Many Environments Do You Need?

The development, testing, acceptance, and production (DTAP) approach is probably the most common one in the industry.

One of the earliest formalizations of DTAP came from Cal Evans in 2009.[1] He noted that "The structure of development environments hasn't radically changed over the past 20 years." That means the DTAP pattern is now over 35 years old, so it predates cloud computing, IaC, and the widespread adoption of version control.

Consider what the software development sequence looked like before DTAP emerged:

1. *Development* happened first, on individual laptops or desktops. Developers had access to a shared development server and database, which was often populated with a stale copy of production data. They'd manually copy files to the server via FTP or basic scripts. Version control existed via the Concurrent Versions System (CVS) or Subversion, but it wasn't universal. Git was only invented in 2005.

2. *Testing* came next and was where code got integrated. On a predetermined schedule, the testing server would check out all the code, refresh the database, and execute testing scripts. Unit tests ran first, and integration and regression tests ran next to verify that all the pieces fit together and nothing that was previously working had gotten broken. (We'll talk about testing at the end of this chapter.) Quality assurance (QA) engineers owned this environment, but they had no access to production data.

3. *Acceptance* came after testing, and mirrored production in services and revision levels. This was where developers lost control:

operations and release engineers took over. When code was ready for release, it was checked out onto the Acceptance server. Teams often re-ran integration and regression tests as a final sanity check. This was where the client actually saw and tested the code, using their own test plan.

4. *Production deployment* was the final phase. At the time, Cal optimistically wrote, “If all goes well, this should be a fully automated process—or, at least, a highly automated operation.” In practice, it was anything but. Deployments happened in scheduled maintenance windows, often on weekends, with extensive sign-offs and manual steps.

The DTAP gates weren’t just about catching bugs; they encoded organizational boundaries. Developers owned development and testing. Operations owned acceptance and production. Code was thrown “over the wall” between these silos.

This made sense in a world where the following things were true:

- Environments were expensive physical servers, which were impossible to reproduce reliably.
- Deployments were manual, risky, and infrequent.
- A bad deployment could mean hours or days of downtime, and rollbacks were slow.
- CI/CD tooling barely existed. (Jenkins, then called Hudson, appeared in 2005 but wasn’t mainstream.)

DTAP was a safety net for a world without better options. The organizational separation between dev and ops was a feature, not a bug: it forced deliberate handoffs and human verification at each stage.

But by the late 2000s, DevOps culture began challenging this model. If one team owned the entire cycle from development to production, did it really need four separate environments with formal handoffs among them?

Today, especially for ML/AI development, having three environments (development, acceptance, and production) is enough, for the following reasons:

- ML/AI development comes with certain requirements, such as access to production data in the development environment. This means we achieve a higher level of confidence that the system performs as expected at an earlier stage.
- IaC makes environments reproducible. We can deploy to development, acceptance, and production environments with identical configurations and security guidelines. This solves environment drift, which is one of DTAP's core problems.
- CI/CD is mature. Automated pipelines can build, test, and deploy code in minutes. The manual handoffs that DTAP formalized are now bottlenecks, not safeguards.
- Teams that follow DevOps principles and own code from development through production don't need organizational gates between environments.

However, you need to be aware that having four environments may still be necessary in regulated industries, such as critical infrastructure, finance, and health care, where multi-environment gates ensure safety for the users and serve as proof of compliance.

The Big Book of MLOps,[2] which was developed by Databricks, also mentions three environments: `dev`, `acc`, and `prd`. We use the three-environment setup in this book as well.

Platform Best Practices

Let's discuss an example of a Databricks platform setup. It's common for each individual product, a business unit, or a team to have a set of Databricks workspaces, one per environment—`dev`, `acc`, and `prd`—and a

schema or a set of schemas in environment-specific catalogs (`mlops_dev`, `mlops_acc`, and `mlops_prd`). Figure 5-4 illustrates this structure.

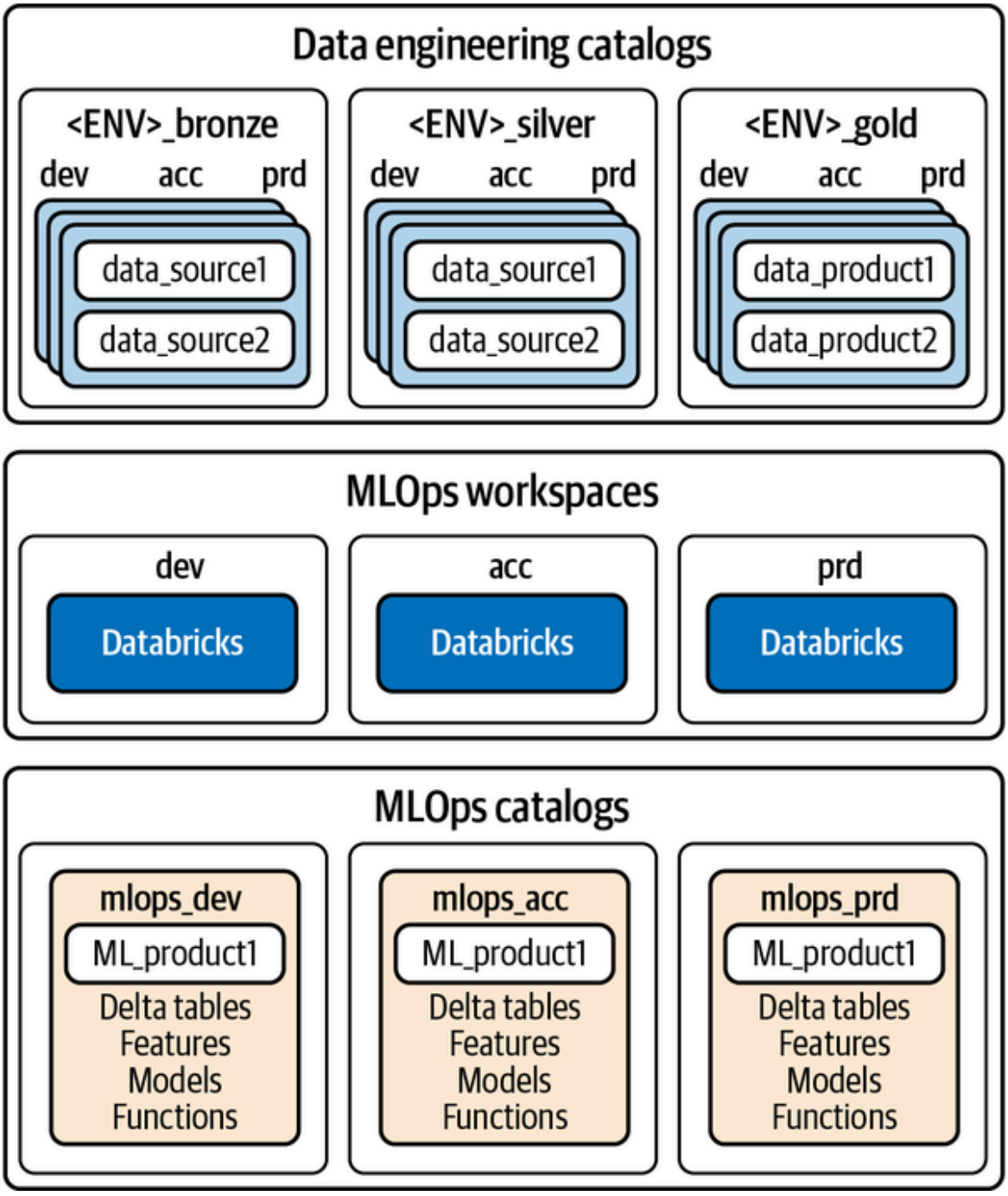


Figure 5-4. Example of Databricks platform setup for MLOps

ML teams act as both data consumers and data producers: they use curated datasets created by data engineering teams for training the models, and they

generate outputs like predictions or embeddings that are consumed by downstream systems. The data that ML teams consume is stored in catalogs related to data products, while the data they produce is stored under ML product schemas in MLOps catalogs.

Data engineering teams commonly organize source data by using the medallion architecture, with separate catalogs for each environment and layer: `<env>_bronze`, `<env>_silver`, and `<env>_gold`:

Bronze

: This layer holds raw, ingested data.

Silver

: This layer includes cleaned and structured tables.

Gold

: This layer contains business-ready datasets.

Each data source or product has a dedicated schema, which enables clear ownership and structured access control.

An important best practice is for the platform team to be able to read production data during development while strictly controlling where they can write data. In practice, ML teams should be able to read data product schemas from the production Gold catalog (`prd_gold`), even when working from the development workspace. This ensures that training and inference use the same data definitions and quality guarantees across all environments.

Lower-environment copies of data often differ in subtle but important ways (e.g., missing history, relaxed quality checks, schema drift). This can lead to issues that only appear in production, but using production Gold data for development significantly reduces these risks and helps minimize model drift.

At the same time, write access must remain environment specific. From each ML workspace (`dev`, `acc`, and `prd`), teams can write data only to their own schemas in an environment-specific ML catalog (`mlops_dev`, `mlops_acc`, or `mlops_prd`). This prevents accidental pollution of production assets, keeps environment promotion explicit, and maintains clear ownership boundaries.

For platform teams, this pattern balances consistency and safety. Data product teams retain ownership of Gold datasets in `prd_gold`, while ML teams consume them as read-only dependencies. All ML-generated artifacts are isolated in ML-owned catalogs. With Unity Catalog governance and read-only permissions, exposing production data for development is low risk and avoids the operational burden of maintaining multiple "production-like" data copies.

This structure enables modular, secure development across teams. Data engineers focus on building trusted data products, while ML teams build and deploy models independently on a consistent foundation of production-grade data.

I've just mentioned a number of things related to permissions, and that raises another important point: when you're designing a Databricks platform, you need to have an access control strategy.

Access Control Strategy

In Databricks, access controls are applied at two distinct levels: the workspace level and the Unity Catalog level. Access is governed through securables—objects that can have permissions assigned to them. See Figure 5-5 for a list of securables.

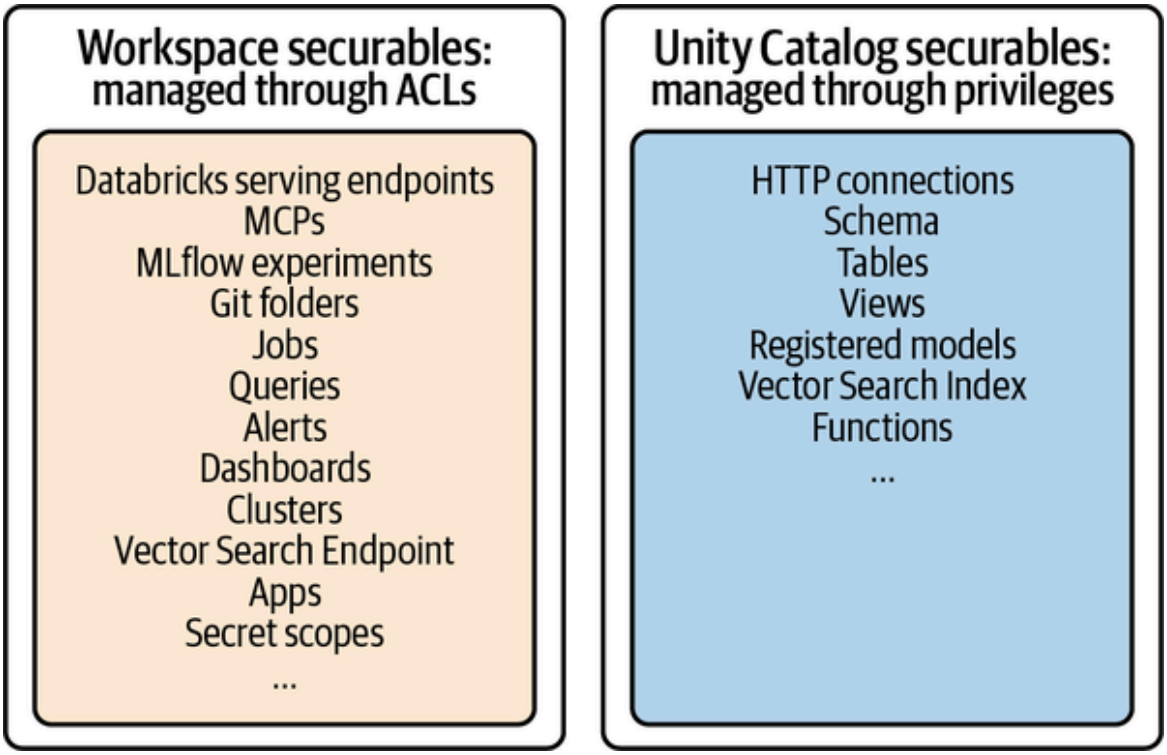


Figure 5-5. Workspace and Unity Catalog securables

Workspace securables include items like notebooks, dashboards, endpoints, clusters, and jobs, and access to them is managed within each individual workspace. *Unity Catalog securables* include catalogs, schemas, tables, and other data or AI assets, and access to them is defined at the metastore level.

Workspace securables are managed through access control lists (ACLs).[3] Workspace admins have the `CAN_MANAGE` permission on all objects in their workspace (including clusters), while users have the `CAN_MANAGE` permission on objects they create.

Unity Catalog securables are managed through privileges.[4] Metastore admins can create catalogs, manage permissions, and transfer ownership for any object, read and update the metadata of all objects in the metastore, and even delete a metastore. While metastore admins don’t have direct read or write access to data by default, they can grant themselves access by transferring ownership.

Consider a ML engineer working on two projects simultaneously: hotel booking pricing and demand forecasting. Each project has its own set of

Databricks workspaces and Unity Catalogs. To prevent the engineer from accessing demand forecasting data while working in the hotel booking pricing workspace, you can use workspace binding.

Unity Catalog supports two access modes for securables: `OPEN` and `ISOLATED`. In `OPEN` mode, the securable can be accessed from any workspace. In `ISOLATED` mode, access is restricted to a specified list of workspaces.

Let's review how we can organize the permissions for a team working on the hotel booking pricing project. Figure 5-6 outlines the permissions for the team and service principals on the workspace level.

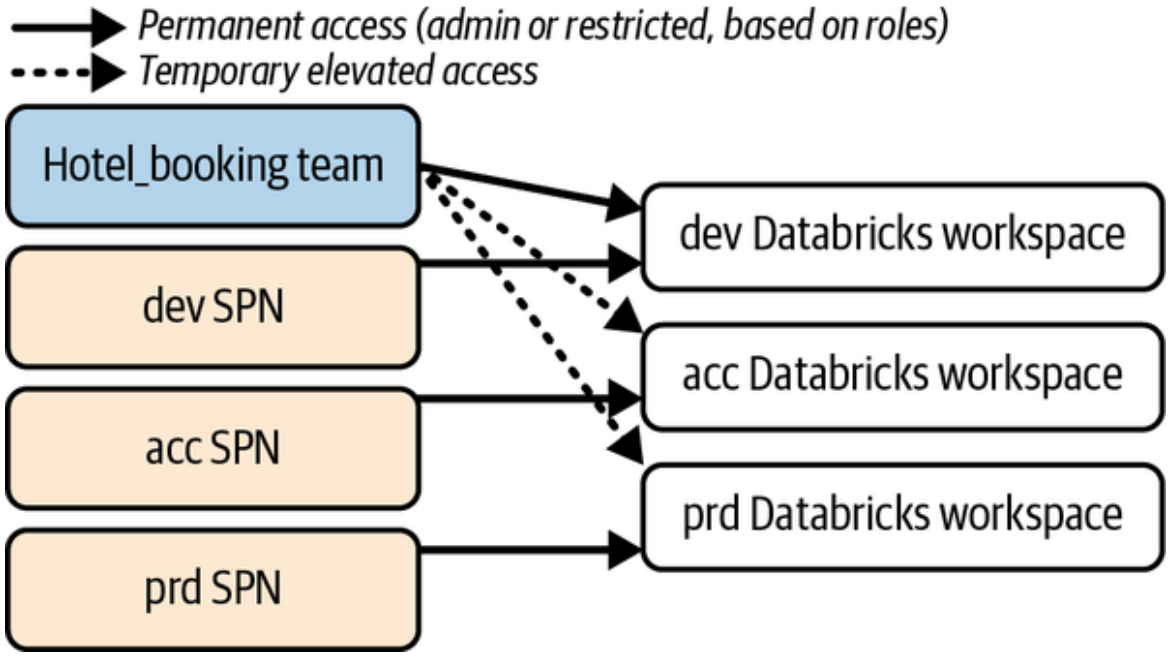


Figure 5-6. Project workspace permissions

The team should have permanent access to the development Databricks workspace, where most experimentation, model development, and collaboration happen. Workspace admin access should be limited to senior engineers or tech leads. Granting admin rights to all users is risky, as it enables unrestricted cluster creation and other unnecessary permissions. A better approach is to create a Databricks group and assign cluster policies that limit what kind of clusters that group can create. It is also important to keep in mind that this setup introduces collaboration challenges in

development environments. The user who creates an object—such as a notebook, query, job, or Model Serving endpoint—must explicitly grant the appropriate permissions to the group so that other users can view, edit, or run shared assets.

Access to the acceptance (`acc`) and production (`prd`) workspaces should be restricted by default and only granted temporarily when needed, such as for testing deployment pipelines or investigating production issues. This temporary elevated access minimizes risks while still supporting operational needs.

Additionally, service principals (SPNs) are used for automation and CI/CD pipelines. Each environment has a dedicated SPN with scoped permissions that are limited to its corresponding workspace. This separation ensures that automation processes only operate within their intended boundaries and follow the principle of least privilege.

Figure 5-7 shows the permissions for the team and service principals on the Unity Catalog level.

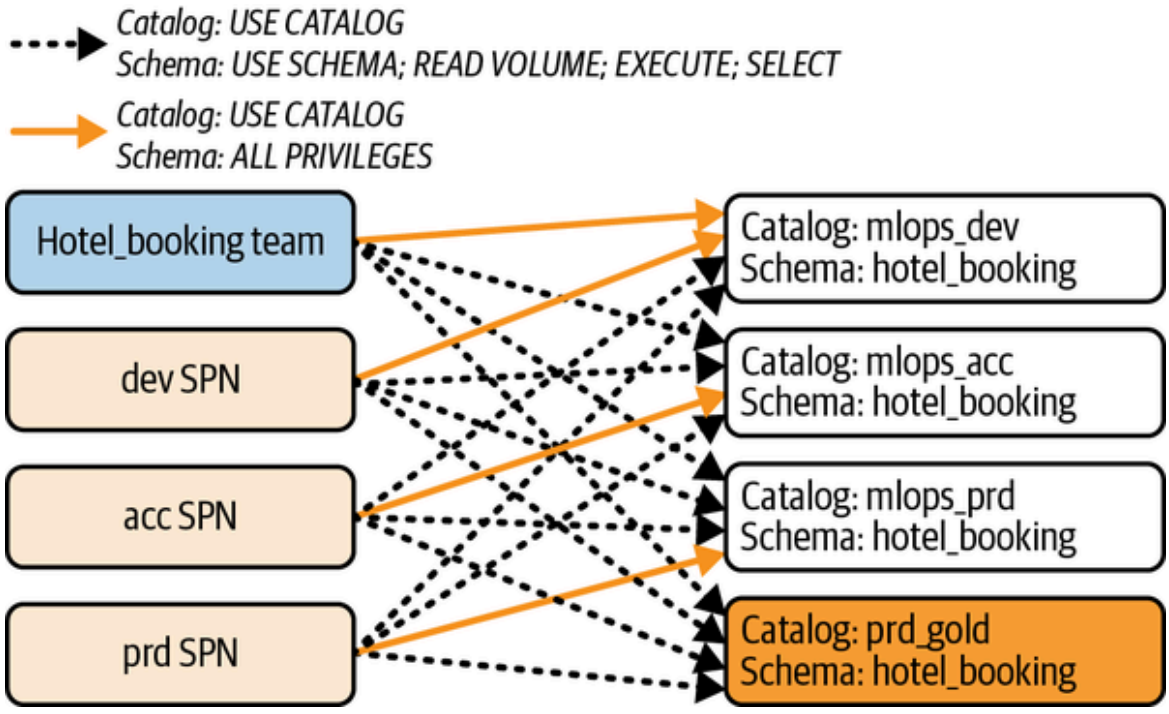


Figure 5-7. Project Unity Catalog privileges

The hotel booking team has `ALL_PRIVILEGES` permissions on the `mlops_dev.hotel_booking` schema. In other environments (`mlops_acc`, `mlops_prd`, and `prd_gold`), they have read-only access (`SELECT`, `READ_VOLUME`, and `EXECUTE`), which allows them safe use of production data for training and validation.

SPNs have `ALL_PRIVILEGES` access on schemas in the corresponding catalogs (`mlops_dev`, `mlops_acc`, and `mlops_prd`) and read-only access to `prd_gold.hotel_booking` to ensure consistent data consumption across environments, without risk of modifying production data. The privileges on the schema level are propagated to all underlying objects.

Promoting Code and Models to Higher Environments

Now that we've covered how different environments (`dev`, `acc`, and `prd`) are structured and secured, we can look at how code and models move among them. First, let's talk about code promotion, which is tightly coupled with Git branching strategy. Let's explore how Git serves as the backbone of deployment in a multi-environment Databricks setup.

Git Branching Model

Anyone who uses version control has probably come across Gitflow, which is a branching model created by Vincent Driessen. After Driessen published the post titled "A Successful Git Branching Model" in 2010,[5] Gitflow became the default branching strategy for many teams over the following years.

This model assumes two default branches, development and main, along with multiple long-lived feature and release branches. This structure often leads to unnecessary complexity and a lot of branches to manage (see Figure 5-8).

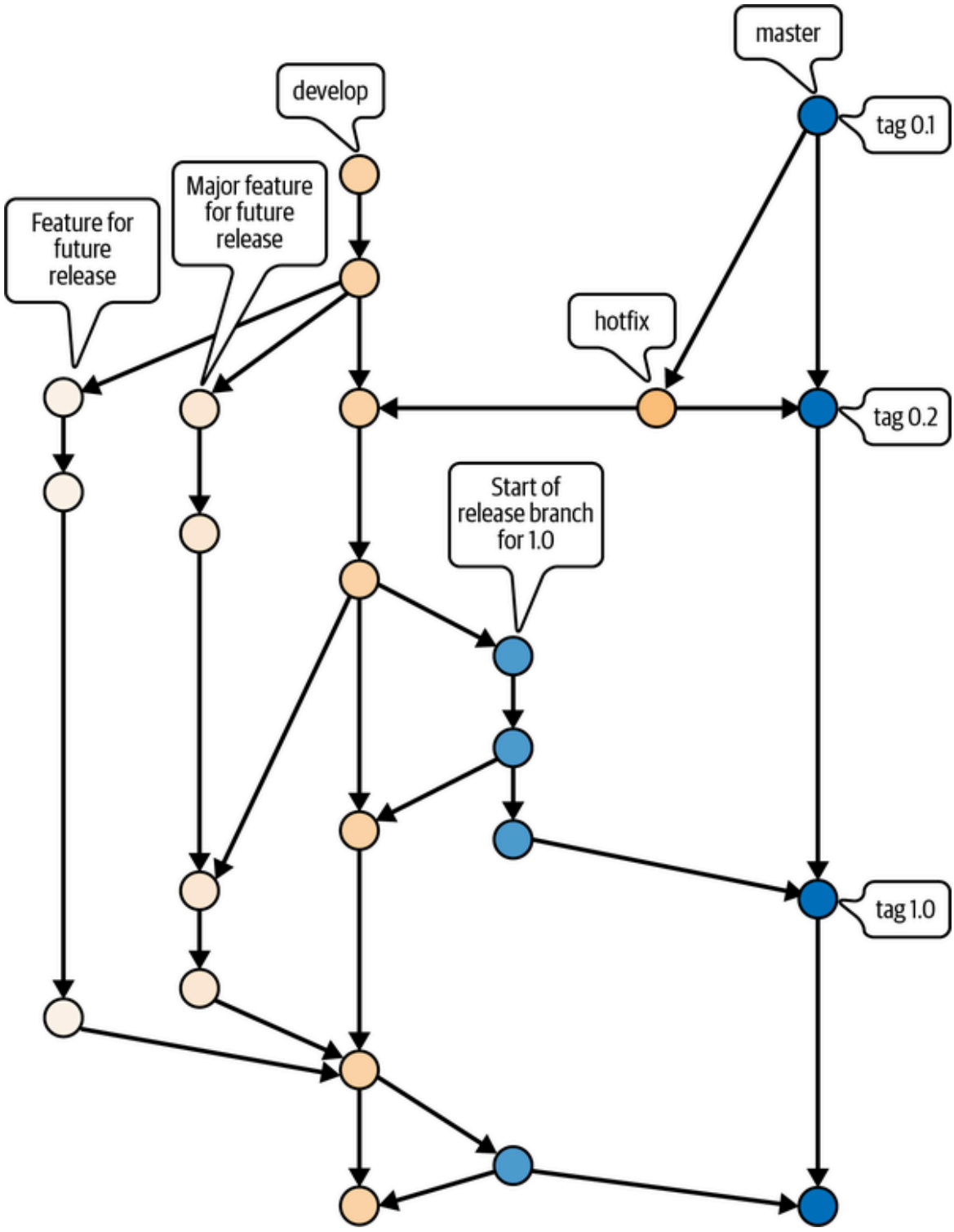


Figure 5-8. Gitflow, adapted from "A Successful Git Branching Model"

As Vincent mentions in his note of reflection for his original blog,[6] dated 2020, the world of software has changed since 2010, and for most teams, it

is no longer necessary to support multiple versions of software. For modern applications that require continuous delivery, he recommends a simpler approach called GitHub flow, which is a Git branching model developed and used by GitHub. In this model, developers work on short-lived feature branches and create pull requests to the main branch. This development has become popular over the past few years because of its simplicity.

In the past, I've tried to adapt Gitflow to ML projects, and I've found it overly complicated. All the ML projects I work on follow the GitHub flow development model, which is easier to understand and does not create unnecessary overhead for ML teams.

Here's how the GitHub flow approach works:

1. A feature branch feature/foo is created from the main branch. After a number of commits, the branch is ready.
2. A developer creates a pull request (PR).
3. At PR to the main branch, a CI pipeline run gets triggered.
4. It runs pre-commit checks and unit tests. An approval is required to merge the pull request.
5. When the feature branch is merged to main, a CD pipeline run gets triggered and targets two environments: acceptance and production.
6. The CD pipeline in the acceptance environment can run directly, but deployment to the production environment should be approved by a reviewer first. Usually, some manual checks on the acceptance environment are required to make sure everything works as expected.
7. Figure 5-9 provides a simplified illustration of this GitHub flow approach.

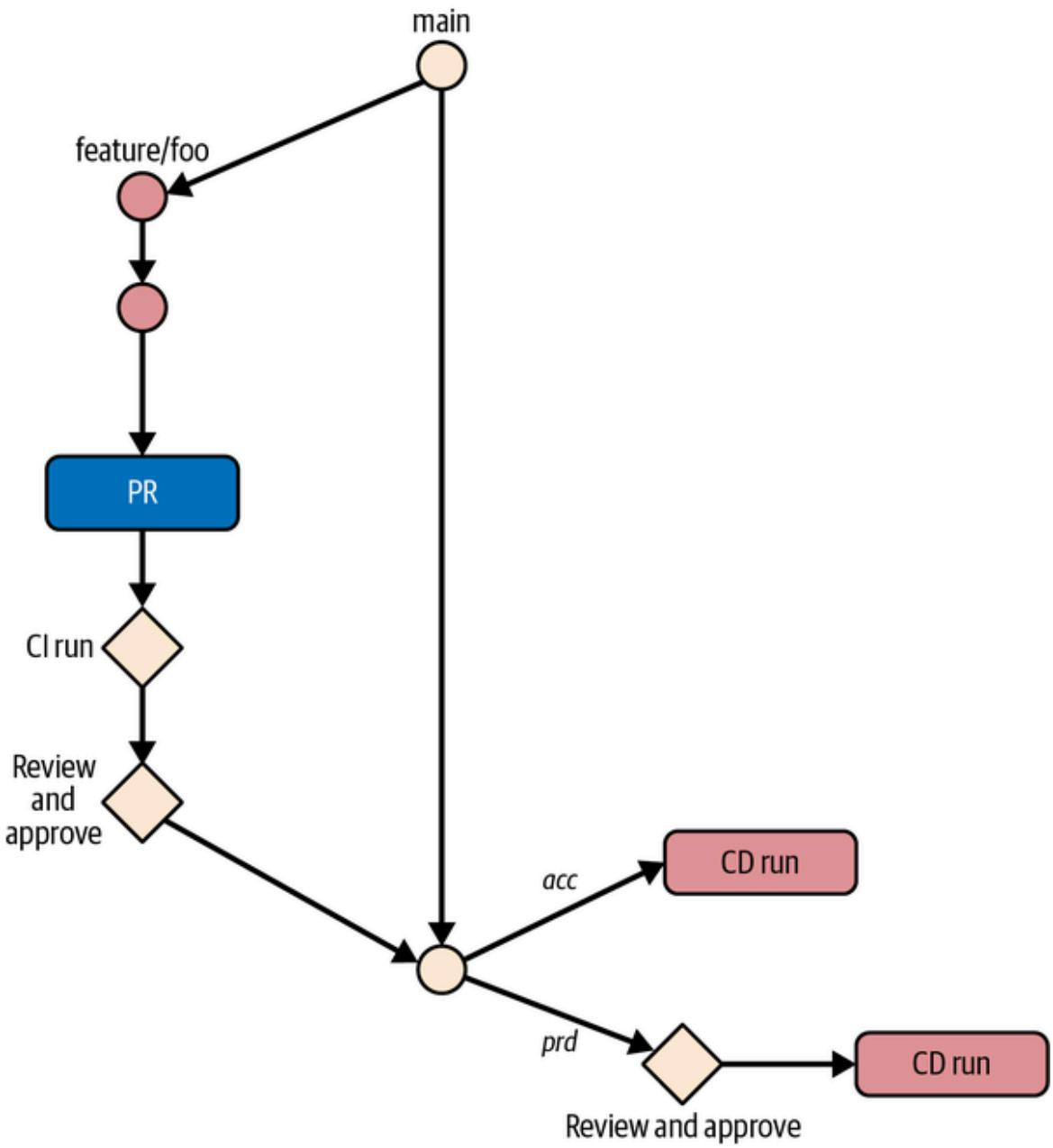


Figure 5-9. Simplified GitHub flow

Let’s now take a close look at the CI and CD pipelines, which I mentioned previously when explaining the Git branching strategy.

Continuous Integration Pipeline

Let’s take a look at the continuous integration (CI) pipeline for the MLOps part of the book (from the *ci_mlops.yml* file in the *.github/workflows*

folder), which is triggered to run on each PR to the main branch if files in the *mlops* folder have changed:

```
on:
 pull_request:
   branches:
     - 'main'
   paths:
     - 'mlops/**'
```

The pipeline has one job. The first step in this job is to check out the repository under GitHub workspace, so the workflow can access the code. Notice that we fetch the history of all branches and tags by setting `fetch_depth` to `0` (the default is `1`), to make sure that the next step can run correctly:

```
jobs:
 pytest_and_checks:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd #v6.0.2
       with:
         # Fetch all history for all branches and tags
         fetch-depth: 0
         ref: ${{ github.head_ref }}
```

The following steps install uv and all the dependencies, and they run pre-commit checks and PyTest:

```
- name: Install uv
  uses: astral-sh/setup-uv@0cec208311dfd045dd5311c1add060b2062131d57 #v8.0.0

- name: Install the dependencies
  run: |
    uv sync --extra ci

- name: Run pre-commit checks
  run: |
    uv run pre-commit run --all-files

- name: run pytest
```

```
    run: |
      uv run pytest
```

It’s also important to note that we do not use tags but instead commit hash for public GitHub Actions. This is a good security practice that minimizes the risk of malicious attacks, which can happen if the tag gets overwritten by a malicious actor.[7]

Continuous Deployment Pipeline

Let’s take a look at the continuous deployment (CD) pipeline, which is triggered to run on push to the main branch if files in the `mlops` folder have changed:

```
on:
 push:
   branches:
     - 'main'
   paths:
     - 'mlops/**'
```

This pipeline defines a single deployment job with a matrix strategy, which allows the same job definition to run across multiple environments using variable combinations. In this case, the job runs twice (once each in the `acc` and `prd` environments), based on the values defined in the matrix.

GitHub environments are used to manage environment-specific variables, secrets, and federated credentials under a consistent name. By referencing `${{ matrix.environment }}`, the job dynamically applies the appropriate configuration for each run.

The job requires specific permissions:

id-token

> This is write permission to fetch an OpenID Connect (OIDC) token for authentication.

contents

> This is write permission to push a Git tag after a successful production deployment.

The `DATABRICKS_HOST` environment variable is set using the GitHub environment-specific variable with the same name:

```
jobs:
 deploy:
   runs-on: ubuntu-latest
   strategy:
     matrix:
       environment: [acc, prd]
   environment: ${{ matrix.environment }}
   permissions:
     id-token: write # to fetch openID token
     contents: write # to push tag
   env:
     DATABRICKS_HOST: ${{ vars.DATABRICKS_HOST }}
```

The first step checks out the repository under GitHub workspace:

```
steps:
  - name: Checkout Source Code
    uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd #v6.0.2
```

In this example, we use Amazon Web Services (AWS), so we configure AWS authentication and retrieve Databricks credentials from AWS Secrets Manager. We could have used a similar process for Azure or Google Cloud Platform (GCP). This setup enables secure access to Databricks during deployment. Note that in such a case, our cluster specification would need modification as well.

The second step uses the `configure-aws-credentials` GitHub action to assume an identity and access management (IAM) role defined in the `ARN_ROLE` variable and then set the AWS region. This grants the pipeline temporary credentials to access AWS resources.

The third step fetches secrets from AWS Secrets Manager by using those credentials. The secret contains two key-value pairs: `DATABRICKS_CLIENT_ID` and `DATABRICKS_CLIENT_SECRET`. These values are exported as environment variables with the same names, which are then used to authenticate with Databricks when running `databricks bundle deploy`. Note that the Databricks CLI requires three environment variables for authentication with the service principal: `DATABRICKS_CLIENT_ID`, `DATABRICKS_CLIENT_SECRET`, and `DATABRICKS_HOST`:

```
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials
        @ec61189d14ec14c8efccab744f656cffd0e33f37 #v6.1.0
  with:
    role-to-assume: ${{ vars.ARN_ROLE }}
    aws-region: eu-west-1

- name: Fetch secrets
  uses: aws-actions/aws-secretsmanager-get-secrets
        @2cb1a461cbd4865ac4299648312e4704c646cd53 #v3.0.1
  with:
    secret-ids: |
      , ${{ vars.SECRET_ID }}
    parse-json-secrets: true
```

The final steps in the pipeline prepare the deployment environment and handle promotion logic for production. First, the Databricks CLI is installed using the official setup action. This is required to run the `databricks bundle deploy` command, which publishes the bundle definition to the target environment.

Next, the uv package manager is installed. It's used to build any Python packages defined in the bundle configuration. Finally, the deployment step runs `databricks bundle deploy`, passing the current `git sha` as a variable. Note that deploying a bundle does not trigger any jobs. Instead, jobs will run based on defined triggers within the Databricks workflow:

```
- name: Install Databricks CLI
```

```
      uses: databricks/setup-cli@2260866f83a41a2df55e1cfe7ffe038b78325bf6
    #v0.295.0
      with:
        version: 0.295.0

    - name: Install uv
      uses: astral-sh/setup-uv@0cec208311dfd045dd5311c1add060b2062131d57 #v8.0.0

    - name: Deploy to Databricks
      env:
        DATABRICKS_BUNDLE_ENV: ${{ matrix.environment }}
      run: databricks bundle deploy --var="git_sha=${{ github.sha }}"
```

In our CD pipeline example, we have authenticated with AWS using federated credentials that are set up for each GitHub environment. However, before we can do that in the real world, we need to create a set of service principals on Databricks and a set of IAM roles, policies, and secrets on AWS.

OpenID Connect

Figure 5-10 shows an example of the setup. It assumes that we have three different Databricks workspaces (`dev`, `acc`, and `prd`). Each of these workspaces contains a service principal, and the client secret and client id of these service principals are stored in the respective secrets in AWS Secret Manager. We also need to create custom IAM policies that allow us to read these secrets from the secret manager. These policies are attached to the respective IAM roles, which are assumed when CI/CD pipelines get executed under their respective environments.

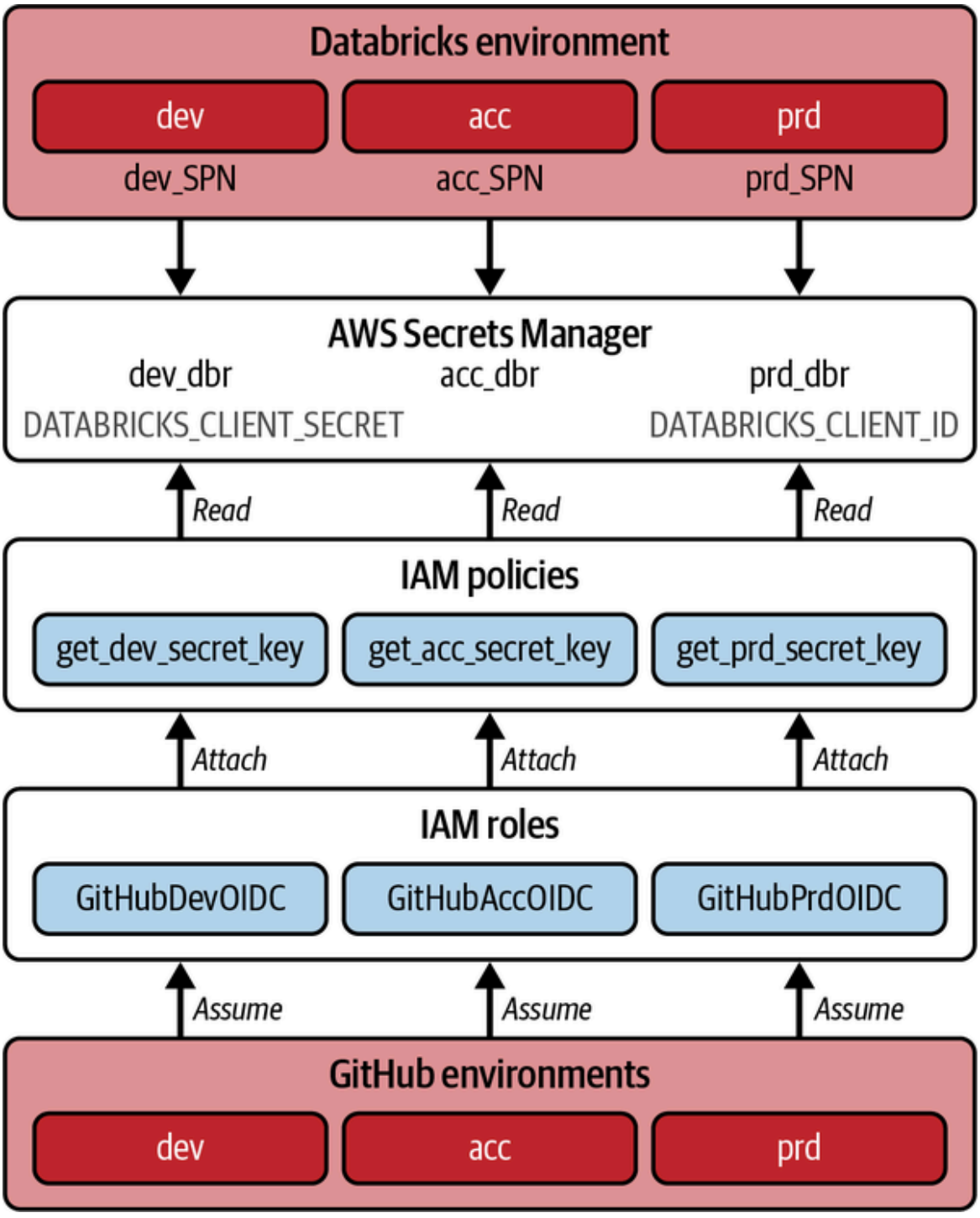


Figure 5-10. IAM roles and policies

You can find the trust policy definition for the IAM role. Please note that you must create an OpenID Connect identity provider by setting `provider`

to **`token.actions.githubusercontent.com`** and audience to **`sts.amazonaws.com`**:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated":
    "arn:aws:iam::ACCOUNT_ID:oidc-
provider/token.actions.githubusercontent.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "token.actions.githubusercontent.com:aud": "sts.amazonaws.com"
        },
        "StringLike": {
          "token.actions.githubusercontent.com:sub":
            "repo:ORG_NAME/REPO_NAME:environment:ENVIRONMENT_NAME"
        }
      }
    }
  ]
}
```

In our setup, we do not use GitHub secrets. Instead, we use the OIDC setup, which is generally considered to be a more secure solution. However, we use environment variables that contain different values in each environment. Figure 5-11 shows the view of configured environments in GitHub.

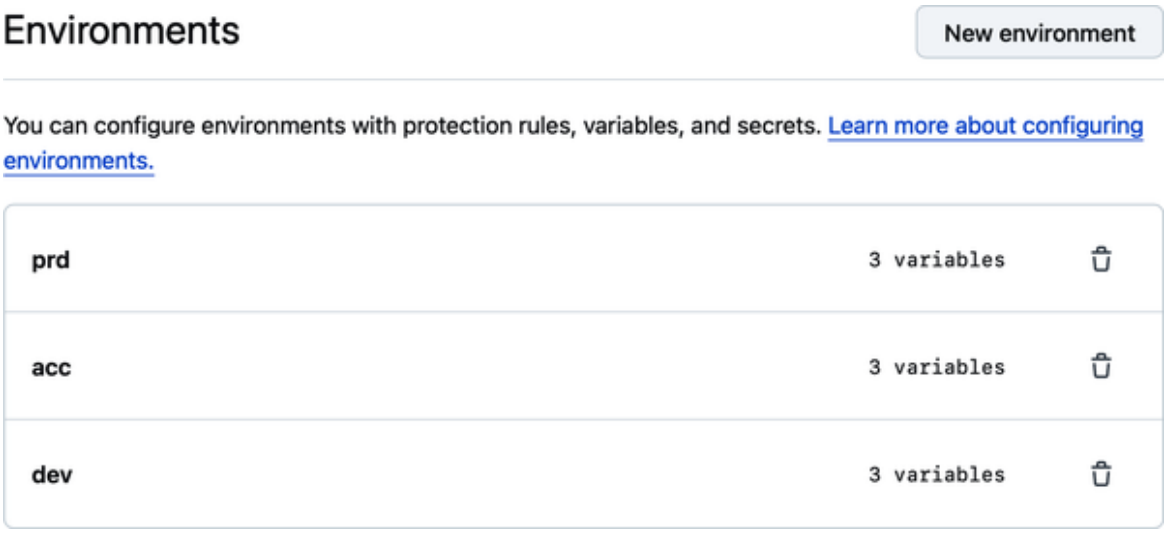


Figure 5-11. Environments in GitHub repository

Here are example values for the variables in the development environment:

- `ARN_ROLE`: arn:aws:iam::123456789:role/GitHubDevOIDC
- `DATABRICKS_HOST`: https://dbc-c2e8445d-159d.cloud.databricks.com
- `SECRET_ID`: dev_dbr

Using CI/CD pipelines alone is not sufficient to ensure secure deployments. When I described the GitHub flow, I mentioned the approval step multiple times. We can configure this step by using the branch and deployment protection rules to enforce a manual review before code is merged or deployed. Let's dive into it.

Branch and deployment protection rules

Implementing branch protection rules is essential for preventing developers from committing directly to important branches, like main. GitHub offers rich functionality for these rules, but some of them are more critical than others. While features like requiring status checks to pass are nice to have, I believe the absolute minimum for any project is to enable the "Require a pull request before merging" rule, along with the sub-options shown in Figure 5-12.

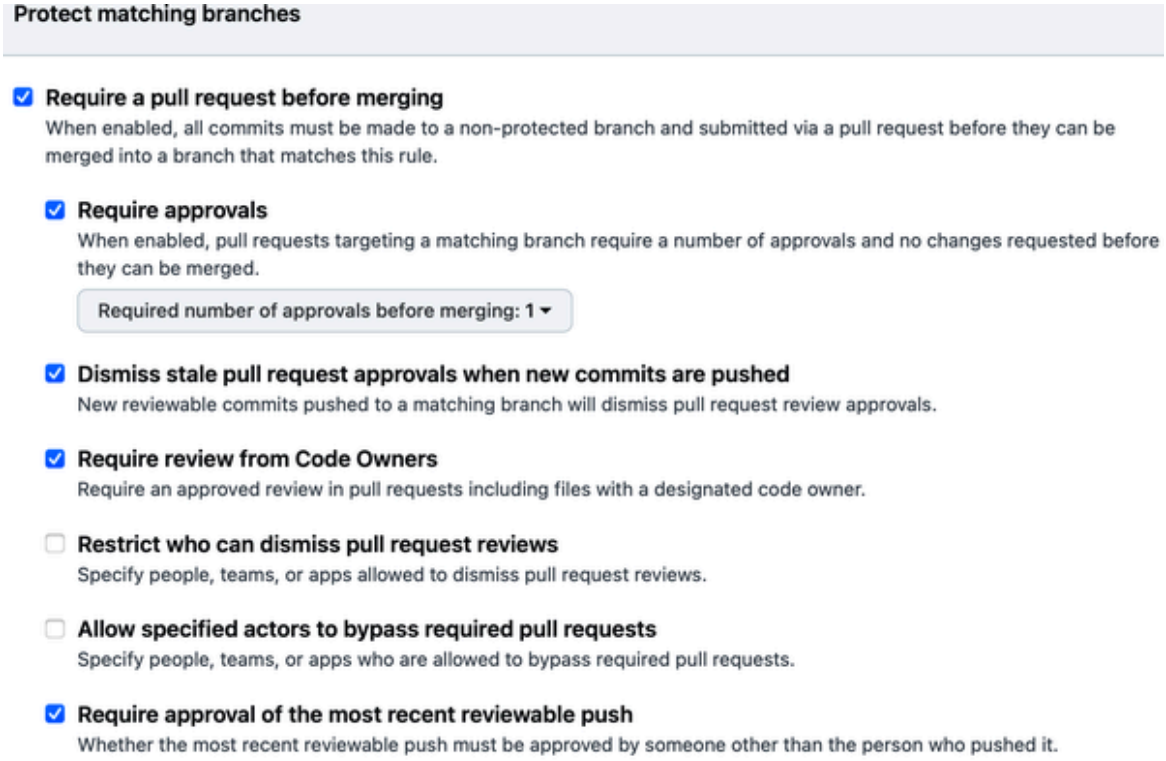


Figure 5-12. Recommended minimal branch protection rules

These include the following:

- Requiring at least one (ideally, two) approvals
- Requiring a review if a new commit is pushed after the pull request is approved
- Using the CODEOWNERS file: automatically assigning reviewers based on the code being changed
- Enforcing approval: requiring the most recent push to be approved by someone other than the person who pushed it

I also recommend enabling the “Do not bypass the above settings” option to ensure these rules cannot be circumvented.

I mentioned earlier that deployment to production should be approved by a reviewer. We can enable this by using the deployment protection rules that are selected in Figure 5-13, which we can set for each environment and

repository. These rules provide necessary guardrails without making the CD pipelines more complicated than they need to be.

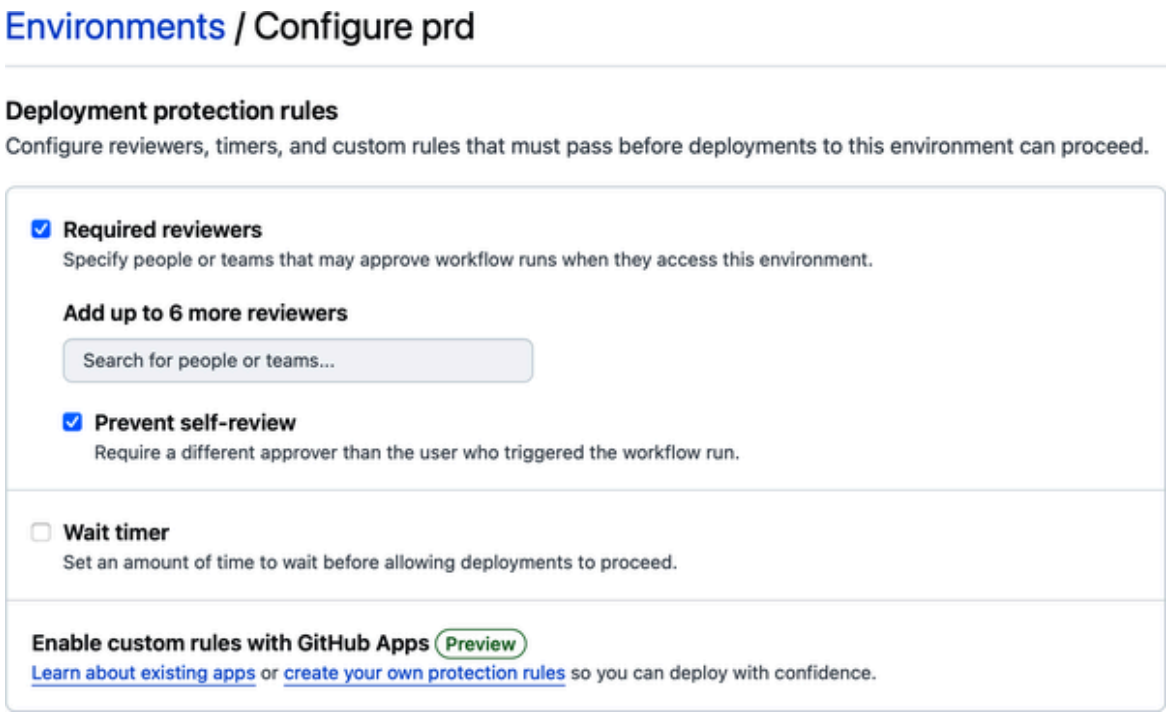


Figure 5-13. Deployment protection rules

So far, we've explored key considerations for promoting code across environments to ensure secure and reliable deployment. We assumed two things: models are retrained independently in each environment, and retraining is stateless, starting from scratch every time. But this isn't always realistic.

For deep learning models, which can be large and expensive to train, it's often preferable to resume training from previously saved weights. In these cases, you typically want to train the model once, often in a development environment, and promote it without retraining. However, copying models between Unity Catalogs (e.g., from `dev` to `acc` to `prd`) is not the best approach. Instead, it is more efficient to use a shared Unity Catalog, where the model is stored once and promotion is handled through aliases and approval workflows, rather than physical duplication. Let's discuss how you can accomplish this.

Promoting Models to Higher Environments

Let's say the ML team is working on updating the weights for the hotel booking embedding model in the Databricks development workspace. The resulting model is registered in the central Unity Catalog, and it should eventually be deployed behind a Model Serving endpoint in the Databricks production workspace. Figure 5-14 demonstrates how to do this in a secure way.

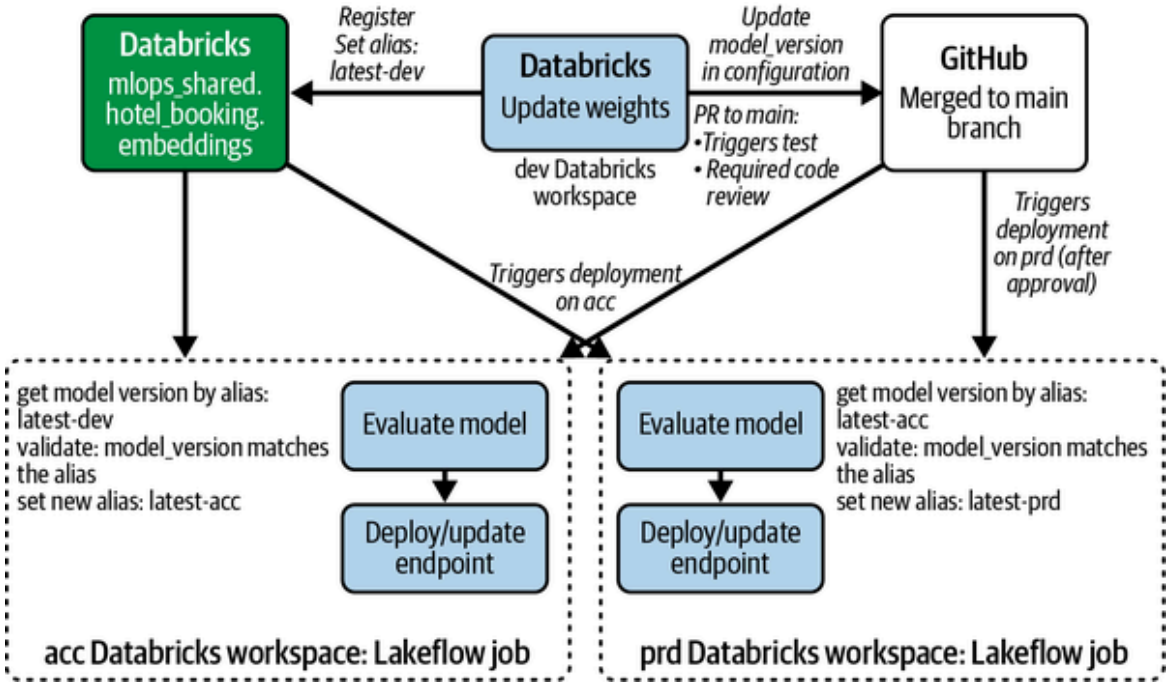


Figure 5-14. Promoting a model to higher environments

In this workflow, the embedding model is managed centrally in the `mlops_shared.hotel_booking.embeddings` catalog and promoted across environments using MLflow aliases. The model is large, so it cannot be trained locally. After updating weights in the development workspace, the model is registered and assigned the `latest-dev` alias.

In the simplest scenario, the model update did not require the code change, but we still need to store references to the updated model version in the configuration file (for example, *project_config.yml*). A PR to the main branch gets created, and it triggers tests and requires code review. After the code is merged to the main branch, the CD pipeline gets triggered, and

deployment to acceptance and production gets executed there. Deployment to production is protected and requires approval. Earlier in the chapter, we looked at the CD pipeline, which only updates the Lakeflow job definition, but it can also trigger the execution of the Lakeflow job.

In the acceptance environment, the Lakeflow job gets the model version by `latest-dev` alias, verifies that the model version matches the one defined in the configuration file, and evaluates the model. For embeddings, programmatic evaluation typically involves computing similarity scores or ranking quality using predefined benchmarks or labeled pairs. If the evaluation confirms that the model performs as expected, the alias is updated to `latest-acc` and the model is deployed to the acceptance endpoint. A similar process follows in the production workspace, where we get the model version by `latest-acc` alias, assuming that the pipeline ran in the acceptance environment first.

In the approach we've discussed, the model stays in a shared catalog, and promotion across environments is handled through aliases and code promotion. This is gated by a manual approval step, rather than by duplicating the model. This setup enables efficient, auditable, and environment-aware model lifecycle management.

Earlier, when we built the CI pipeline, testing showed up as a single step. It is easy to treat this as a checkbox and move on, but it deserves a closer look. What actually belongs in that step, and why ML systems ask more of it than ordinary software does, is the subject of the next section.

Testing

Many ML practitioners have converged on the same observation: production incidents frequently originate from data quality issues, schema changes, feature pipeline failures, training-serving skew, deployment mistakes, and monitoring gaps, rather than flaws in the underlying model architecture.

Jeremy Jordan, a core maintainer of PyTorch Lightning, explains what goes into effective testing in his blog.[8] Developing ML models relies on a large amount of traditional software development: processing data, building features, orchestrating training, and exposing interfaces to external systems. Effective testing of ML systems therefore requires both a traditional software testing suite for the surrounding infrastructure and a model testing suite for the trained artifacts.

In this section of the book, we limit ourselves to practical considerations, since testing could be a whole book on its own. You can find test examples in the repository that accompanies this book.

ML Model Testing

Model tests are divided into pre-train tests and post-train tests:

Pre-train tests

> These run before training and short-circuit a wasted job: the output shape matches the labels, class probabilities sum to one, a single gradient step decreases the loss, and there is no label leakage between the training and validation sets.

Post-train tests

> These interrogate the behavior the model actually learned. They include the following kinds of tests:
>
> *Invariance tests*
>
> > These are tests for perturbations that must not change the prediction.
>
> *Directional expectation tests*
>
> > These are tests for perturbations that must change the prediction in a known direction (e.g., more bathrooms should not lower a predicted house price).

Minimum functionality tests

These are tests on critical data slices.

It is also important not to confuse model evaluation with model testing:

Model evaluation

This summarizes performance and tells you whether the model is good on average. We performed model evaluation earlier in this book, before registering a model for deployment.

Model testing

This consists of explicit checks for specific behaviors the model must exhibit.

The distinction between these two matters, because an evaluation can improve overall while introducing a regression or bias on an important slice of data.

There are also tests for data specifically: assertions about datasets, schemas, null and range constraints, and distribution expectations. On Databricks, DQX provides declarative checks of this kind, and the same idea extends to monitoring the input tables that feed your processing pipelines.

We won't go deep into any of these here. They sit on top of a more fundamental layer. The rest of this section focuses on software engineering tests and their Databricks-specific considerations, because that foundation is where ML teams most often underinvest. For concrete model testing examples, read Eugene Yan's blog.[9]

Software Testing

For decades, software engineering has organized tests around the *test pyramid*: there are many fast unit tests at the base, fewer integration tests in

the middle, and a small number of end-to-end tests at the top. Figure 5-15 shows the test pyramid on Databricks that follows the same idea. We'll explain the details later.

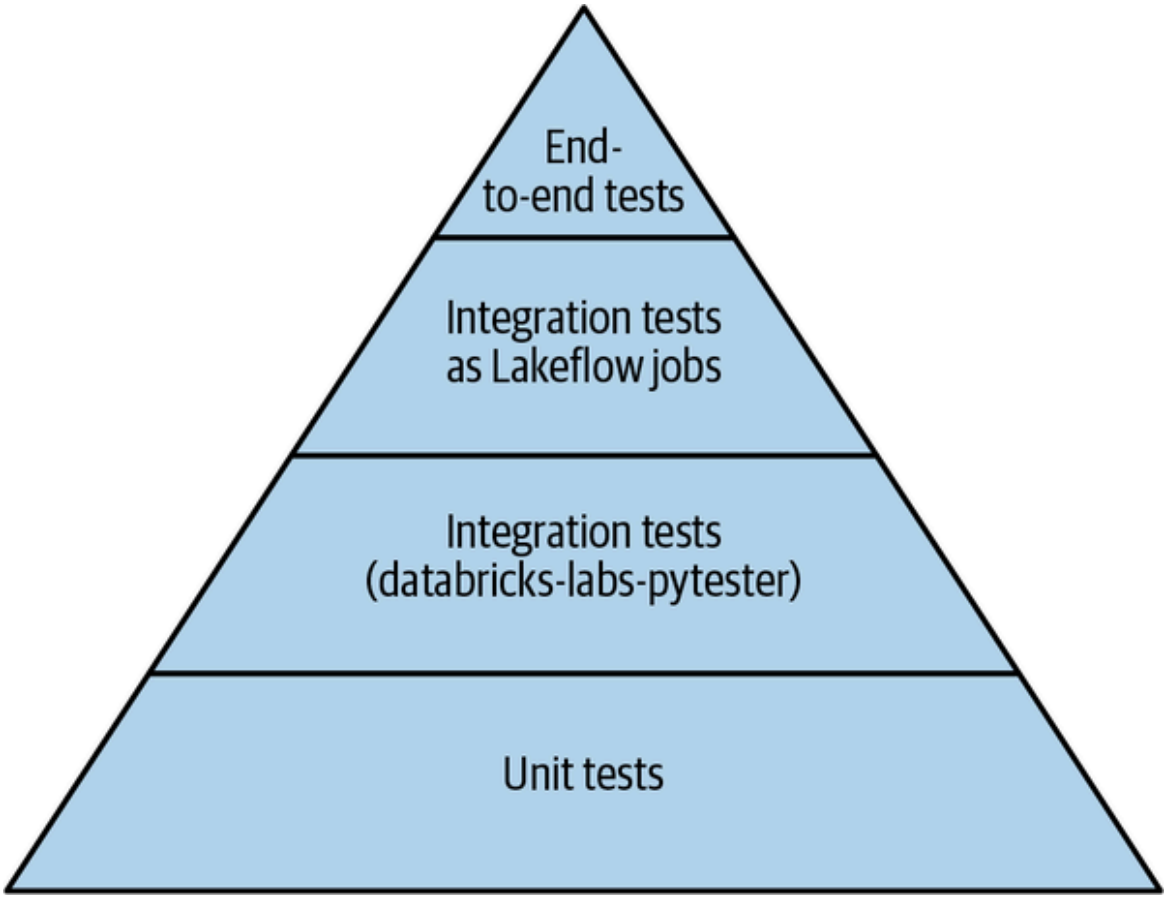


Figure 5-15. Test pyramid on Databricks

The trade-off is simple: as tests become more realistic, they also become slower, more expensive, and harder to debug. The goal is to catch everything you can at the cheapest layer and reserve higher-level tests for what only they can verify. The same model applies to ML systems on Databricks.

There is a lot of great literature on software testing. These are the ones we particularly recommend:

- *Effective Testing: From Slop to Software* by Matt Harrison (self-pub.)

- *Python Testing with pytest* by Brian Okken (O'Reilly)

Let's go through the layers one by one, but not in too much detail, as whole books are devoted to this topic.

Unit testing

A *unit test* isolates a small piece of logic and runs entirely locally: with no Databricks workspace, no cluster startup, no Unity Catalog access, and no network calls. These are the cheapest tests you have: they run in milliseconds, give deterministic feedback, and execute on every commit. They catch obvious mistakes early, and with AI-assisted development tools, writing them is practically free. There is no longer any excuse to skip them.

Your code does use actual SDK clients, and since we do not want to use them, we need to mock. Python's `unittest.mock` and `pytest-mock` make this straightforward.

You should mock only at genuine system boundaries such as REST APIs, SDK clients, and storage systems, and you should use real objects for code within your application. Excessive mocking couples tests with implementation details and produces suites that stay green while the system is broken.

Two rules prevent most mistakes:

- Patch where the name is looked up, not where it is defined. Patch the import used by the module under test, not the module that defines the object.
- Use `autospec=True` whenever possible. A mock with `autospec` derives its interface from the real object, so if a method is renamed, removed, or given different parameters, the test fails immediately. Without it, mocks accept attributes and signatures that no longer exist, so they produce green tests over broken code.

Databricks considerations

For Databricks projects, we reserve unit tests for logic that's expressible independently of the platform: data transformations on plain Python values, configuration parsing, prompt construction, request validation, and business rules. If a test needs a running cluster, a workspace, or a catalog, it belongs in the integration layer.

Spark transformations are where unit testing on Databricks becomes complicated. Ideally, we would test them against a local Spark session that's running entirely in memory. In a standard PySpark project, this is straightforward: we install `pyspark`, create a local `SparkSession`, and run transformations against small test DataFrames.

The complication is that `databricks-connect` and `pyspark` are mutually exclusive. Once `databricks-connect` is installed, local Spark sessions become unavailable. Also, `databricks-connect` is pulled in by the version of `databricks-agents` we use in the book.[10]

One way out is to keep these dependencies out of the project's core library and install them only in the job environment that executes the workload. We can then test the transformation code locally with `pyspark`. This works, but it requires discipline and makes the dependency setup more error-prone. We take the simpler route: we accept that all Spark execution goes through `databricks-connect`, which brings us to integration testing.

Integration testing

There are two levels of integration testing on Databricks:

- The first level *runs Spark logic on Databricks compute*. Once Spark execution goes through `databricks-connect`, tests that exercise Spark logic run remotely rather than locally. The `spark` fixture from `databricks-labs-pytester` is exactly this: a Databricks Connect–backed session, rather than a local one. It still isolates a single transformation, uses small datasets, and gives fast feedback, but its execution is remote. These tests run against

remote Spark on serverless compute, execute quickly, and can be launched directly from an IDE. They occupy the role that local Spark unit tests play in a standard PySpark project.

- The second level *validates an entire pipeline end to end*: workflow orchestration, task dependencies, parameter flow, permissions, and persistence to Unity Catalog. A typical pattern is to create temporary schemas, trigger the deployed job, assert on the resulting tables, and clean up afterward.
- At this level, the job definition itself is part of the system under test. Many failures originate not in the code but in the configuration: there may be a missing wheel, an incorrect `environment_key`, a broken task dependency, or a parameter wired to the wrong task. When pipelines are defined through DABs, these failures are easy to test. The `databricks bundle validate` test catches definition errors before execution starts, and deployment to an isolated target is itself a test. A failed deployment is a failed test.

Taras Chaikovsk describes integration testing patterns on Databricks in detail in the Databricks community technical blog.[11]

Practical considerations: Integration tests in CI

The orchestrated ML pipeline can run as part of CI, on a small data sample, to confirm that the whole thing is wired together. A run that samples data, assembles features, and then trains, evaluates, registers, and deploys a model exercises every wiring decision in a way no isolated test can.

The obstacle is duration. Even on sampled data, a run takes long enough that you do not want a GitHub runner to wait until the job finishes. The pattern that avoids this is asynchronous (see Figure 5-16).

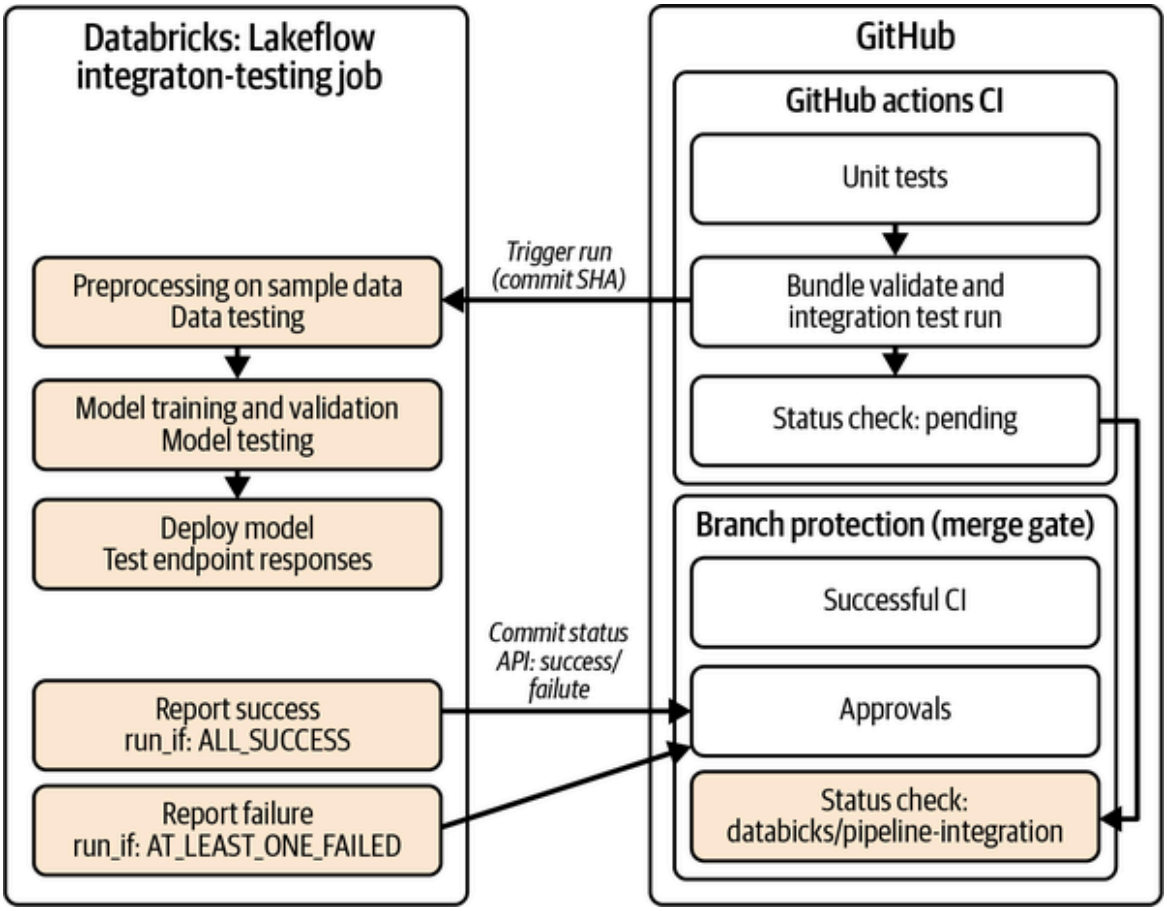


Figure 5-16. Testing pattern with GitHub Actions and Lakeflow jobs

The CI workflow validates the bundle, deploys it to the test target, triggers the job run, sets the commit status to pending through the GitHub Commit Status API, and exits green. The commit SHA is passed in as a job parameter so the run can report back against the right commit.

A commit status can be `error`, `failure`, `pending`, or `success`, and it will surface on any PR involving the commit. A commit status can include `target_url` and `description`: the URL points to the Databricks run output, the description summarizes what happened, and both make the status useful in the GitHub UI. The `context` field names the service that is reporting the status (for example `databricks/pipeline-integration`), so this check stays distinct from other checks on the same commit. The `repo:status` OAuth scope grants access to statuses without granting access to repository code.

The run reports its own results through two tasks at the end of the job. One runs with `run_if: ALL_SUCCESS` and posts `success`. The other runs with `run_if: AT_LEAST_ONE_FAILED` and posts `failure`. Splitting it in two is what makes failure reportable: a single final task runs only on full success, so a failure anywhere would leave the commit stuck on `pending`. Each task knows its outcome from the condition that triggered it, so neither has to inspect the run. Both reporters depend on every pipeline task, not just the last one, so any task that fails is a direct dependency failure the condition can see. The full lifecycle is three writes: `pending` at trigger time and then exactly one of `success` or `failure`.

Branch protection turns that status into a merge gate. You should mark the status context as a required status check. The pull request stays `pending` until a reporter fires, and merging is blocked until the check is green.

Notice that Figure 5-16 also lists data tests, model tests, and endpoint payload tests that confirm the output is what we expect. These can run as part of the Lakeflow job. The first level of integration testing, which consists of the Spark tests on Databricks compute, runs in the CI pipeline alongside the unit tests, since CI already authenticates to Databricks to trigger the job.

End-to-end testing

Integration tests prove that the pipeline works. End-to-end tests prove that the system works, including the consumers downstream.

The model is served behind an endpoint, and a frontend team, a mobile app, or another service consumes it. The contract between the model and its consumer is a real interface that consists of the request format, response schema, latency expectations, error handling, and fallback behavior. It breaks in production like any other interface. End-to-end tests validate it from the consumer’s perspective.

The common failure mode looks like this:

1. The ML team tests the pipeline, evaluates the model, and tests the endpoint in isolation.
2. An incident still happens because the frontend expected one field and the endpoint returned another, or latency doubled after a model update, or the consumer cannot handle a class label the retrained model now outputs.

None of this is specific to ML. It is the contract testing that any service architecture needs. However, ML teams skip it disproportionately because the boundary between the model team and the product team is broader than usual, and "It's done" gets defined as "The model is served."

These tests run in an acceptance environment where a real model is served and either a real consumer or a realistic test harness for it calls the endpoint. These tests are slow and require coordination across teams, so they sit at the top of the pyramid: you need a few of them, but you cannot skip them.

End-to-end tests and production monitoring are two points on a continuum, rather than separate boxes. An end-to-end test validates the consumer contract at a point in time in a controlled environment. Monitoring validates the same contract continuously against real traffic. For ML systems, with their delayed labels and gradually shifting prediction distributions, some of the most important validation can only happen in production. A/B testing, which we briefly covered earlier, is one form of validation that runs against live traffic. Monitoring is the subject of the next chapter.

Conclusion

This chapter moved from model development to a structured deployment strategy built on CI/CD and DABs. The Databricks platform design provides the foundation for reliable and secure ML operations across environments, from development to acceptance to production.

We defined end-to-end ML pipelines declaratively in *databricks.yml*. Configuring jobs as code and passing parameters to Python tasks gives us

traceability, portability, and automation, which are the key enablers of MLOps. The same bundle definition runs in every environment.

We also covered testing for ML systems, from unit tests on individual transformations, through integration tests that prove the pipeline runs, to the end-to-end tests that validate the consumer contract. The pyramid tells you how to expend effort, conducting many fast tests at the bottom and a few slow ones at the top, but skipping none of them.

In the next chapter, we'll shift our focus to monitoring, which is the final piece that closes the loop by ensuring models stay reliable once deployed.

---

1 Cal Evans, "Professional Programming: DTAP—Part 1: What Is DTAP?," *php[architect]*, July 27, 2009, *https://oreil.ly/Jg9yM*

2 Databricks, *The Big Book of MLOps*, 2nd ed. (Databricks, 2023), *https://oreil.ly/r5fYY*.

3 Consult the Databricks documentation for more details on ACLs.

4 Consult the Databricks documentation for more information on privileges.

5 Vincent Driessen, "A Successful Git Branching Model," January 5, 2010, *https://oreil.ly/HzKrc*.

6 The note of reflection is published in the same blog where the original idea was described.

7 For more details, see an article about the GitHub Actions Worm written by Palo Alto Networks.

8 Jeremy Jordan, "Effective Testing for Machine Learning Systems," August 19, 2020, *https://oreil.ly/_4m1E*.

9 Check out his blog post on testing machine learning code and systems.

10 `databricks-agents` removed `databricks-connect` from its requirements in version 1.9.1. If you pin 1.9.1 or later, the conflict does not arise and you can skip this workaround.

11 Taras Chaikovsk, "Integration Testing for Lakeflow Jobs with Pytest and Databricks Connect," *Databricks Community Technical Blog*, December 2025, *https://oreil.ly/hSI23*.

Chapter 6. Monitoring ML Applications

According to The Institute for Ethical AI & Machine Learning, 42% of ML projects do not have monitoring in place (see Figure 6-1). Even though that figure is better than the one from the year before (50% with no monitoring), it shows how immature most companies are when it comes to MLOps.

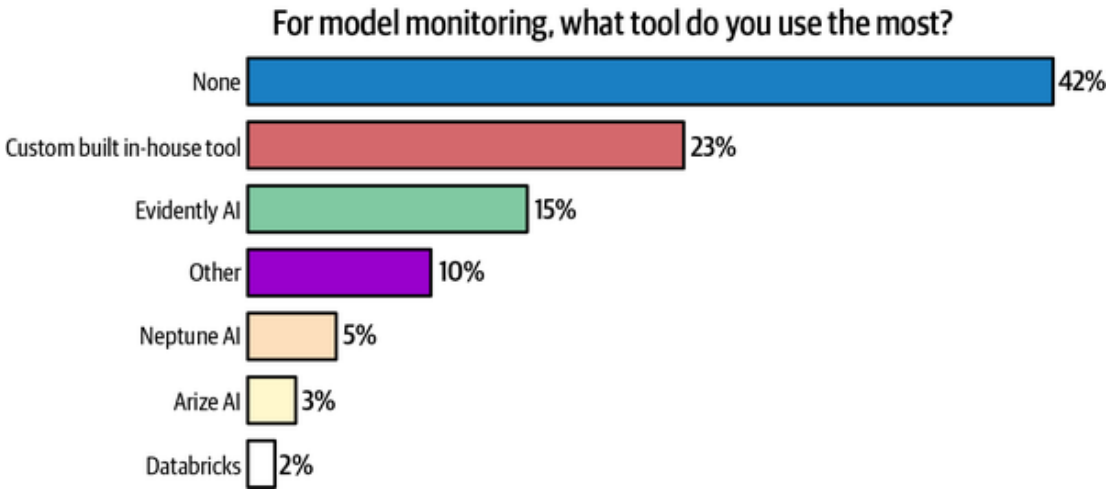


Figure 6-1. Updated image from the State of Production ML in 2025 survey

But monitoring isn't a "nice-to-have." Without it, models fail silently. Predictions degrade. Bias can increase over time. Infrastructure costs can spiral. And perhaps most damaging of all, stakeholders will lose trust in the system when performance suddenly drops and no one knows why. In regulated industries, lack of monitoring can even create legal and compliance risk.

By the end of this chapter, you will understand all of the following:

- The different layers of ML monitoring
- How to design a practical monitoring strategy
- What metrics actually matter in production
- How to set alerts on aspects that matter
- How monitoring connects to retraining and continuous improvement

In short, this chapter will move you from “Model deployed” to “Model managed.”

What to Monitor

Monitoring requirements for ML projects are highly use-case dependent. While generic software metrics are important, ML-specific monitoring focuses on the model’s relationship with changing data and its ultimate impact on business outcomes. Let’s walk through a few examples.

Recommender Systems

In a *recommender system*, such as one for an online grocery store, a user receives product suggestions based on their basket content and historical context. The feedback loop in this scenario is rather short; you can quickly determine if a recommended item was added to the basket and purchased.

You can use metrics like precision@K, recall@K, and F-score@K and monitor them over time:

Precision@K

> This measures what share of items within the top *K* positions are relevant.

Recall@K

> This measures the share of relevant items captured within the top *K* positions.

F-score@K

> This provides a balanced measure of Precision and Recall at *K*.

If you want to explore the nuances further, Evidently AI has an extensive explanation of these metrics. Overall, the higher the value of these metrics, the better the quality of the recommender.

However, we should not forget the purpose of the recommender system in the first place—generating monetary value. For the recommender to fulfill its purpose, users must have a *higher basket value* with the recommender than they would with random suggestions or no system at all. If the basket value isn’t increasing, then

high performance metrics are misleading. Keep in mind, however, that it often requires a longer feedback loop to gather enough data to confirm this financial impact.

In the case of such a recommender system, we can evaluate model quality offline during pre-deployment, monitor those same performance metrics in near-real time once the system is live, and track the business metrics daily.

Regression and Demand Forecasting

In regression tasks, such as predicting weekly sales by product category, calculating metrics like RMSE in real time is impossible because the ground truth (actual sales numbers) takes a full week to materialize.

However, there are two things we can monitor with a short feedback loop:

Business user feedback

> We can monitor for an unusual volume of manual corrections to model outputs.

Estimated model performance

> We can utilize methods like direct loss estimation (DLE) to approximate performance metrics before the ground truth is available.

The DLE method works by training a secondary *nanny model* to predict the absolute error of the primary *child model*. While both models use the same input features, the nanny model uses the absolute error of the monitored model as its target, This allows for performance estimation on new observations where the ground truth is not known. For a deeper technical breakdown, check out the NannyML documentation, which offers extensive resources on this approach.

The ultimate performance metric is again the business value. Evaluation should focus on the specific costs associated with over-prediction versus under-prediction and whether the model provides a measurable lift over simple baselines, such as a moving average.

Large Language Models

Monitoring LLMs is even harder because outputs are stochastic and context dependent. In their article “Unsolved Problems in MLOps,” Niall Murphy and Todd Underwood provide an example of how Azure monitors LLM performance:

> *Azure uses two ways of validating that new models in the Azure UI are working well: first, a set of LLMs judging the output of the new LLMs, and second, whether or not enough Microsoft employees have hit the equivalent of the “thumbs up” button when exposed to its recommendations.*[1]

Monitoring AI projects often comes down to monitoring custom business metrics that are tailored to a specific organization. This need for specialized key performance indicators (KPIs) is a primary reason why many companies develop bespoke monitoring tools.

Regardless of the use case, the goal is to detect *abrupt or gradual changes in model outputs*, which can signal that the world around the model has shifted. This is known as *concept drift.*

For example, imagine a model that predicts housing prices based on features like the age and size of the house, the neighborhood, the year when it was most recently renovated, and the number of bedrooms. Over time, the relationships among these variables and the final price may change. If the government introduces a new housing subsidy for families or if market preferences shift, a large home might suddenly sell for less than historically expected. At that point, because the underlying relationship between house size and price has changed, the model’s original assumptions are no longer valid. This shift, where the logic of the data itself evolves, is the essence of concept drift.

Concept Drift versus Data Drift

In contrast, *data drift* happens when the input data distribution shifts even though the underlying relationships remain the same. Continuing with the housing example, imagine that there’s a spike in new construction in a specific neighborhood, but without any changes to government policy or buyer preferences. The model might begin to underperform simply because it hasn’t encountered enough examples before. In this scenario, nothing fundamental has changed about how house features relate to price; only the composition of the data has shifted.

Significant data drift, even in features with high predictive power, may not hurt model quality, especially if the model generalizes well. For this reason, drift detection should never be viewed in isolation. It must be combined with model performance estimation and business metrics to determine whether the observed change truly matters.[2]

Fairness, Bias, and Regulatory Compliance

Another aspect of monitoring is *fairness and bias*, which has become a formal regulatory requirement under the EU Artificial Intelligence Act. Credit-scoring systems, employment systems, and other "high-risk" AI systems must demonstrate fairness, transparency, and risk management throughout their lifecycle, and organizations running such systems must engage in continuous monitoring after deploying them. In practice, this requires organizations to verify that their models perform equitably across different groups. For instance, in a loan approval scenario, applicants with similar financial profiles must receive comparable creditworthiness scores, regardless of demographic attributes like gender or ethnicity. These requirements are discussed in Chapter 10.

System and Infrastructure Health

Besides model- and business-specific aspects, it's also important to keep an eye on the *generic monitoring* that every software system needs. Even the best model is useless if the system around it is failing.

This includes tracking *system health*, *errors*, *latency*, and *infrastructure costs*. A data pipeline outage or slow feature store can degrade predictions more than any modeling issue. Spikes in serving latency can silently harm user experience, while unnoticed infrastructure scaling can quickly inflate cloud bills. In mature MLOps setups, these signals are integrated into the same observability layer as model and data metrics.

Monitoring, in the end, is about *closing feedback loops* across models, data, and infrastructure so that issues are detected early and can be acted on before they affect users or business outcomes.

To recap, a comprehensive monitoring strategy for ML applications covers four key pillars, each of which has multiple components:

Model performance metrics

Prediction quality

You perform real-time tracking of metrics like precision@K, recall@K, and F-score@K for recommenders, or RMSE, MAE, and MAPE for regression tasks.

Estimated performance

When ground truth arrives late (e.g., weekly or monthly), you use proxy estimates (when possible) to predict model error in near-real time.

Drift and stability

You watch for abrupt or gradual changes in model outputs or input distributions that may indicate concept drift or data drift.

Fairness and bias

You monitor whether the model performs similarly on data associated with different demographic groups (if such data is available).

Business metrics

Value alignment

You connect model output to revenue lift, conversion rates, or reduced waste.

Impact over accuracy

Even if your offline metrics look good, you must make sure your model is moving key business indicators in the right direction. If not, it's not a successful model.

Operational and human-in-the-loop signals

User feedback

You track user interactions and feedback (clicks, purchases, and thumbs up/down).

Manual overrides

You monitor business user overrides or corrections since a rise in manual edits often indicates declining trust or model relevance.

System and infrastructure health

Operational robustness

Monitoring system health, errors, latency, and infrastructure costs is part of every robust MLOps setup.

Cost management

A broken pipeline, a slow feature store, or escalating compute cost can degrade business outcomes as much as a poor model.

Next, we'll take a deep dive into Databricks Data Quality Monitoring to explore how Databricks simplifies these tasks by unifying data and AI observability in a single platform.

Introduction to Databricks Data Quality Monitoring

Databricks Data Quality Monitoring provides built-in capabilities to monitor both data and models in production. It enables you to track the statistical properties and quality of data across selected tables, so it helps you identify issues such as missing values or distribution shifts.

> **NOTE**
>
> You can monitor the input tables that are used to train ML models, as well as serving endpoints. You can do this by first processing inference tables that store model inputs and predictions and then setting up monitoring of the resulting aggregated tables.

Data Quality Monitoring supports two primary automated monitoring scenarios—time series and inference—as well as snapshot mode for custom analysis:

Time series scenario

> This is suitable for input tables that are used for model training or feature generation, in which data arrives periodically and includes a timestamp column.

Inference scenario

> This is designed for tables that store model inputs and predictions, which allow you to monitor both data quality and model performance over time. Note that you need the ground truth to compute the model performance metrics.

When monitoring is enabled on a table, Databricks automatically creates three core assets:

- A profile metrics table
- A drift metrics table
- A Databricks SQL dashboard for visualization

Figure 6-2 depicts the relationships among these assets.

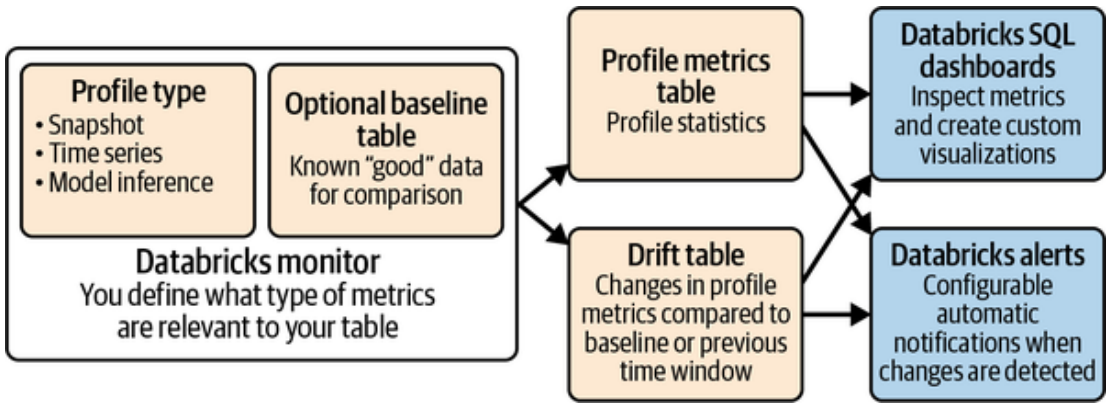


Figure 6-2. The three core monitoring assets, adapted from Databricks documentation

The *profile metrics table* contains summary statistics for each column and for each combination of the *time window*, *slice*, and *grouping columns*. These include measures such as counts, means, standard deviations, and value distributions. For

inference analyses, the table also includes *model accuracy metrics* like precision, recall, or RMSE, depending on the model type.

The *drift metrics table* tracks *changes in data distributions* over time. Rather than focusing on specific values, it measures how much a given metric changes compared to either the *previous time window* (via consecutive drift) or an optional *baseline table*, such as the training dataset, which is used as a fixed reference for comparison. Note that the schema of the baseline table must match the schema of the input table.

Consecutive drift is computed only when a preceding time window exists after aggregation according to the chosen granularity. For time series and inference scenarios, the following time windows are supported:

- 5 minutes
- 30 minutes
- 1 hour
- 1 day
- *N* weeks
- 1 month
- 1 year

If you need more control—for example, to monitor tables without a timestamp column or to analyze data on an ad hoc schedule—you can use a snapshot profile instead. *Snapshot profiles* process the entire table at each run and can optionally compare results to a baseline table. Even without a baseline, Databricks automatically compares metrics between snapshots, to help you detect changes or drift over time.

Across all monitoring modes, you can define *slices* to break down metrics by dimensions such as region, product category, or customer segment. This allows you to identify data quality issues or model drifts that appear only within specific subgroups, even when overall metrics seem stable.

Profile Metrics Table

Let's take a closer look at the profile metrics table structure to understand how we can use it for monitoring purposes.

The table organizes data using several key *grouping columns*:

Window and granularity

: This column defines the specific time interval (e.g., a one-hour window) using a `struct<start: timestamp, end: timestamp>` format.

Model ID (`model_id_col`*)*

: This column is specific to the inference scenario. If an input table contains predictions from multiple model versions (such as during A/B testing), this column allows Data Quality Monitoring to compute separate metrics for each version, rather than aggregating them.

Log type (`log_type`*)*

: This column indicates which table was used to calculate the metrics. When set to `BASELINE`, metrics are computed for the baseline (reference) table; when set to `INPUT`, metrics are computed for the currently monitored table.

Slicing (`slice_key` *and* `slice_value`*)*

: These columns capture the slicing expression and its value, thus enabling you to compare performance across different categories or segments.

Column metadata (`column_name` *and* `data_type`*)*

: These columns include the column used to calculate each metric and its corresponding Spark data type. The special value of `:table` is used for metrics (such as overall model accuracy) that apply to the entire table, rather than a single column.

Summary statistics are computed for every combination of these grouping columns. Table 6-1 shows the summary statistics present in the profile metrics table.

Table 6-1. Profile metrics table structure (adapted from Databricks documentation)

Column name	Data type	Description
`count`	`bigint`	Number of non-null values
`num_nulls`	`bigint`	Number of null values in the column
`avg`	`double`	Arithmetic mean of the column, ignoring nulls
`quantiles`	`array<double>`	Array of 1,000 quantiles
`distinct_count`	`bigint`	Approximate number of distinct values in the column
`min`	`double`	Minimum value in the column
`max`	`double`	Maximum value in the column
`stddev`	`double`	Standard deviation of the column
`num_zeros`	`bigint`	Number of zeros in the column
`num_nan`	`bigint`	Number of not-a-number (NaN) values in the column
`min_size`	`double`	Minimum size of arrays or structures in the column
`max_size`	`double`	Maximum size of arrays or structures in the column
`avg_size`	`double`	Average size of arrays or structures in the column

Column name	Data type	Description
`min_len`	`double`	Minimum length of string and binary values in the column
`max_len`	`double`	Maximum length of string and binary values in the column
`avg_len`	`double`	Average length of string and binary values in the column
`frequent_items`	`array<struct<item: string, count: bigint>>`	Top 100 most frequently occurring items
`non_null_columns`	`array<string>`	List of columns with at least one non-null value
`median`	`double`	Median value of the column
`percent_null`	`double`	Percent of null values in the column
`percent_zeros`	`double`	Percent of values that are zero in the column
`percent_distinct`	`double`	Percent of values that are distinct in the column

In the inference scenario, providing ground truth labels enables the automatic calculation of model performance metrics. Table 6-2 outlines the specific metrics computed for classification problems.

Table 6-2. Metrics computed for classification problems in the profile metrics table (adapted from Databricks documentation)

Column name	Type
accuracy_score	double
confusion_matrix	struct<prediction: string, label: string, count: bigint>
precision	struct<one_vs_all: map<string,double>, macro: double, weighted: double>
recall	struct<one_vs_all: map<string,double>, macro: double, weighted: double>
f1_score	struct<one_vs_all: map<string,double>, macro: double, weighted: double>

Table 6-3 shows the metrics computed for regression problems.

Table 6-3. Metrics computed for regression problems in the profile metrics table (adapted from Databricks documentation)

Column name	Type
mean_squared_error	double
root_mean_squared_error	double
mean_average_error	double
mean_absolute_percentage_error	double
r2_score	double

For classification problems, we can also monitor fairness and bias. First, we must create a boolean slice expression to separate data into two groups. The group where the expression evaluates to True is treated as the *protected group* (the group we are checking for bias against), and the group where the expression evaluates to False is called the *unprotected group*. For example, if the slice is defined by `slice_key = "age < 18"` and `slice_value = True`, we will be checking whether the algorithm is biased against individuals younger than 18 years old.

The monitor then automatically computes fairness metrics by comparing model performance between these groups. These fairness metrics include the following:

predictive_parity

: This compares model precision between groups.

predictive_equality

: This compares false positive rates.

equal_opportunity

: This compares recall.

statistical_parity

: This compares the proportions of positive predictions.

At this point, we've walked through the profile metrics table, which provides a snapshot of the statistical properties and quality of your data over time. These metrics are useful for understanding how your data behaves under normal conditions, establishing baselines for monitoring, and quickly detecting unexpected changes such as missing values and anomalies in key features.

Now, let's take a closer look at the *drift metrics table*, which builds on these profiles to quantify how data distributions change between time windows or against a defined baseline.

Drift Metrics Table

The drift metrics table uses the same grouping columns as the profile metrics table, except that it doesn't have the `log_type` column. Instead, it includes the following additional columns:

drift_type

This column indicates whether the drift is measured against a fixed `BASELINE` or the `CONSECUTIVE` (previous) time window.

window_cmp

This column specifies the comparison window used when the `drift_type` is `CONSECUTIVE`.

Table 6-4 lists all computed drift metrics and gives descriptions of them.

Table 6-4. Metrics computed for drift detection (adapted from Databricks documentation)

Column name	Data type	Description
`count_delta`	`double`	Difference in `count`
`avg_delta`	`double`	Difference in `avg`
`percent_null_delta`	`double`	Difference in `percent_null`
`percent_zeros_delta`	`double`	Difference in `percent_zeros`
`percent_distinct_delta`	`double`	Difference in `percent_distinct`
`non_null_columns_delta`	`struct<added: int, missing: int>`	Number of columns with any increase or decrease in non-null values
`chi_squared_test`	`struct<statistic: double, pvalue: double>`	Chi-square test for drift in distribution, which is calculated for *categorical columns* only
`ks_test`	`struct<statistic: double, pvalue: double>`	Kolmogorov-Smirnov test for drift in distribution, which is calculated for *numeric columns* only
`tv_distance`	`double`	Total variation distance, which is calculated for *categorical columns* only
`l_infinity_distance`	`double`	L-infinity distance, which is calculated for *categorical columns* only
`js_distance`	`double`	Jensen–Shannon distance, which is calculated for *categorical columns*

Column name	Data type	Description
		only
`wasserstein_distance`	`double`	Wasserstein distance metric, which is calculated for *numeric columns* only
`population_stability_index`	`double`	Population stability index metric, which is calculated for *numeric columns* only

Let's now go through all the metrics we've mentioned, discuss their meaning, and show you how to calculate them based on practical examples.

Metrics for Monitoring Data Drift

If you're familiar with statistics, you've likely come across methods for comparing two distributions: via hypothesis tests or distance metrics. You may have seen them in statistics courses, ML textbooks, or production monitoring tools used in MLOps.

These same techniques are widely used for *data drift detection*, to help identify shifts between training data and incoming data that could impact model performance. While different resources mention some of these metrics, it's surprisingly hard to find a single resource that brings them all together.

Chi-squared Test

The *chi-squared test* is frequently used to compare the distributions of categorical variables, and it has widespread application in various domains. It's often used in A/B testing (to verify whether users who received a different version of the model react in a different way), and drift detection (to verify that a categorical variable shows the same category frequencies over time).

To illustrate how to calculate the chi-squared statistic and probability value (*p*-value), let's look at the categorical `type_of_meal` variable across the last two monitoring windows.

We start by defining our hypotheses:

Null hypothesis (H_0)

> Both datasets come from the same underlying distribution.

Alternative hypothesis (H_1)

> The datasets come from different distributions, which suggests that the observed difference is caused by data drift.

Next, we use the data to test the null hypothesis. If we can reject it, we conclude that data drift has occurred. If not, we conclude that the observed differences could be explained by random variation and that there is no evidence of data drift.

Based on the observed values (*O*) recorded in the profile metrics table, Table 6-5 provides an overview of the category counts for each window.

Table 6-5. Observed meal plan frequencies by window

	Window 1	Window 2	Total
Meal Plan 1	224	629	*853*
Not Selected	42	141	*183*
Meal Plan 2	23	65	*88*
Total	*289*	*835*	*1124*

Now, let’s calculate the expected values (*E*), as shown in Table 6-6.

Table 6-6. Expected meal plan frequencies (E)

	Window 1	Window 2
Meal Plan 1	289 × 853 / 1,124 = 219,3	835 × 853 / 1,124 = 633,7
Not Selected	289 × 183 / 1,124 = 47,1	835 × 183 / 1,124 = 136
Meal Plan 2	289 × 88 / 1,124 = 22,6	835 × 88 / 1,124 = 65,3

The observed chi-squared statistics are calculated as follows:

$$\chi^2_{obs} = \sum \frac{(E-O)^2}{E} = 0,873$$

This matches the exact value we observed in the drift table for the last period.

Now, let's now calculate the *p*-value, which is a measure of whether the results of an experiment mean something or are just chance occurrences. It is defined as follows:

$$p = 1 - F_{\chi^2}(\chi^2_{obs}; df)$$

where $F_{\chi^2}(\chi^2_{obs}; df)$ is a cumulative distribution function of the chi-squared distribution.

Also, $df = (r-1)(c-1) = 2$, as we have three rows (r) and two columns (c).

Finally, $p = 1 - F_{\chi^2}(0,873; 2) \approx 0,646$.

The *significance level (α)* defines how much evidence we need to reject the null hypothesis. If the *p-value* ≤ *α*, we reject H_0, if the *p-value* > *α*, we fail to reject H_0.

In our case, the *p*-value is 0,65, which is much greater than a typical α of 0,05**.** This means the observed difference could easily be due to random chance, so we don't reject the null hypothesis.

Overall, a chi-squared test is a common choice among ML practitioners as it is easy to interpret. It's important to take into consideration that the test is not recommended for categorical features with many low-frequency categories unless the sample size is large.

Kolmogorov-Smirnov Test

The *Kolmogorov-Smirnov test* is a widely used test for investigating the significance of the difference between two distributions, but it's applied to numeric variables. Like the chi-squared test, it evaluates whether two datasets come from the same underlying distribution (H_0) or if a significant shift, which is indicative of data drift, has occurred (H_1).

Let's create a fictional example for the `number_of_week_nights` variable to help us understand how the test is calculated. The first window contains $n = 4$ entries, and the second window contains $m = 6$ entries:

- Window 1: 1, 5, 7, 3
- Window 2: 2, 3, 3, 5, 8, 1

Now we can calculate the cumulative relative frequency (CRF) for both windows and the absolute difference between those values for both windows (see Table 6-7).

Table 6-7. Calculation of the Kolmogorov-Smirnov (K-S) D-statistic

	Window 1 freq	Window 2 freq	Window 1 CRF	Window 2 CRF
1	1	1	0,25	0,167
2		1	0,25	0,333
3	1	2	0,5	0,667
5	1	1	0,75	0,833
7	1		1	0,833
8		1	1	1

The maximum difference is 0,167, which is the value of Kolmogorov-Smirnov statistic $D_{n,m}$.

Rejection of the null hypothesis occurs at a significance level of (α) if $D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{n \cdot m}}$ (the critical level of D-statistics), where $c(\alpha) = \sqrt{-ln(\frac{\alpha}{2}) \cdot \frac{1}{2}}$.

With the significance level 0,05, the critical level of D-statistics is 0,88, $D_{n,m} < D_{crit}$, which means the null hypothesis can't be rejected and there is no data drift.

For simplicity, Databricks also calculates *p*-value for the Kolmogorov-Smirnov test, which allows it to easily evaluate whether there is data drift.

It's important to note that the Kolmogorov-Smirnov test is sensitive to sample size, so it can lead to false alerts in case of small datasets.

Total Variation Distance

The total variation distance (TVD) is another measure of how two different categorical distributions diverge, and it's frequently used in data drift monitoring. Unlike the chi-squared test, which relies on statistical significance and *p*-values, the TVD is a *direct numerical measure* of how much two distributions diverge independent of sample size.

Let's come back to the same example we used to explain the chi-squared test and calculate the TVD. The observed values for the *type_of_meal* variable for each of the windows are presented in Table 6-8.

Table 6-8. Observed frequencies for the `type_of_meal` variable across two time windows

	Window 1	Window 2	Total
Meal Plan 1	224	629	*853*
Not Selected	42	141	*183*
Meal Plan 2	23	65	*88*
Total	*289*	*835*	*1,124*

Now, let’s calculate the relative frequencies for each window and the difference between these relative frequencies for each category in Table 6-9.

Table 6-9. Calculation of absolute differences in relative frequencies between windows

	***P*(Window 1)**	***Q*(Window 2)**	**| *P* – *Q* |**
Meal Plan 1	224 / 289 = 0,775	629 / 835 = 0,753	*0,022*
Not Selected	42 / 289 = 0,145	141 / 835 = 0,169	*0,024*
Meal Plan 2	23 / 289 = 0,080	65 / 835 = 0,078	*0,002*
			Total: 0,048

We use this formula to calculate the TVD:

$$TVD(P,Q) = \frac{1}{2}\sum_i |P_i - Q_i| = \frac{1}{2} \cdot 0,048 = 0,024$$

This result means that the categorical distribution has changed by only 2,4% between the two windows. Since it’s a small change, we can say the two distributions are very similar, so there is *no meaningful evidence of drift*.

The TVD is easy to compute, and it ranges from 0 (identical distributions) to 1 (completely different distributions). It also does not depend on sample size or statistical thresholds, so it’s intuitive for continuous drift monitoring.

L-Infinity Distance

The *L-infinity distance* (also known as the *maximum absolute difference*) measures the largest change between category frequencies across two distributions.

It is defined as follows:

$$L_\infty(P,Q) = \max_i |P_i - Q_i|$$

where P_i and Q_i are the relative frequencies of category *i* in each window.

Using the same example as in our TVD calculation and our previously computed probabilities, the largest absolute difference between categories is 0,024 (for category “Not Selected”).

This means the maximum shift in any single category between the two time windows is about 2,4%, which confirms that no individual category changed substantially. The distributions remain very similar, and there is no evidence of data drift.

L-infinity distance is easy to compute, ranges from 0 to 1, and captures the biggest individual category change. It's often used alongside TVD, where TVD shows the overall change and L-infinity highlights the largest category-level difference.

Jensen-Shannon Distance

The Jensen-Shannon distance (JSD) measures how different two probability distributions are from each other. It is a bounded, symmetric, and smoothed version of the Kullback-Leibler divergence D_{kl}, which makes it well-suited for *data drift detection*.

It's defined as follows:

$$JSD(P \parallel Q) = \sqrt{\frac{1}{2} D_{kl}(P||M) + \frac{1}{2} D_{kl}(Q||M)}$$

where $M = \frac{1}{2}(P + Q)$, and $D_{kl} = \sum_i P_i \log_2 \frac{P_i}{M_i}$.

Using the same previously computed relative frequencies, let's compute *M* and *KL* divergences in Table 6-10.

Table 6-10. Calculation of KL divergence components for Jensen-Shannon distance

	P (Window 1)	Q (Window 2)	M	$D_{kl}(P\|\|M$
Meal Plan 1	0,775	0,753	0,764	0,013
Not Selected	0,145	0,169	0,157	−0,011
Meal Plan 2	0,080	0,078	0,079	0,001
Sum				0,003

We can now compute the Jensen–Shannon distance:

$$JSD(P \parallel Q) = \sqrt{\frac{1}{2} \cdot (0,003 + 0,001)} = 0,045$$

The value of the JSD runs from 0 for identical distributions to 1 for completely different distributions. In our example, the value of the metric is very small (close to 0), which confirms that the two distributions are almost identical.

While JSD is highly intuitive and stable, it is more computationally expensive than other categorical metrics provided by Databricks.

Wasserstein Distance

The *Wasserstein distance* (also known as the *earth mover's distance*) measures how much "work" is needed to transform one numeric distribution into another. In one dimension, it equals the area between the two empirical cumulative distribution functions:

$$W_1(P, Q) = \int_{-\infty}^{\infty} |F_p(x) - F_q(x)| dx$$

It's expressed in the same units as the variable, so it's easy to interpret.

Let's use the same example we used to explain the Kolmogorov-Smirnov test. The observations for two windows were as follows:

- Window 1: 1, 5, 7, 3
- Window 2: 2, 3, 3, 5, 8, 1

Using the `number_of_week_nights` variable, let's recalculate the cumulative relative frequencies and the step size Δx for both windows in Table 6-11.

Table 6-11. Calculation of the Wasserstein distance area components

x	Window 1 freq	Window 2 freq	Window 1 CRF	Window 2 CRF
1	1	1	0,25	0,167
2		1	0,25	0,333
3	1	2	0,5	0,667
5	1	1	0,75	0,833
7	1		1	0,833
8		1	1	1

The Wasserstein distance is the sum of the areas:

$$W_1(P, Q) = 0,083 \cdot 1 + 0,083 \cdot 1 + 0,167 \cdot 2 + 0,083 \cdot 2 + 0,167 \cdot 1 \approx 0,83$$

It's equal to 0,83 nights, which means that on average, values in one window must shift by ≈0,83 nights to match the other window's distribution. To understand whether this is a lot, we need to compare the shift to the feature's range and standard deviation. Here, the range is 1–8 nights, so 0,83 is ~12% of the range, which can be considered a moderate shift.

One of the advantages of the Wasserstein distance is that it captures *how far* distributions move, not just whether they differ.

Population Stability Index

The Population Stability Index (PSI) measures how much a numeric (or binned) feature's distribution has shifted over time. It compares distributions by aggregating changes across bins. It's easy to interpret and widely used in model monitoring. The PSI is calculated as follows:

$$PSI(P, Q) = \sum_i (Q_i - P_i) \ln(\frac{Q_i}{P_i})$$

Let's take the same example again, with the `number_of_week_nights` variable:

- Window 1: 1, 5, 7, 3
- Window 2: 2, 3, 3, 5, 8, 1

Now, we can create bins and compute probabilities in Table 6-12.

Table 6-12. Calculation of the PSI across binned intervals

Interval	P_i (Window 1)	Q_i (Window 2)	$(Q_i - P_i) \ln\left(\frac{Q_i}{P_i}\right)$
(–∞, 2.0]	0,25	0,3333	0,027
(2.0, 4.0]	0,25	0,3333	0,027
(4.0, 6.0]	0,25	0,1667	0,029
(6.0, ∞)	0,25	0,1667	0,029

Finally, we aggregate these individual bin results to calculate the PSI:

$$PSI(P, Q) = 0,027 + 0,027 + 0,029 + 0,029 = 0,112$$

A PSI of 0,112 can be considered a moderate shift in distribution. The following thresholds for PSI (which originated from credit risk monitoring practices) are widely used as industry guidelines:

- PSI < 0,1 indicates a negligible shift.
- 0,1 < = PSI < 0,25 indicates a moderate shift.
- PSI > = 0,25 indicates a significant shift.

In general, PSI does not have a theoretical cutoff, and its interpretation depends on the variable's natural variability, sample size, and application context. This makes it more difficult to apply PSI consistently in practice, compared to other drift metrics. It is also important to note that PSI is sensitive to the choice and number of bins, which can significantly affect the result.

Now that we have explored the structure of profile and drift metrics tables and how these values are calculated, let's move from theory to implementation. We are

going to merge the data from the inference table with the ground truth data and set a monitor on the resulting dataset.

From Inference Tables to Data Quality Monitoring

In Chapter 4, we enabled an inference table for our custom Model Serving endpoint. This table automatically logs inference requests and responses each time the endpoint is called.

To demonstrate data drift, let's invoke the endpoint using 4,000 records from the `hotel_booking` table: 2,000 selected randomly from the entire dataset and another 2,000 specifically from the corporate market segment. Note that it can take approximately 20 minutes for the inference table to accumulate enough data for analysis:

```
import time

import pandas as pd
import requests
from databricks.sdk import WorkspaceClient
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig


cfg = ProjectConfig.from_yaml(config_path="../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema

spark = SparkSession.builder.getOrCreate()
input_data = spark.table(f"{catalog}.{schema}.hotel_booking").toPandas()

# Define required columns for endpoint
required_columns = [
    "Booking_ID",
    "number_of_adults",
    "number_of_children",
    "number_of_weekend_nights",
    "number_of_week_nights",
    "car_parking_space",
    "special_requests",
    "lead_time",
    "type_of_meal",
    "room_type",
    "arrival_month",
    "market_segment_type",
```

```
]

sample_per_phase = 2000

sampled_all = input_data[required_columns].sample(
   n=sample_per_phase, replace=True
)

corporate = input_data[input_data["market_segment_type"] == "Corporate"]
sampled_corporate = corporate[required_columns].sample(
    n=sample_per_phase, replace=True)

sampled_records = pd.concat([sampled_all, sampled_corporate], ignore_index=True)
```

The script for this task, *chapter6_1.call_endpoint.py*, is located in the *notebooks* folder.

We're not changing the approach from Chapter 4, except that we're now including a `client_request_id` field that contains the `Booking_ID`. This addition will be important later, when we join the inference table with the ground truth data.

Here's an example of what the request payload looks like:

```
{
   "client_request_id": "INN36285",
   "dataframe_records": [
       {
           "number_of_adults": 2,
           "number_of_children": 0,
           "number_of_weekend_nights": 0,
           "number_of_week_nights": 1,
           "car_parking_space": 0,
           "special_requests": 0,
           "lead_time": 103,
           "type_of_meal": "Not Selected",
           "room_type": "Room_Type 1",
           "arrival_month": 8,
           "market_segment_type": "Online",
       }
   ]
}
```

Now, let's investigate the structure of an inference table and provide example values.

Inference Table Structure

Table 6-13 shows the structure of the inference table. This table, in its raw form, cannot be used directly for monitoring. All model features are stored in the `request` column as JSON-like strings, rather than as separate columns. Similarly, the response column contains the model predictions in a JSON-like format.

To enable effective monitoring, we first need to process and transform this table into a structured format in which each feature and prediction is represented as an individual column.

Table 6-13. Inference table structure

Column	Data type	Comment	Example
`databricks_request_id`	`string`	This is the request ID.	`c0e67132-e924-4fb4-b83d-ab5603758baf`
`request_date`	`date`	This is the UTC date on which the request was received.	`2025-10-25`
`client_request_id`	`string`	This is the user-provided request identifier.	`INN34434`
`request_time`	`timestamp`	This is the timestamp when the request is received.	`2025-10-25T21:44:34.637+00:00`
`status_code`	`int`	This is the HTTP status code.	`200`
`sampling_fraction`	`double`	This is the sampling fraction used in the event that the request is down-sampled.	`1.0`
`execution_duration_ms`	`bigint`	This is the time it took for the model to generate predictions. It does not include overhead network latencies.	`5`

Column	Data type	Comment	Example
request	string	This is the raw request JSON body that was sent to the Model Serving endpoint.	{"client_request_id": "INN34434", "dataframe_records": [{ "number_of_adults": 2, "number_of_children": 0, "number_of_weekend_nights": 0, "number_of_week_nights": 2, "car_parking_space": 0, "special_requests": 0, "lead_time": 317, "type_of_meal": "Meal Plan 1", "room_type": "Room_Type 1", "arrival_month": 7, "market_segment_type": "Offline"}]}
response	string	This is the raw response JSON body that was returned by the Model Serving endpoint.	{ "predictions": { "Total price per night": [77.83] } }
logging_error_codes:	array	These are errors that occurred when the data could not be logged (for example, when request or response exceeded the maximum allowed size).	[]

Column	Data type	Comment	Example
`served_entity_id`	`string`	This is the unique ID of the served entity.	`40e11be0ddd340eca7e975b13005edc1`
`requester`	`string`	This is the ID of the user or service principal whose permissions are used to call the serving endpoint. It will be NULL for route-optimized custom model endpoints.	`maria@cauchy.io`

Now that we're familiar with the structure of the inference table, we can move on to preparing and processing it so we can use it to enable monitoring.

Processing an Inference Table

To process the inference table, we need to specify the structure of the request and the response columns so we can parse it later:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import (
   ArrayType,
   DoubleType,
   IntegerType,
   StringType,
   StructField,
   StructType,
)

spark = SparkSession.builder.getOrCreate()

request_schema = StructType([
   StructField("dataframe_records", ArrayType(StructType([
```

```
        StructField("number_of_adults", IntegerType(), True),
        StructField("number_of_children", IntegerType(), True),
        StructField("number_of_weekend_nights", IntegerType(), True),
        StructField("number_of_week_nights", IntegerType(), True),
        StructField("car_parking_space", IntegerType(), True),
        StructField("special_requests", IntegerType(), True),
        StructField("lead_time", IntegerType(), True),
        StructField("type_of_meal", StringType(), True),
        StructField("room_type", StringType(), True),
        StructField("arrival_month", IntegerType(), True),
        StructField("market_segment_type", StringType(), True),
    ])), True)
])

response_schema = StructType([
    StructField("predictions", StructType([
        StructField("Total price per night",
                    ArrayType(DoubleType()), True)
    ]), True)
])
```

Let's apply the defined schemas and use the `explode` function:

```
from hotel_booking.config import ProjectConfig

cfg = ProjectConfig.from_yaml(
    config_path="../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema
inf_table = spark.table(
    f"{catalog}.{schema}.hotel_booking_monitoring_payload")

inf_table_parsed = (
    inf_table
    .withColumn("p_request",
                F.from_json(F.col("request"),
                            request_schema))
    .withColumn("p_response",
                F.from_json(F.col("response"),
                            response_schema))
    .withColumn("prediction",
                F.col("p_response.predictions.`Total price per night`")[0])
    .withColumn("record",
                F.explode(F.col("p_request.dataframe_records")))
)
```

Next, we select the required columns for the monitoring table and join it with the `hotel_booking` table that contains the ground truth (the actual price). Notice that the final dataframe now contains the request time, the `databricks_request_id` as

an identifier, the execution duration, all the model features, the prediction, the model name, and the label:

```
df_final = inf_table_parsed.select(
   "request_time",
   "databricks_request_id",
   "execution_duration_ms",
   F.col("client_request_id").alias("Booking_ID"),
   F.col("record.number_of_adults").alias("number_of_adults"),
   F.col("record.number_of_children").alias("number_of_children"),
   F.col("record.number_of_weekend_nights").alias("number_of_weekend_nights"),
   F.col("record.number_of_week_nights").alias("number_of_week_nights"),
   F.col("record.car_parking_space").alias("car_parking_space"),
   F.col("record.special_requests").alias("special_requests"),
   F.col("record.lead_time").alias("lead_time"),
   F.col("record.type_of_meal").alias("type_of_meal"),
   F.col("record.room_type").alias("room_type"),
   F.col("record.arrival_month").alias("arrival_month"),
   F.col("record.market_segment_type").alias("market_segment_type"),
   "prediction",
   F.lit("hotel-booking-pyfunc").alias("model_name")
)

ground_truth_set = spark.table(f"{catalog}.{schema}.hotel_booking")

df_final_with_status = df_final.join(
   ground_truth_set.select("Booking_ID", "average_price"),
   on="Booking_ID", how="left")
```

Then, we write the results to the Delta table and enable ChangeDataFeed, which is required for incremental updates of the monitor:

```
monitoring_table = f"{catalog}.{schema}.model_monitoring"

df_final_with_status.write.format(
   "delta").mode("append").saveAsTable(monitoring_table)

# Important to update monitoring
spark.sql(f"""ALTER TABLE {monitoring_table}
          SET TBLPROPERTIES (delta.enableChangeDataFeed = true);""")
```

Creating a Data Quality Monitor

Finally, let’s create a data quality monitor:

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.dataquality import (
   AggregationGranularity,
```

```
    DataProfilingConfig,
    InferenceLogConfig,
    InferenceProblemType,
    Monitor,
)

w = WorkspaceClient()

output_schema = w.schemas.get(full_name=f"{catalog}.{schema}")
monitored_table = w.tables.get(full_name=monitoring_table)

w.data_quality.create_monitor(
    monitor=Monitor(
        object_type="table",
        object_id=monitored_table.table_id,
        data_profiling_config=DataProfilingConfig(
            output_schema_id=output_schema.schema_id,
            assets_dir=(
                f"/Workspace/Shared/DQM/{monitoring_table}"
            ),
            inference_log=InferenceLogConfig(
                problem_type=(
                    InferenceProblemType.INFERENCE_PROBLEM_TYPE_REGRESSION
                ),
                prediction_column="prediction",
                timestamp_column="request_time",
                granularities=[
                    AggregationGranularity.AGGREGATION_GRANULARITY_5_MINUTES
                ],
                model_id_column="model_name",
                label_column="average_price",
            ),
        ),
    ),
)
```

We specified the inference log problem type as *regression*, and we provided the prediction column, timestamp column, granularity, model ID column, and label column.

> **NOTE**
>
> In this example, we selected a *five-minute granularity* purely for demonstration purposes. In practice, however, you should choose this parameter carefully based on the specific characteristics of your problem and the expected data dynamics.

The *granularity* defines the time window over which metrics are aggregated and compared for drift detection. If the granularity is set too small, each window may contain few records, and that can cause excessive noise and potentially lead to false drift detections. On the other hand, if the granularity is too large, genuine changes in data distribution or model performance may be smoothed out, and that can make it difficult for you to detect meaningful drift in time. A good approach is to align the granularity with the natural cadence of your data and adjust it empirically, based on the volume and variability of incoming data.

Once you've initiated the monitor creation process, it may take some time before the profile metrics and drift metrics tables are fully populated. This is because Databricks triggers a Lakeflow Spark Declarative Pipeline (SDP) in the background to handle the processing. The duration depends on the size and complexity of the underlying data, as these factors determine the compute time required for the metrics.

At the end of the exercise, under the `hotel_booking` catalog, we'll have the following tables related to monitoring:

```
hotel_booking
└── Tables
    ├── hotel_booking_monitoring_payload
    ├── model_monitoring
    ├── model_monitoring_drift_metrics
    └── model_monitoring_profile_metrics
```

Using the Dashboard

The dashboard is generated immediately and becomes available in the assets directory. The default dashboard contains many visual blocks and can initially feel overwhelming. Figure 6-3 depicts a subset of such a dashboard—notice how the performance drops once the second data subset is introduced.

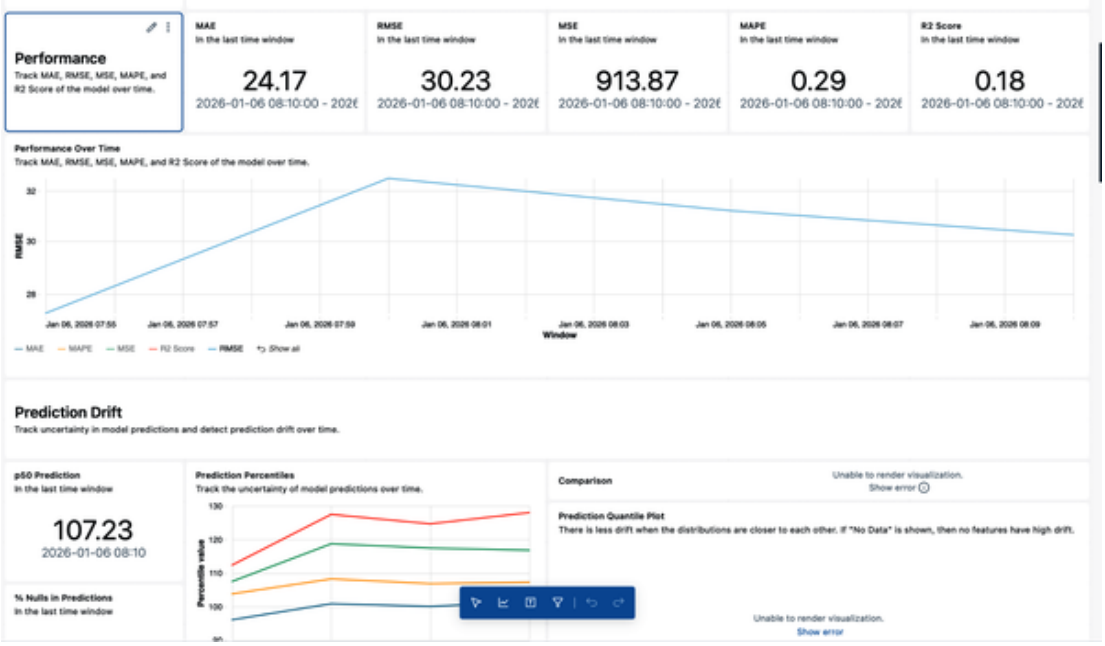


Figure 6-3. A subset of the monitoring dashboard showing performance and prediction drift

Feature drift information is also available in the dashboard. Figure 6-4 demonstrates what categorical features exhibited a drift during the last time window, based on the calculated metrics.

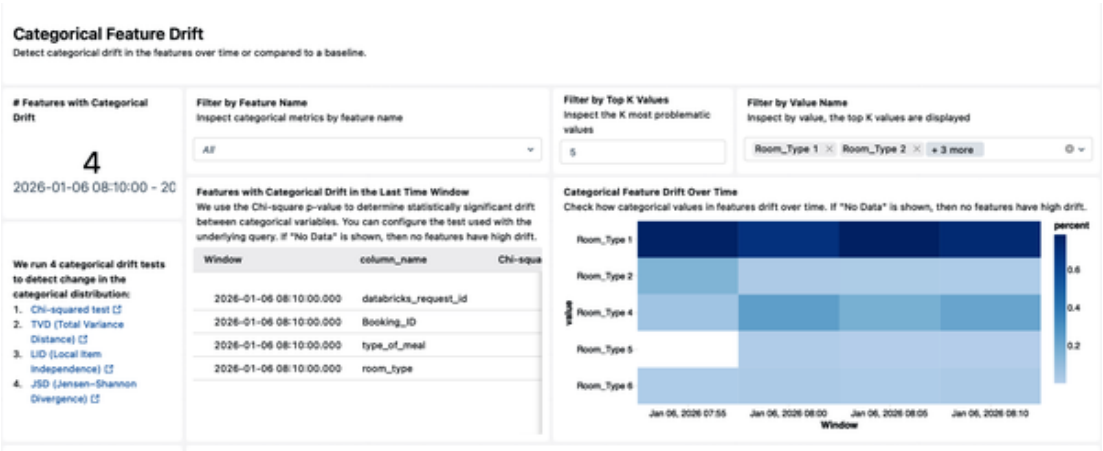


Figure 6-4. Categorical feature drift

You can customize the dashboard to fit your preferences directly through the Databricks UI. Under the hood, Databricks dashboards are defined as JSON files that describe the layout and the underlying SQL queries. It's a good practice to keep these JSON definitions under version control, so you can track changes and collaborate effectively. You can then deploy and manage dashboards programmatically with DABs, to ensure consistency and reproducibility among environments.

Since we expect the constant payload to be generated in the inference table, we need to periodically update the monitoring table and the monitor. Let's discuss the monitor refresh process.

Refreshing the Monitor

Refreshing the monitor is rather simple. We only have to run the `create_refresh()` command, which will trigger the underlying SDP run again:

```
from databricks.sdk.service.dataquality import (
   Refresh,
   RefreshTrigger,
)

w.data_quality.create_refresh(
   object_type="table",
   object_id=monitored_table.table_id,
   refresh=Refresh(
       object_type="table",
       object_id=monitored_table.table_id,
       trigger=RefreshTrigger.MONITOR_REFRESH_TRIGGER_MANUAL,
   ),
)
```

In Chapter 5, we looked at the definition of the monitoring Lakeflow job. Compared to the model training and deployment pipeline, it's rather simple and only includes one task: running `refresh_monitor.py`. We have packaged all the previous logic into this script, which utilizes the `MonitoringManager` class from the `hotel_booking.monitoring` module:

```
from loguru import logger
from pyspark.sql import SparkSession

from hotel_booking.config import ProjectConfig
from hotel_booking.monitoring import MonitoringManager
from hotel_booking.utils.common import create_parser

args = create_parser()

cfg = ProjectConfig.from_yaml(
   config_path=f"{args.root_path}/files/project_config.yml",
   env=args.env
)

spark = SparkSession.builder.getOrCreate()
```

```
logger.info("Initializing MonitoringManager")
manager = MonitoringManager(spark=spark, config=cfg)

logger.info("Starting monitoring table update")
num_records = manager.update_monitoring_table()

if num_records > 0:
   logger.info(f"Successfully processed {num_records} records")
   manager.refresh_monitor()
   logger.info("Monitor refresh completed")
else:
   logger.info("No new records to process")
```

The task has the following steps:

1. Check whether new data is populated in the inference table.
2. If new data is available, parse it, join it with the ground truth, and update the monitoring table.
3. Once the monitoring table is updated, refresh the monitor. This action will update the profile and drift metrics tables, plus the corresponding dashboard.

The ML pipeline we defined earlier is set to run weekly, while the monitoring pipeline runs daily. We discussed the cadence of model retraining in Chapter 5.

The frequency we use is illustrative. Your monitoring frequency should align with how quickly you need to detect and respond to issues. Daily monitoring works well for batch inference systems where predictions are consumed in aggregate. But if your model serves real-time requests and degradation directly impacts users, you may want to monitor hourly or even continuously. You should also consider the cost implications: higher monitoring frequency means higher costs. The goal is to find the sweet spot that keeps you informed of the model quality without needlessly burning through your budget.

Having a monitoring dashboard and keeping the underlying tables updated is a good start, but it’s not enough on its own. No one is going to sit there staring at metrics, waiting for something to break. When things go wrong, you want to find out immediately. Setting up alerting is one way of dealing with it.

Alerting

There are different ways to set up alerting, and the most convenient one is to use DABs (which we covered in Chapter 5). The alert deployment is specified in the *resources/alert.yml* file.

To create an alert, we first need to create a SQL warehouse, which we will use to run the specified query according to the predefined schedule. In our example, we choose the smallest serverless SQL warehouse with auto termination of five minutes to reduce the costs. At the moment of writing this book, it's not possible to specify a usage policy for a SQL warehouse, so we've provided tags for cost tracking purposes:

```
resources:
 sql_warehouses:
    hotel_booking_sql_warehouse:
     name: hotel_booking_sql_warehouse
     cluster_size: 2X-Small
     enable_serverless_compute: true
     max_num_clusters: 1
     auto_stop_mins: 5
     warehouse_type: PRO
     tags:
       custom_tags:
       - key: "project"
         value: "hotel_booking"
```

Now, let's define an alert. This one triggers when the percentage of records with a mean absolute error above 30 in the `model_monitoring_profile_metrics` table exceeds 45%. Note that we're pointing at the table in the production catalog. The query runs daily at 10:00 Amsterdam time, using the warehouse we configured earlier, and when triggered, it notifies users via email or a configured notification destination:

```
alerts:
  my_alert:
    display_name: hotel-booking-mae-alert
    evaluation:
      notification:
        subscriptions:
        - user_email: "maria@cauchy.io"
        # - destination_id: "12345"
      comparison_operator: GREATER_THAN
      source:
```

```
    name: 'percentage_higher_than_30'
  threshold:
    value:
      double_value: 45
query_text: SELECT (COUNT(CASE WHEN mean_absolute_error > 30 THEN 1 END)
* 100.0 / COUNT(CASE WHEN mean_absolute_error IS NOT NULL AND NOT
  isnan(mean_absolute_error) THEN 1 END)) AS percentage_higher_than_30
  FROM mlops_prd.hotel_booking.model_monitoring_profile_metrics
schedule:
  quartz_cron_schedule: "0 0 10 * * ?"
  timezone_id: Europe/Amsterdam
  pause_status: ${var.schedule_pause_status}
warehouse_id: ${resources.sql_warehouses.hotel_booking_sql_warehouse.id}
```

Note that the alert won't run in acceptance because we set `schedule_pause_status` to `PAUSED` for that target, and it won't run in development because that target has development mode enabled. We covered this when we went through the *databricks.yml* file definition in Chapter 5.

It's possible to configure different notification destinations. Figure 6-5 shows how workspace admins can configure notification destinations. Some destinations like Microsoft Teams, PagerDuty, and Slack are supported out-of-the-box, but a generic webhook option is also supported.

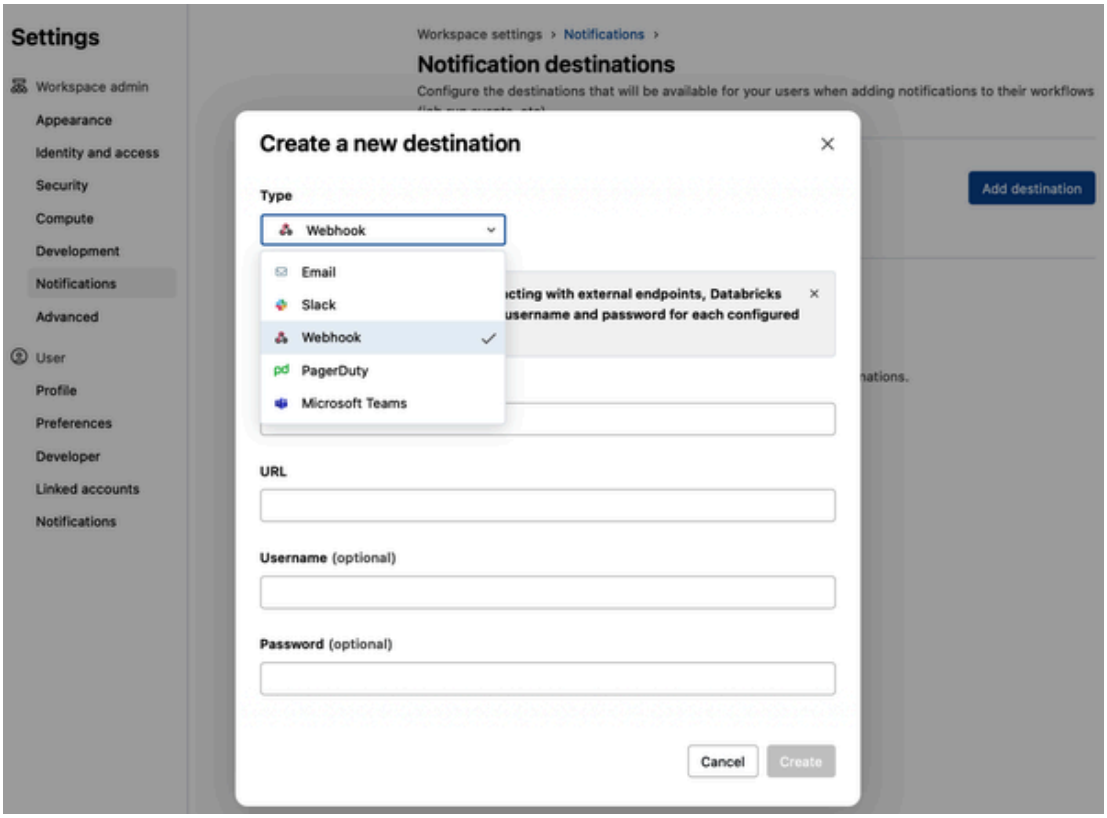


Figure 6-5. Configuring notification destinations in Databricks workspace

So far, we've talked about monitoring that's related to model features and model performance, but as we discussed at the beginning of the chapter, there are more things to monitor. We'll come back to monitoring in Chapter 9, where we'll discuss the cost dimension. Cost matters for any production system, but it is especially important for LLMs; token-based pricing means spend can grow quickly and often invisibly, which makes it one of the first operational signals you'll want to track. Because we apply two different usage policies across the book, by the end of the MLOps and LLMOps chapters, you will be able to see cost segregated between them.

Conclusion

In this chapter, we explored the many facets of monitoring ML systems in production. If there's one takeaway from this chapter, it's that monitoring is hard. Deciding what to measure, setting thresholds that catch real problems, routing alerts to the right people, and actually acting on what you find—all of it takes iteration. You won't get it right on the first try, and that's fine. The important thing is having the infrastructure in place so you can observe, learn, and adjust.

A model in production without monitoring is a liability. You may get lucky, and it may keep performing well. Or it may silently degrade for weeks while your users make decisions based on increasingly unreliable predictions. Monitoring doesn't guarantee your model will work, but at least you'll know when it isn't working.

This chapter closes the MLOps part of the book. The next three chapters turn to LLM-based systems, and you'll find that the operational concerns don't disappear so much as change shape. Monitoring is one of those threads: it returns for agents in Chapter 9, where nondeterministic, open-ended output makes "Is it still working?" a harder question to answer for LLMs than it was for a model with a fixed output schema.

---

1 Read the full article .

2 Check out Santiago Viquez's article "Don't Let Yourself Be Fooled by Data Drift". It discusses this problem in detail.

Chapter 7. Foundation Models and Context Engineering

The number of Generative AI (GenAI) use cases has grown rapidly over the past few years, moving well beyond experimentation into applications that deliver tangible business value.

For example, Airbnb completed a large-scale LLM-driven code migration to update nearly 3,500 React component test files. It completed in just six weeks a project that had initially been estimated to require 1.5 years of manual engineering, by combining frontier models with robust automation.

Similarly, DoorDash built AutoEval, an LLM-powered, human-in-the-loop system for evaluating the quality of search result pages. Instead of relying on slow and inconsistent human labeling, the system increased evaluation speed by 98% and expanded capacity by a factor of nine. Beyond scalability, it improved alignment with expert judgment and enabled continuous quality monitoring, allowing human experts to focus on higher-value tasks such as defining guidelines, handling edge cases, and improving models.

These examples illustrate a broader shift: organizations are no longer asking *if* they should use LLMs but *how* to operationalize them effectively. From my consulting experience, this is where the real challenge begins.

Introduction to LLMOps

Many teams are eager to build “agents” and experiment with increasingly complex LLM-powered systems, but far fewer have the infrastructure, processes, and practices required to run them reliably in production. The result is a growing gap between what is possible in demos and what can be sustained in real-world systems. This and the next two chapters focus on

closing that gap. Together, they cover the full deployment lifecycle of LLM-powered systems much like we covered the deployment lifecycle of traditional ML systems in Chapters 2 through 6.

Rather than optimizing for the "best" possible model or architecture, the emphasis here is on *operational excellence*: how to build systems that are observable, maintainable, and scalable in practice. Topics such as preprocessing, chunking strategies, embedding selection, and architecture design are important, and we will reference them where relevant, but they are not the primary focus.

This field goes by several names, including LLMOps, AgentOps, and GenAIOps. We use LLMOps deliberately. Agents are our main examples, but they are one type of architecture among several, and many nonagentic LLM systems raise the same operational questions. As we discussed in Chapter 1, what sets these systems apart from traditional ML is their nondeterminism and their reliance on pre-trained foundation models. Naming the discipline after the model keeps it general and preserves the line from MLOps to LLMOps.

Before we go further, however, it is important to clarify a term that will appear frequently throughout this chapter: *agent*.

What Is an Agent?

The term *agent* is used extensively across the AI landscape, often with slightly different meanings, which can easily lead to confusion if left undefined.

In this book, we adopt a pragmatic definition inspired by the OpenAI Agent SDK:[1]

> *An agent is a large language model (LLM) configured with instructions, a model, and access to tools.*

This definition keeps things grounded in how systems are actually built and deployed. Concretely, an agent consists of three core components:

Instructions

> These define the role and behavior of the model.

A model

> This determines the agent's underlying capabilities and parameters.

Tools

> These extend the model's abilities through external functions or APIs.

Even within this setup, developers retain control over key aspects of execution, such as how many iterations are allowed, how conversation history is managed, which tools are exposed, and what parameters govern the model's behavior.

It is worth noting that not all organizations define agents in the same way. For example, in the article "Building Effective Agents", Anthropic distinguishes between an augmented LLM and an agent: an *augmented LLM* is simply a model with access to tools and memory, whereas an *agent* is a system that operates in a loop of *decide → act → observe.*

While this perspective is useful, especially when discussing more advanced systems, it introduces a boundary that is often less important in practice. In real-world applications, the more relevant question is not how we label the system but *who is making decisions*.

Here are two simple rules of thumb:

- If *your code* determines what actions are taken, the system is not an agent.
- If *the LLM* determines what actions are taken, the system is an agent.

This distinction helps clarify common patterns. A traditional retrieval-augmented generation (RAG) system, in which the application controls when documents are retrieved, is not an agent. However, once the LLM is responsible for deciding whether retrieval is necessary and which tools to call, the system becomes agentic.

This leads to an important conclusion: *not every system that uses an LLM is an agent*. From this perspective, agents are best understood not as a separate category but as a point on a spectrum. At one end are fully deterministic pipelines, in which every step is explicitly controlled. At the other are highly autonomous systems, in which the model dynamically decides how to achieve a goal. Most real-world systems fall somewhere in between.

In this chapter, the system we'll build sits at the simpler end of this spectrum. It is an agent in its most basic form: an LLM that is equipped with instructions and tools and is capable of making limited decisions. Despite its simplicity, it reflects the core patterns used in production systems.

Structure of the LLMOps Chapters

In this and the next few chapters, we will build a system that creates LinkedIn posts about recent arXiv papers on AI in a specific style. I realize this is a toy example and a different architecture might have made more sense for this use case, but my goal is to showcase all the Databricks components that are relevant for bringing AI systems to production.

Here is a list of the LLM system components we are going to build and discuss in the next few chapters:

Foundation models on Databricks (in Chapter 7)

> When you build these, you will be selecting and hosting the "brain" of your system.

Context engineering (in Chapter 7)

This involves managing the external data and history that ground your agent, and it includes the following:

Getting from raw data to an agent context

You'll do this to ingest and process external documents.

Using Model Context Protocol servers on Databricks

You'll do this to standardize tool access.

Using agent memory with Lakebase

You'll use this to maintain state across conversational sessions.

MLflow for AI agents (in Chapter 8)

When you build this, you'll be tailoring the experiment lifecycle for GenAI. The steps involved in this process are as follows:

Logging and registering an agent

This includes defining the logic of an agent, logging it in MLflow experiment tracking, and registering it in Unity Catalog.

MLflow tracing

This includes recording reasoning chains and tool calls for deep observability.

Agent evaluation

This involves systematically measuring output quality.

LLM system deployment (in Chapter 9)

You'll do this to operationalize the workflow.

LLM system monitoring (Chapter 9)

You'll do this to monitor safety and performance in the real world.

Figure 7-1 provides a visual representation of these components.

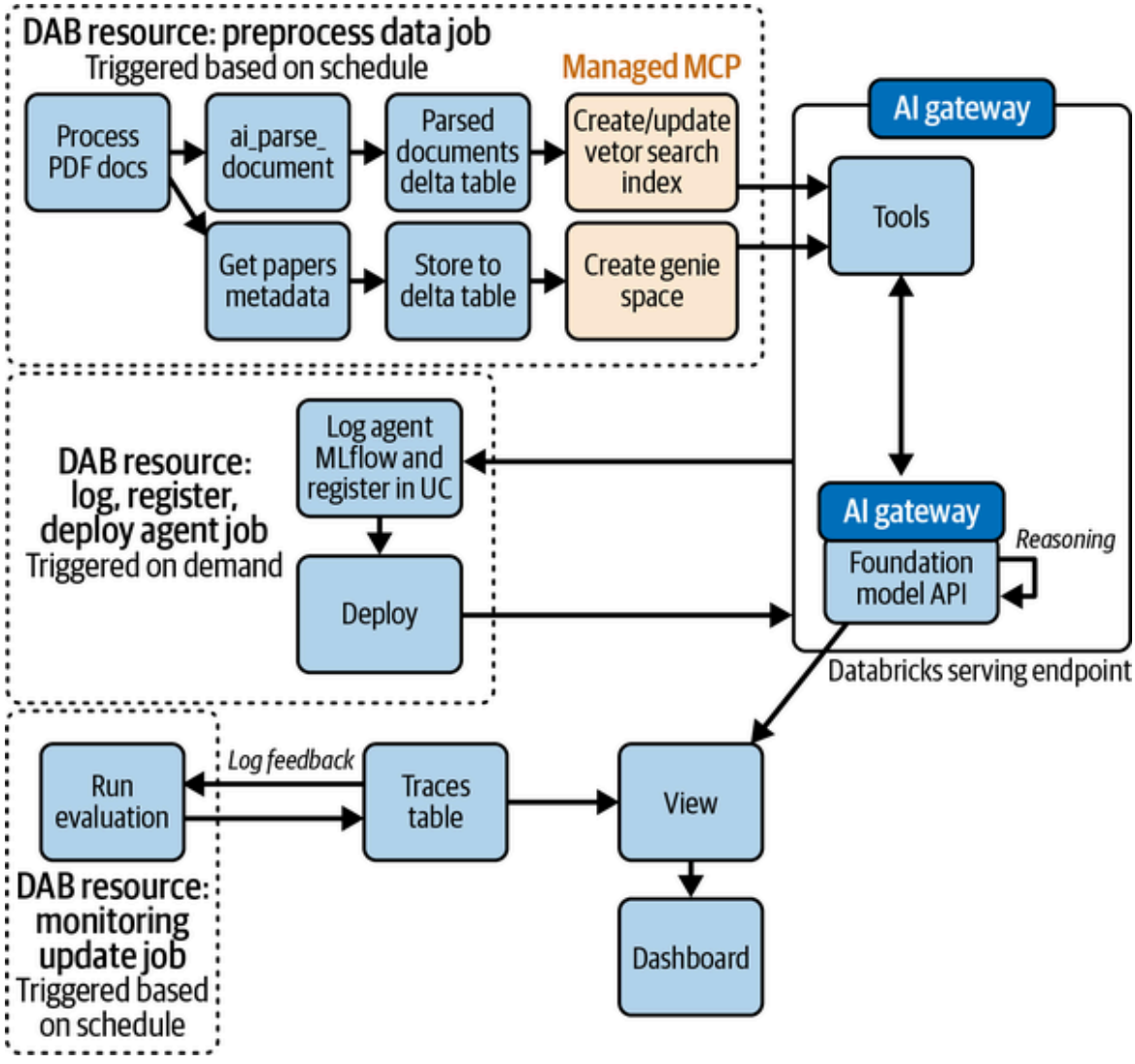


Figure 7-1. LLM system components

Let's go through these components one by one, starting with foundation models on the Databricks platform.

Foundation Models on Databricks

Working with foundation models on Databricks gives you flexibility: you can employ ready-to-use hosted models, deploy models with provisioned throughput, or connect to external providers like OpenAI or Anthropic. In this section, we'll explore each option, discuss supported model types, and show you how to query them.

Hosting Options

The Databricks platform hosts a variety of ready-to-use foundation model APIs, including models like Meta's Llama 4, OpenAI's GPT-5.5, and Anthropic's Claude Opus 4.8. Beyond these hosted models, you can deploy open-source models through provisioned throughput serving endpoints, or you can connect to external providers like Azure OpenAI or Amazon Bedrock.

Regardless of which option you choose, all models use Databricks Model Serving, which follows the OpenAI API specification. It supports three endpoint types:

llm/v1/chat

> This corresponds to the OpenAI Chat Completions API.

llm/v1/embeddings

> This corresponds to the OpenAI Embeddings API.

llm/v1/completions

> This follows the legacy OpenAI Completions API.

This unified interface lets you query foundation models in the same way, regardless of the provider or where the model is hosted. This consistency makes it much easier to experiment with different models, perform evaluations, and enable fallback strategies. You can even use an OpenAI client to interact with foundation models on Databricks. Note that at the

time of writing, the responses API is only supported for Databricks-hosted OpenAI closed-sourced foundation models.

Here is an example of how we can interact with the `databricks-gpt-oss-120b` model:

```
from databricks.sdk import WorkspaceClient
from openai import OpenAI

w = WorkspaceClient()
host = w.config.host
token = w.tokens.create(lifetime_seconds=1200).token_value

client = OpenAI(
   api_key=token,
   base_url=f"{host}/serving-endpoints"
)

response = client.chat.completions.create(
   model="databricks-gpt-oss-120b",
   messages=[
       {"role": "user",
        "content": "What is MLOps?"}
   ]
)
```

Another benefit of using foundation models on Databricks is access to the Databricks Unity AI Gateway (see Figure 7-2). We deployed an AI gateway for our Model Serving endpoint earlier in the book and demonstrated how it enables usage tracking, payload logging, and traffic management. We can apply AI Gateway to any Databricks Model Serving endpoint, including foundation model APIs. We can also enable Databricks AI Guardrails to ensure PII data is deleted or masked and harmful content is blocked, for both model input and output.

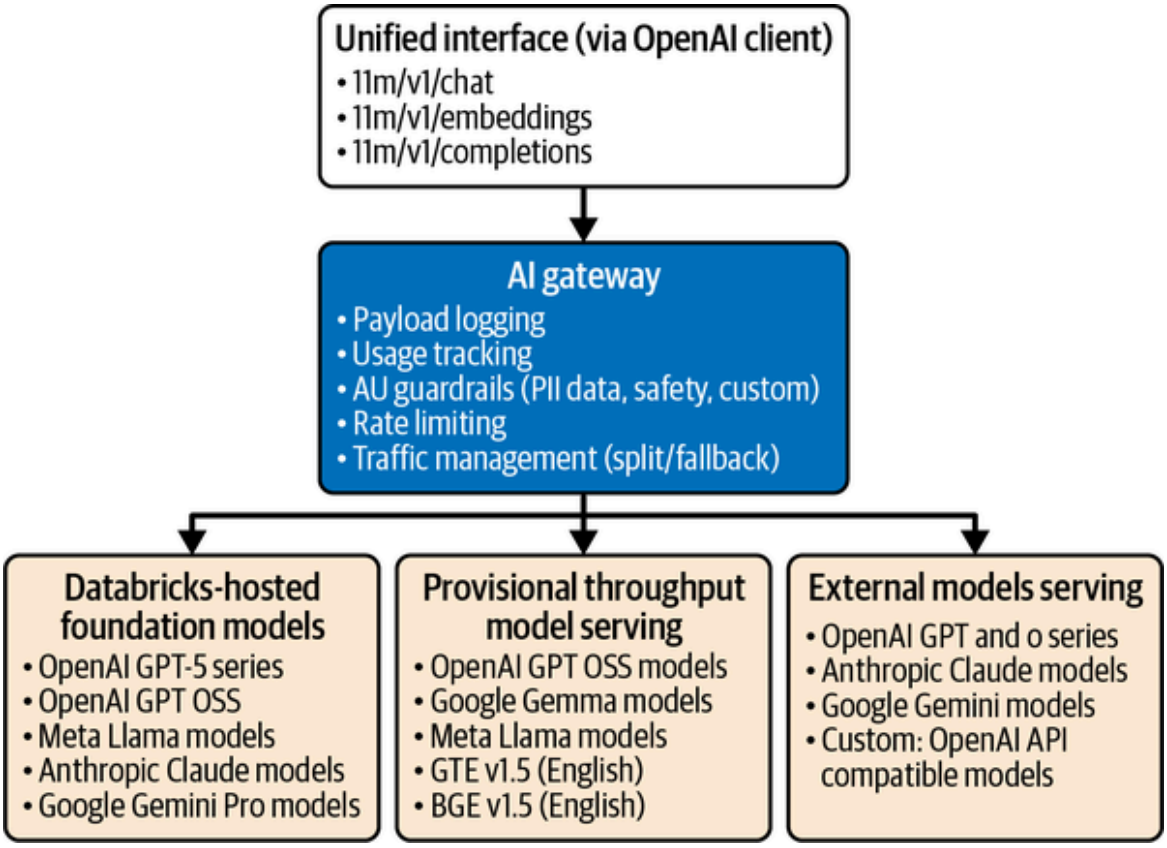


Figure 7-2. Depiction of Databricks AI Gateway

We can also use foundation model APIs in batch mode by using `ai_query`.[2] Some Databricks-hosted models are optimized for batch processing, but `ai_query` works for any endpoint deployed through Databricks Model Serving.

As we mentioned earlier, Databricks offers ready-to-use hosting options for foundation models. We discuss them in detail in the following sections.

Databricks-hosted models (pay-per-token)

These models are ideal for experimentation and exploration, as they require no additional configuration or upfront infrastructure commitment. The availability of these models in your workspace will depend on your cloud provider and region.[3] However, they are subject to workspace-level rate limits in input/output tokens and queries per hour. The most restrictive rate limit applies at any given time; if you exceed it, you will receive a 429 error until the window resets.

Provisioned throughput endpoints

These endpoints are recommended for production, and they provide performance guarantees. You can deploy models behind provisioned throughput endpoints directly from the `system.ai` schema. You can also bring your own model (perhaps one you've fine-tuned) from sources like Hugging Face. If you use the latter approach, you'll need to log the model using MLflow (using the transformers flavor) and register it in Unity Catalog.

Provisioned throughput works only for selected foundation model families and for specific runtime configurations (size and context window) that Databricks has tuned and supports.[4] If you choose to bring your own model and its metadata falls outside supported ranges, the provisioned throughput endpoint creation will fail too.[5]

For some models, provisioned throughput is expressed in tokens per second (input and output tokens combined), rather than requests per second. Since LLM requests vary widely in token length, requests per second is no longer a reliable measure of capacity. For example, if a request requires 2,048 input tokens and 256 output tokens (for 2,304 tokens in total), a provisioned throughput of 2,304 tokens per second will allow the endpoint to process one such request per second. If you increase this to 4,608 tokens per second, the endpoint will be able to process two such requests per second, with each request still taking roughly one second to complete.

Optimizing LLM inference performance is not an easy task, and GPU autoscaling is slower than CPU autoscaling for regular Model Serving endpoints. Databricks recommends conducting benchmarking to help you choose the correct range of tokens per second.[6] You can also scale the endpoint to 0 to avoid unnecessary costs, but that is not recommended for production environments.

For other models, provisioned throughput is expressed in model units, which is an abstraction layer on top of tokens per second. For the Llama 4 Maverick model, for example, 50 model units means approximately 3,250 tokens per second.

External models

External models are hosted outside of Databricks, which means you may need to configure firewall rules to access them.[7] Various providers are supported, and you can also use custom providers as long as their APIs are OpenAI-compatible.

Even though you could access external models directly from the agent logic (as long as the secrets are configured and the networking is resolved), creating a serving endpoint that accesses an external model is a good idea because of the benefits mentioned previously, specifically around governance.

You can also use the custom provider option when you need to use functionalities, such as those offered by OpenAI, that are not yet covered by out-of-the-box solutions. For instance, while Databricks' existing OpenAI external Model Serving options currently support text and image inputs with text-only outputs, the Custom Provider solution enables support for images as output (encoded as base64) use cases through Databricks.[8]

Types of Foundation Models

Foundation models on Databricks are categorized into four functional types to help you select the most effective tool for your specific application:

General purpose models

> These models are fine-tuned on large datasets of human dialogue, which enables them to generate contextually relevant responses in multiturn conversations. They are ideal for use cases such as customer support, virtual assistants, and interactive tutoring systems. Examples include Anthropic Claude models, Google Gemini models, and OpenAI GPT series models.[9]

Embedding models

These models transform complex data, such as text, images, and audio, into embeddings that capture the essential features and relationships within data. They are recommended for applications that require semantic understanding, similarity comparison, and efficient retrieval or clustering of complex data. Examples include Cohere, Databricks, OpenAI, and Google text embedding models.

Vision models

These models are designed to analyze visual data, and they are used for object recognition and image classification. Note that while they accept image inputs, standard configurations typically produce text outputs unless you're using a Custom Provider setup. Examples include Anthropic Claude models, Google Gemini models, and OpenAI GPT series models.

Reasoning models

These models excel at breaking down complex tasks and explaining their decision-making process. They are best suited to content creation, summarization, and agent orchestration. Examples include Anthropic Claude models, Google Gemini models, and OpenAI GPT series models with reasoning capabilities.

In this book, we are going to use the `databricks-gte-large-en` embedding model to generate embeddings and store them in the AI Search index. For the core LLM logic within the agent definition, we will use `databricks-gpt-oss-120b`.

Now that we have a better understanding of foundation models on Databricks, let's move on to one of the most important topics when it comes to building agents: context engineering.

Context Engineering

In the early days of AI engineering, it was all about prompt engineering. But while well-crafted system prompts remain important today, modern agent systems demand much more. We need strategies to manage the entire *context state*, which includes external data, available tools, conversation history, and intermediate reasoning steps (see Figure 7-3).

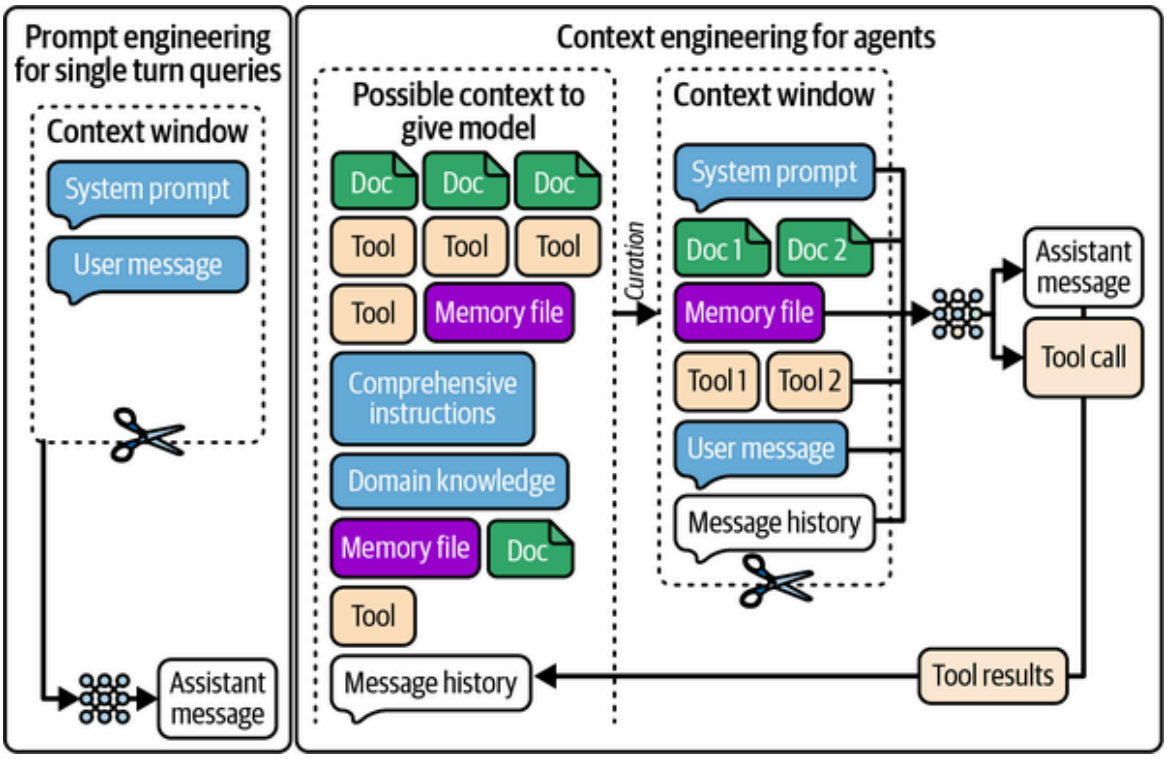


Figure 7-3. Prompt engineering vs context engineering, from "Effective context engineering for AI agents" by Anthropic

Consider an agent running in a loop. Each *turn* of the loop generates new information (tool outputs, retrieved documents, and user clarifications) that may be relevant for subsequent reasoning. But context windows are finite. The challenge isn't just *getting* information to the model; it's about shaping more behavior and *curating* what goes in, *when*, and *how.* This is what modern *context engineering* is all about.

In this section, we'll explore three core aspects of context engineering:

Getting from raw data to agent context

This is how agents access and retrieve information beyond their training data.

Model Context Protocol servers on Databricks

This is a standardized protocol for exposing tools and data sources to agents.

Using agent memory with Lakebase

This is how you maintain state and conversation history across interactions.

Reference Architecture: The arXiv Curator

To make these concepts concrete, we'll build a system that generates LinkedIn posts about recent arXiv papers on AI that are written in a consistent style. To achieve this, our system will perform the following steps:

1. Download and process full paper PDFs by extracting the complete text, not just metadata. This will give us access to results, examples, and insights buried deep in the paper.
2. Chunk and embed the content by storing vector representations in a Databricks AI Search index for semantic retrieval.
3. Track paper metadata in a Delta table by recording which papers we've processed, when, and their key attributes. Then, we'll also create a Genie agent (formerly, Genie space) on top of this table for natural language querying.
4. Expose these resources via Model Context Protocol (MCP). Both the AI Search index and Genie agent are available as managed Databricks MCP servers. This means our agent can access them through a standardized tool interface, which we'll explore in detail in the next section.

Now, we could take a simpler approach: the arXiv API already provides titles and abstracts, which might be sufficient for basic posts. But we want to create higher-quality, more insightful content that's hard to create based on the abstracts.

I intentionally made the architecture more sophisticated than is strictly necessary for our toy example. These patterns (PDF ingestion, vector retrieval, structured metadata, and tool protocols) appear in nearly every production LLM system. Building them here, even for a simple use case, teaches transferable skills you'll apply to more complex scenarios.

For a real production system, though, I encourage you to think the other way around: go simple first and more complex only if necessary. I recommend checking out the blog by Anthropic on building effective agents to learn about using different complexity levels when it comes to AI systems.

Now, let's dive into the details of how we can get from raw data to an agent context for our use case.

Getting from Raw Data to Chunks

Before we dive into the ingestion and data processing logic, let's review the project configuration, as it provides important context for the steps that follow.

Project configuration

The code for this LLMOps chapter is organized like the earlier MLOps examples. You can find it in the same GitHub repository under the *llmops* folder.

This is what the *project-config.yml* file looks like:

```
system_prompt: |
 You are a LinkedIn content creation assistant specialized
 in generating engaging posts about AI and machine learning research.

 Your role is to:
```

```
  1. Search for relevant arXiv papers using the AI Search tool
     based on the user's topic
  2. Query the Genie agent for additional context and insights
     about the research area
  3. Generate a LinkedIn post in a professional yet engaging style that:
     - Highlights key findings or innovations from recent research
     - Makes complex technical concepts accessible to a broad audience
     - Includes relevant paper citations and IDs
     - Uses a conversational tone appropriate for LinkedIn
     - Incorporates 2-3 relevant hashtags
     - Keeps the post concise (150-250 words)

  Always ground your posts in actual research findings from the tools
  available to you.

  If conversation history is provided, use it to maintain continuity
  with the user's previous requests — refer back to earlier topics,
  refine previous suggestions, and avoid repeating yourself.

experiment_path: /Shared/genai-arxiv-agent
usage_policy_id: 686c435d-90af-4c40-9ca4-442d4a3d8899
lakebase_project_id: arxiv-agent-lakebase

dev:
  catalog: mlops_dev
  schema: arxiv
  volume: arxiv_papers
  genie_space_id: 01f12f9b759a1da0807c0ae7366ee04d
  llm_endpoint: databricks-gpt-oss-120b
  warehouse_id: 1883724c3376103a

acc:
  catalog: mlops_acc
  schema: arxiv
  volume: arxiv_papers
  genie_space_id: 01f12f9b759a1da0807c0ae7366ee04d
  llm_endpoint: databricks-gpt-oss-120b
  warehouse_id: 1883724c3376103a

prd:
  catalog: mlops_prd
  schema: arxiv
  volume: arxiv_papers
  genie_space_id: 01f12f9b759a1da0807c0ae7366ee04d
  llm_endpoint: databricks-gpt-oss-120b
  warehouse_id: 1883724c3376103a
```

Notice the usage policy ID in the *project_config.yml* file. It differs from the one we used earlier. For the LLMOps chapter, we created a separate usage policy to isolate and track costs independently (see Figure 7-4). When we apply the LLMOps usage policy, it automatically adds the `project=arxiv_curator` tag, rather than `project=hotel_booking`, which we used previously.

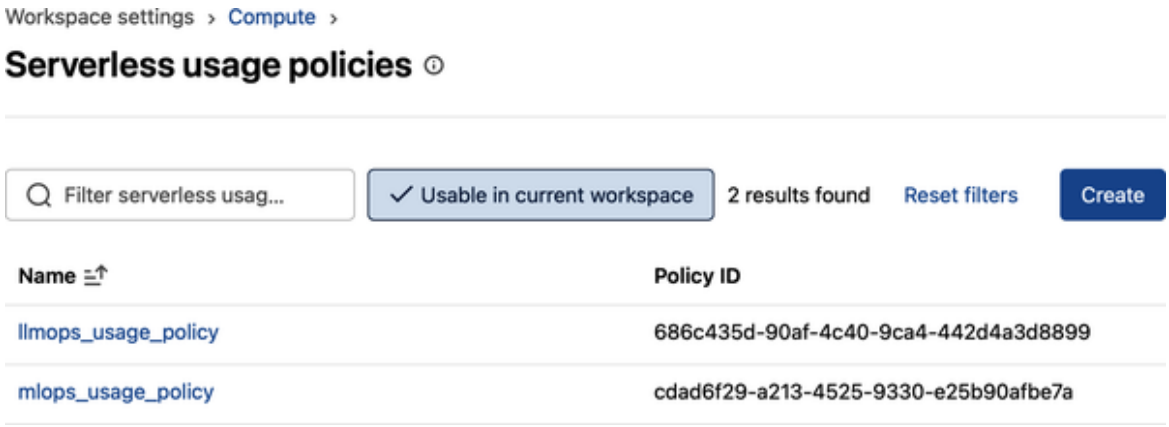


Figure 7-4. Serverless usage policies created for the MLOps and LLMOps chapters

You may notice that the policy again includes three environment-specific configurations, along with a shared configuration for the system prompt. We'll examine these in detail later in this chapter. Although we could use the MLflow Prompt Registry instead, it is primarily beneficial for prompt optimization workflows, and we would have to combine it with managing prompts as code.

Just as when we were processing data for our ML use case, we use Pydantic to verify the configuration, and define the `ProjectConfig` class under the config module:

```
import yaml
from pydantic import BaseModel


class ProjectConfig(BaseModel):
    """Configuration for the arxiv curator project."""

    catalog: str
    schema: str
    volume: str
```

```
    genie_space_id: str
    llm_endpoint: str
    system_prompt: str
    warehouse_id: str
    experiment_path: str
    usage_policy_id: str | None = None
    lakebase_project_id: str | None = None

    @classmethod
    def from_yaml(
        cls: "ProjectConfig", config_path: str, env: str = "dev"
    ) -> "ProjectConfig":
        """
        Load configuration from YAML file.

        Args:
            config_path: Path to the project_config.yml file
            env: Environment name (dev, acc, prd)

        Returns:
            ProjectConfig instance
        """
        if env not in ["prd", "acc", "dev"]:
            raise ValueError(
                f"Invalid environment: {env}. Expected 'prd', 'acc', or 'dev'"
            )

        with open(config_path) as f:
            config_dict = yaml.safe_load(f)
        env_config = config_dict[env]
        env_config["system_prompt"] = config_dict["system_prompt"]
        env_config["experiment_path"] = config_dict["experiment_path"]
        env_config["usage_policy_id"] = config_dict.get("usage_policy_id")
        env_config["lakebase_project_id"] =
config_dict.get("lakebase_project_id")
        return cls(**env_config)
```

Now that we understand the project configuration, we can use it in the data ingestion logic.

Data ingestion

We will utilize the [arXiv Python package] to search for the latest papers in the AI category from the ArXiv database. We will then download the papers, store them in a volume, and save all the metadata in the

`arxiv_papers` table within the configured catalog and schema. The code shown here can be found in the GitHub repository under *notebooks/chapter7_1.arxiv_process_docs.py* in the *llmops* folder. The packaged version of the code can be found in the *src/arxiv_curator* folder under the `data_processor` module.

Here, we use incremental processing based on timestamps. If the `arxiv_papers` table exists, the system queries the timestamp of the most recently processed paper and searches arXiv for all papers published between that timestamp and the current execution time. This ensures that only new papers will be downloaded and processed, to avoid redundant work.

> **NOTE**
>
> Databricks Free Edition has network restrictions that block outbound API calls to certain external services. If you use the Free Edition, you won't be able to call the arXiv API in a Databricks notebook. Instead, you should consider developing locally and then uploading files to Unity Catalog Volumes using Databricks CLI.

First, let's load the project configuration and retrieve the catalog name, schema name, and volume name:

```
from loguru import logger
from pyspark.sql import SparkSession

from arxiv_curator.config import ProjectConfig

spark = SparkSession.builder.getOrCreate()

cfg = ProjectConfig.from_yaml("../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema
volume = cfg.volume
```

Next, we need to make sure the schema and volume exist. (Note that we're reusing the same catalog that we used in the MLOps chapters.):

```
spark.sql(f"CREATE SCHEMA IF NOT EXISTS {catalog}.{schema}")
spark.sql(f"CREATE VOLUME IF NOT EXISTS {catalog}.{schema}.{volume}")
```

For initial runs in which the `arxiv_papers` table does not yet exist, the system retrieves papers from the last three days. This provides a meaningful initial dataset without overwhelming the system with arXiv's extensive historical archives:

```
import time

import arxiv

client = arxiv.Client()
metadata_table = f"{catalog}.{schema}.arxiv_papers"

if spark.catalog.tableExists(metadata_table):
   start = str(
       spark.sql(f"""
       SELECT max(processed)
       FROM {metadata_table}
       """).collect()[0][0]
   )
else:
   start = time.strftime("%Y%m%d%H%M", (time.gmtime(time.time() - 24 * 3600 *
3)))

end = time.strftime("%Y%m%d%H%M", time.gmtime())

search = arxiv.Search(query=f"cat:cs.AI AND submittedDate:[{start} TO {end}]")
papers = client.results(search)
```

Once the system retrieves the search results, it processes each paper individually. For every paper, it extracts the following metadata: paper ID, title, author list, abstract, URL, and publication timestamp. Additionally, the system appends processing metadata, including the timestamp when the paper was processed and the volume path where the downloaded paper is stored:[10]

```
import os

records = []
pdf_dir = f"/Volumes/{catalog}/{schema}/{volume}/{end}"
```

```
os.makedirs(pdf_dir, exist_ok=True)

for paper in papers:
    paper_id = paper.get_short_id()
    # download PDF
    try:
        paper.download_pdf(dirpath=pdf_dir, filename=f"{paper_id}.pdf")
        # collect metadata (keep datetime intact)
        records.append(
            {
                "paper_id": paper_id,
                "title": paper.title,
                "authors": [author.name for author in paper.authors],
                "summary": paper.summary,
                "pdf_url": paper.pdf_url,
                "published": int(paper.published.strftime("%Y%m%d%H%M")),
                "processed": int(f"{end}"),
                "volume_path": f"{pdf_dir}/{paper_id}.pdf",
            }
        )
    except Exception:
        logger.warning(f"Paper {paper_id} was not successfully processed.")
        pass
    time.sleep(3)  # to avoid rate limiting
```

Finally, we write the retrieved metadata into the `arxiv_papers` table:

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

if len(records) > 0:
    metadata_schema = T.StructType(
        [
            T.StructField("paper_id", T.StringType(), False),
            T.StructField("title", T.StringType(), True),
            T.StructField("authors", T.ArrayType(T.StringType()), True),
            T.StructField("summary", T.StringType(), True),
            T.StructField("pdf_url", T.StringType(), True),
            T.StructField("published", T.LongType(), True),
            T.StructField("processed", T.LongType(), True),
            T.StructField("volume_path", T.StringType(), True),
        ]
    )

    metadata_df = spark.createDataFrame(records,
schema=metadata_schema).withColumn(
```

```
        "ingest_ts", F.current_timestamp()
    )
    metadata_df.write.format("delta").mode("append").saveAsTable(f"
{metadata_table}")
```

Now that the papers are stored in the volume and all the metadata is stored in a Delta table, we're going to tackle PDF processing—one of those problems that looks simple until you actually try to do it at scale.

PDF processing

PDFs are everywhere in enterprise environments, and they're consistently one of the hardest data sources to work with reliably. That's because PDFs weren't designed for data extraction. They were designed to ensure that a document looks exactly the same all the time, no matter where you view or print it.

PDFs are built for visual presentation, but they lack metadata to process content reliably. They may look organized on screen, but they are unstructured behind the scenes. This makes them incredibly frustrating to work with when you need to extract structured data for processing, searches, or feeding into your AI systems.

You may find several options when it comes to PDF processing (including scanned documents): open-source projects like Docling, or perhaps solutions provided by cloud vendors like Amazon Textract and Azure Document Intelligence. Also, modern LLMs with vision capabilities can now process PDFs directly. Models like GPT-5, Claude Sonnet 4.5, and Gemini 3 Pro can see the document layout, understand context, handle both native text and scanned images, and extract structured data in a single pass.

Databricks also provides an `ai_parse_document` function that significantly simplifies the PDF processing pipeline and handles all of this complexity behind the scenes, providing high accuracy for a competitive price. Let's see how it works in practice.

> **NOTE**
>
> The `ai_parse_document` function is not supported while executing code locally via VS Code.

First, let's create a table called `ai_parsed_docs` to store our extraction results. We'll insert the document volume path alongside the parsed content, which comes from the `ai_parse_document` function running against all the PDF files we downloaded earlier in this pipeline:

```
spark.sql(
    f"""
CREATE TABLE IF NOT EXISTS
{catalog}.{schema}.ai_parsed_docs (
path STRING,
parsed_content STRING,
processed LONG)
"""
)

spark.sql(

    f"""
INSERT INTO {catalog}.{schema}.ai_parsed_docs
SELECT
path,
ai_parse_document(content) AS parsed_content,
{end} AS processed
FROM READ_FILES(
"{pdf_dir}",
format => 'binaryFile')
"""
)
```

The output of the `ai_parse_document` function consists of elements (chunks) of the following types:

caption

This includes image and figure captions.

figure

: This includes figures and images.

footnote

: This includes footnotes.

page_header

: This includes page headers (like page numbers).

section_header

: This includes section headings.

table

: This includes tables.

text

: This includes regular text content.

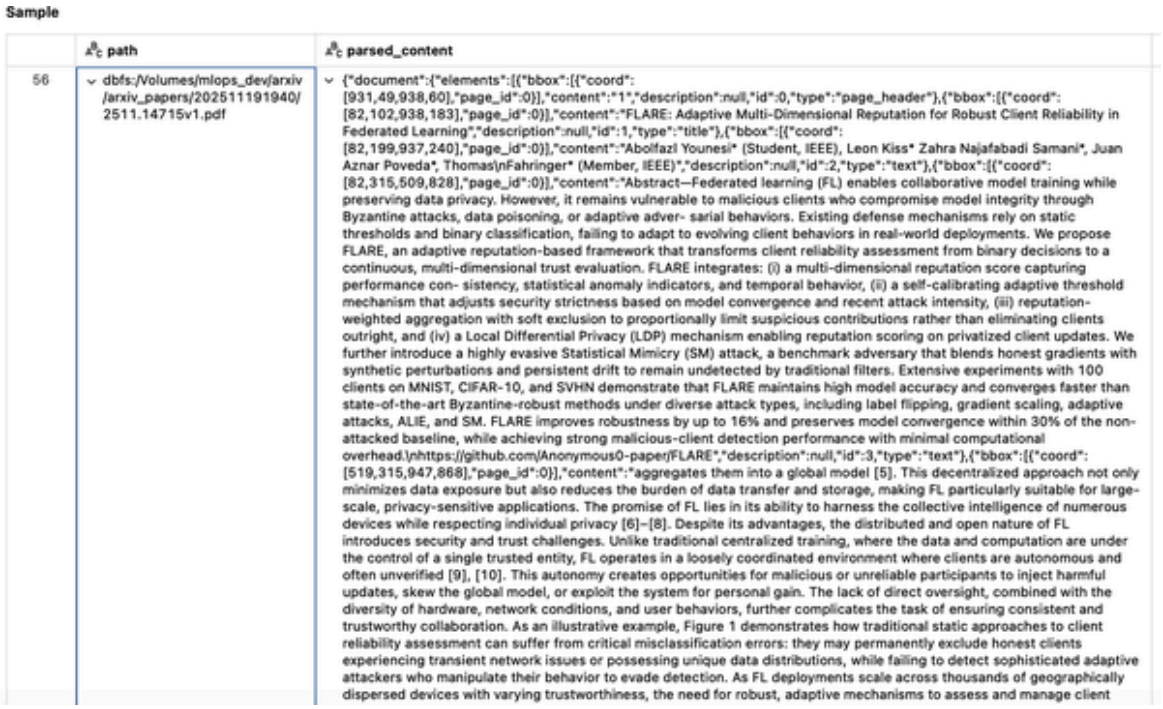
title

: This includes the document title.

Figure 7-5 shows this output.

The `ai_parse_document()` function returns a document that's already broken into pieces along its own structure, with each piece tagged by type. For retrieval, the type that matters most is `text`, since that is where the answerable content sits. In effect, the parser has made the first chunking decision for us, one driven by the document's structure.

In this chapter, we go with the chunking that `ai_parse_document()` produces, to keep things simple. However, it is still worth it to learn about chunking strategy in its own right, since the parser's structural split is not always the one that serves retrieval best.


Sample

	path	parsed_content
56	dbfs:/Volumes/mlops_dev/arxiv/arxiv_papers/202511191940/2511.14715v1.pdf	{"document":{"elements":[{"bbox":[{"coord":[931,49,938,60],"page_id":0}],"content":"1","description":null,"id":0,"type":"page_header"},{"bbox":[{"coord":[82,102,938,183],"page_id":0}],"content":"FLARE: Adaptive Multi-Dimensional Reputation for Robust Client Reliability in Federated Learning","description":null,"id":1,"type":"title"},{"bbox":[{"coord":[82,199,937,240],"page_id":0}],"content":"Abolfazl Younesi* (Student, IEEE), Leon Kiss* Zahra Najafabadi Samani*, Juan Aznar Poveda*, Thomas\nFahringer* (Member, IEEE)","description":null,"id":2,"type":"text"},{"bbox":[{"coord":[82,315,509,828],"page_id":0}],"content":"Abstract—Federated learning (FL) enables collaborative model training while preserving data privacy. However, it remains vulnerable to malicious clients who compromise model integrity through Byzantine attacks, data poisoning, or adaptive adver- sarial behaviors. Existing defense mechanisms rely on static thresholds and binary classification, failing to adapt to evolving client behaviors in real-world deployments. We propose FLARE, an adaptive reputation-based framework that transforms client reliability assessment from binary decisions to a continuous, multi-dimensional trust evaluation. FLARE integrates: (i) a multi-dimensional reputation score capturing performance con- sistency, statistical anomaly indicators, and temporal behavior, (ii) a self-calibrating adaptive threshold mechanism that adjusts security strictness based on model convergence and recent attack intensity, (iii) reputation-weighted aggregation with soft exclusion to proportionally limit suspicious contributions rather than eliminating clients outright, and (iv) a Local Differential Privacy (LDP) mechanism enabling reputation scoring on privatized client updates. We further introduce a highly evasive Statistical Mimicry (SM) attack, a benchmark adversary that blends honest gradients with synthetic perturbations and persistent drift to remain undetected by traditional filters. Extensive experiments with 100 clients on MNIST, CIFAR-10, and SVHN demonstrate that FLARE maintains high model accuracy and converges faster than state-of-the-art Byzantine-robust methods under diverse attack types, including label flipping, gradient scaling, adaptive attacks, ALIE, and SM. FLARE improves robustness by up to 16% and preserves model convergence within 30% of the non-attacked baseline, while achieving strong malicious-client detection performance with minimal computational overhead.\nhttps://github.com/Anonymous0-paper/FLARE","description":null,"id":3,"type":"text"},{"bbox":[{"coord":[519,315,947,868],"page_id":0}],"content":"aggregates them into a global model [5]. This decentralized approach not only minimizes data exposure but also reduces the burden of data transfer and storage, making FL particularly suitable for large-scale, privacy-sensitive applications. The promise of FL lies in its ability to harness the collective intelligence of numerous devices while respecting individual privacy [6]–[8]. Despite its advantages, the distributed and open nature of FL introduces security and trust challenges. Unlike traditional centralized training, where the data and computation are under the control of a single trusted entity, FL operates in a loosely coordinated environment where clients are autonomous and often unverified [9], [10]. This autonomy creates opportunities for malicious or unreliable participants to inject harmful updates, skew the global model, or exploit the system for personal gain. The lack of direct oversight, combined with the diversity of hardware, network conditions, and user behaviors, further complicates the task of ensuring consistent and trustworthy collaboration. As an illustrative example, Figure 1 demonstrates how traditional static approaches to client reliability assessment can suffer from critical misclassification errors: they may permanently exclude honest clients experiencing transient network issues or possessing unique data distributions, while failing to detect sophisticated adaptive attackers who manipulate their behavior to evade detection. As FL deployments scale across thousands of geographically dispersed devices with varying trustworthiness, the need for robust, adaptive mechanisms to assess and manage client

Figure 7-5. Output of `ai_parse_document` function

Chunking

Chunking decides what the retriever can find. A *chunk* is the unit that gets embedded and later returned, so if the answer to a question is split across two chunks or buried in a chunk full of unrelated text, then retrieval suffers no matter how strong the rest of the pipeline is. The strategies we cover in the following sections make trade-offs in three areas: how well a chunk respects the content's structure, how much it inflates your index, and how it fits inside your embedding model's limits.

Chunking strategies

The simplest approach splits documents into equal-length chunks by some unit: characters, words, sentences, or paragraphs. It works well enough for many use cases but has one fundamental problem: it ignores the structure of your content. A 2,048-character chunk may cut off mid-sentence, mid-paragraph, or in the middle of a critical explanation, and the context at those boundaries may therefore be lost.

Overlap is the usual fix. With 2,048-character chunks, you can carry 200 characters of overlap between them, so information near a boundary will still land intact in at least one chunk. The cost is duplication: you will store some content twice, and your index will grow.

A recursive split respects structure more directly. Take an arXiv paper: start at the section boundaries (Abstract, Introduction, Methods, and Results), and if a section exceeds your maximum chunk size, divide it further by subsection (Data Collection, Model Architecture, and Training Procedure), and finally by paragraph. This hierarchy keeps related content together and avoids breaking an explanation across an arbitrary boundary, so what comes back stays contextually complete.

Embedding window considerations

Chunk size is not entirely your choice. It is bounded by the embedding window of the model you pick. The Databricks GTE model has a window of 8,192 tokens, while Databricks BGE Large has 512. If a chunk exceeds the window, it cannot be embedded in full and part of what you intended to index will never make it in.

These limits are measured in tokens, not characters, and the two are not interchangeable. A *token* can be a word, a subword, or a punctuation mark, depending on the model's tokenizer. The phrase "MLOps is awesome!" consists of 17 characters but four tokens: `"MLOps"`, `"is"`, `"awesome"`, and `"!"`. When you set a chunk size in characters, keep this conversion in mind so your chunks stay inside the window.

Processing chunks

To make the data usable for vector search, we need to create another table that stores each chunk in a separate row. Let's create a user-defined function to extract the chunks. Here, we are only extracting chunks of type `text`:

```
import json
import re

from pyspark.sql.functions import (
   col,
```

```
    concat_ws,
    explode,
    udf)

from pyspark.sql.types import (
    ArrayType,
    StringType,
    StructField,
    StructType)


def extract_chunks(parsed_content_json: str) -> list[tuple[str, str]]:
    parsed_dict = json.loads(parsed_content_json)
    chunks = []

    for element in parsed_dict.get("document", {}).get("elements", []):
        if element.get("type") == "text":
            chunk_id = element.get("id", "")
            content = element.get("content", "")
            chunks.append((chunk_id, content))
    return chunks

chunk_schema = ArrayType(
    StructType(
        [
            StructField("chunk_id", StringType(), True),
            StructField("content", StringType(), True),
        ]
    )
)
extract_chunks_udf = udf(extract_chunks, chunk_schema)
```

We also want to extract the paper ID from the path and clean chunks (remove whitespaces, fix hyphenation, and collapse new lines into spaces):

```
def extract_paper_id(path):
    return path.replace(".pdf", "").split("/")[-1]

extract_paper_id_udf = udf(extract_paper_id, StringType())

def clean_chunk(text: str) -> str:
    # fix hyphenation across line breaks:
    # "docu-\nments" => "documents"
    t = re.sub(r"(\w)-\s*\n\s*(\w)", r"\1\2", text)

    # collapse internal newlines into spaces
```

```
    t = re.sub(r"\s*\n\s*", " ", t)

    # collapse repeated whitespace
    t = re.sub(r"\s+", " ", t)

    return t.strip()

clean_chunk_udf = udf(clean_chunk, StringType())
```

Finally, we can create the `arxiv_chunks` table and enable `ChangeDataFeed` so that we can use the table to create and incrementally update the AI Search index. Notice that we are joining some fields from the metadata table so that we can perform reranking and filtering later:

```
df = spark.table(f"{catalog}.{schema}.ai_parsed_docs").where(f"processed =
{end}")

# Load metadata table
metadata_df = spark.table(metadata_table).select(
    col("paper_id"),
    col("title"),
    col("summary"),
    concat_ws(", ", col("authors")).alias("authors"),
    (col("published") / 100000000).cast("int").alias("year"),
    ((col("published") % 100000000) / 1000000).cast("int").alias("month"),
    ((col("published") % 1000000) / 10000).cast("int").alias("day"),
)

# Create the transformed table
chunks_df = (
    df.withColumn("paper_id", extract_paper_id_udf(col("path")))
    .withColumn("chunks", extract_chunks_udf(col("parsed_content")))
    .withColumn("chunk", explode(col("chunks")))
    .select(
        col("paper_id"),
        col("chunk.chunk_id").alias("chunk_id"),
        clean_chunk_udf(col("chunk.content")).alias("text"),
        concat_ws("_", col("paper_id"), col("chunk.chunk_id")).alias("id"),
    )
    .join(metadata_df, "paper_id", "left"))

# Write to table
chunks_table = f"{catalog}.{schema}.arxiv_chunks"
chunks_df.write.mode("append").saveAsTable(chunks_table)
```

```
spark.sql(
   f"""ALTER TABLE {chunks_table}
   SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
   """
)
```

At this point in the pipeline, we have the arXiv papers stored as PDFs in volumes, their metadata stored in Delta tables, the parsed content extracted with `ai_parse_document()`, and the chunks stored in a Delta table. What we still lack is a way to find the right chunks for a given question. That is what Databricks AI Search provides, and setting it up is the focus of the next section.

Databricks AI Search Index

Vector search is fundamentally different from traditional search. Instead of matching keywords, it works with *embeddings*—mathematical representations (vectors) of semantic content. Text with similar meanings produces similar vectors, even if the exact words are different.

For example, in scientific papers, researchers don't always use the same terminology. One paper may discuss convolutional neural networks while another talks about CNNs or ConvNets. Databricks AI Search handles this naturally because it understands that these concepts are semantically related.

Creating the endpoint and index

To get started, we must first create an AI Search endpoint. Databricks AI Search provides standard and storage-optimized endpoint configurations. Standard endpoints have a capacity of 320 million vectors at dimension 768, while storage-optimized ones can accommodate over one billion vectors of that dimension and provide 10 to 20 times faster indexing. Queries on storage-optimized endpoints have a slightly increased latency, and there are some differences in supported operations among different endpoint types.[11]

We can create an AI Search endpoint programmatically by using the `VectorSearchClient`. Notice that here, we pass the usage policy ID that we will use later for cost allocation related to the AI Search endpoint:

```
from databricks.vector_search.client import VectorSearchClient

from arxiv_curator.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")
catalog= cfg.catalog
schema = cfg.schema

vsc = VectorSearchClient()
vector_search_endpoint_name = "vector-search-arxiv-endpoint"
if not vsc.endpoint_exists(name=vector_search_endpoint_name):
   vsc.create_endpoint_and_wait(
       name=vector_search_endpoint_name,
       endpoint_type="STANDARD",
       usage_policy_id=cfg.usage_policy_id)
```

Once the endpoint is active, we can proceed with creating an AI Search index. We use the `create_delta_sync_index` function from the `VectorSearchClient`. This function takes the following arguments:[12]

`columns_to_sync`

: These are the columns to transfer from the source table. If this argument is undefined, all columns will be synced.

`endpoint_name`

: This is the name of the AI Search endpoint where the index is served.

`source_table_name`

: This is the Delta table that contains the input data.

`index_name`

: This is the full name for the new index.

`primary_key`

> This is the unique identifier (in our case, the ID from the chunks table).

`embedding_source_column`

> This is the column that contains the text to be embedded.

`embedding_model_endpoint_name`

> This is the foundation model API that is used to generate embeddings. (We use the Databricks-hosted databricks-gte-large-en model.)

`pipeline_type`

> This is the type of Lakeflow SDPthat is used for synchronization. (We have chosen “TRIGGERED”.)

Notice that here, we pass the usage policy ID that we will use later for cost allocation related to the AI Search index. (So think about costs related to Lakeflow SDP runs.):

```
vs_index_fullname = f"{catalog}.{schema}.arxiv_index"
embedding_model_endpoint = "databricks-gte-large-en"

if not vsc.index_exists(
   endpoint_name=vector_search_endpoint_name,
   index_name=vs_index_fullname,
):
   index = vsc.create_delta_sync_index(
       endpoint_name=vector_search_endpoint_name,
       source_table_name=f"{catalog}.{schema}.arxiv_chunks",
       index_name=vs_index_fullname,
       pipeline_type="TRIGGERED",
       primary_key="id",
       embedding_source_column="text",
       embedding_model_endpoint_name=embedding_model_endpoint,
       usage_policy_id=cfg.usage_policy_id
   )
```

```
else:
    index = vsc.get_index(index_name=vs_index_fullname)
    index.sync()
```

When the new data is available in the source table, we will have to run the sync command:

```
index = vsc.get_index(index_name=vs_index_fullname)
index.sync()
```

As we discussed in earlier chapters regarding Feature Serving and the Databricks Online Feature Store, the same logic applies to synchronization modes here: we use `TRIGGERED` mode unless we consistently need fresh data. For lower latency requirements, `CONTINUOUS` mode ensures that updates arrive within seconds.

In the Catalog Explorer, you can view which Lakeflow SDP is associated with the AI Search index, along with its current status (see Figure 7-6).

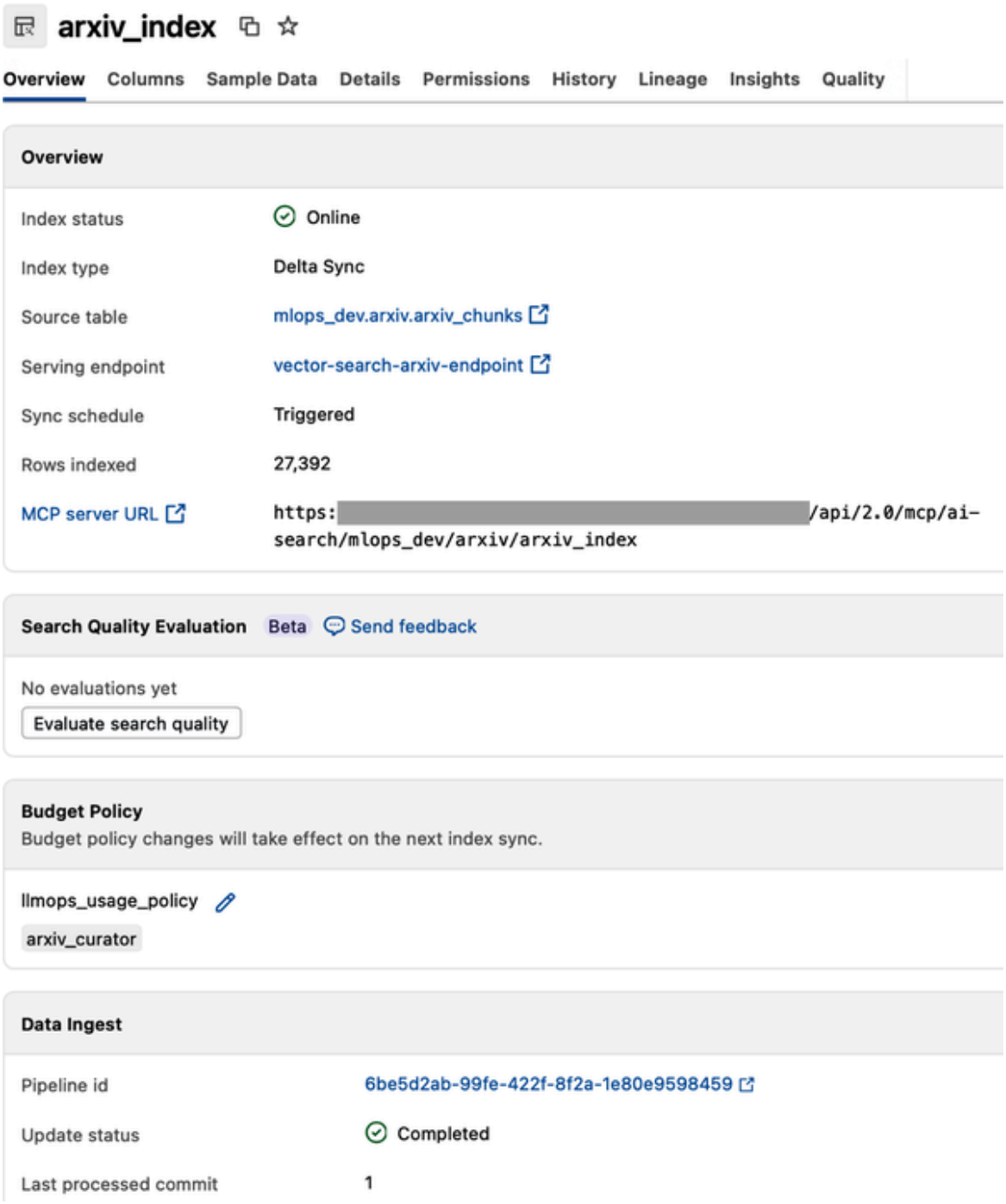


Figure 7-6. Databricks AI Search index view in the Catalog Explorer UI

It's important to mention here that the `create_delta_sync_index()` command does not just create the AI Search Index but also computes the embeddings. This allows us to query the index using natural language. The same model we use to create the embeddings is automatically applied to our query to retrieve the most relevant documents.

You can also bring your own embeddings. In this case, the source Delta table should contain precomputed embeddings instead of text. While this offers more control, it also means you will be responsible for generating the corresponding embeddings within your LLM application at query time. Figure 7-7 demonstrates the differences between Databricks-managed and self-managed embedding workflows.

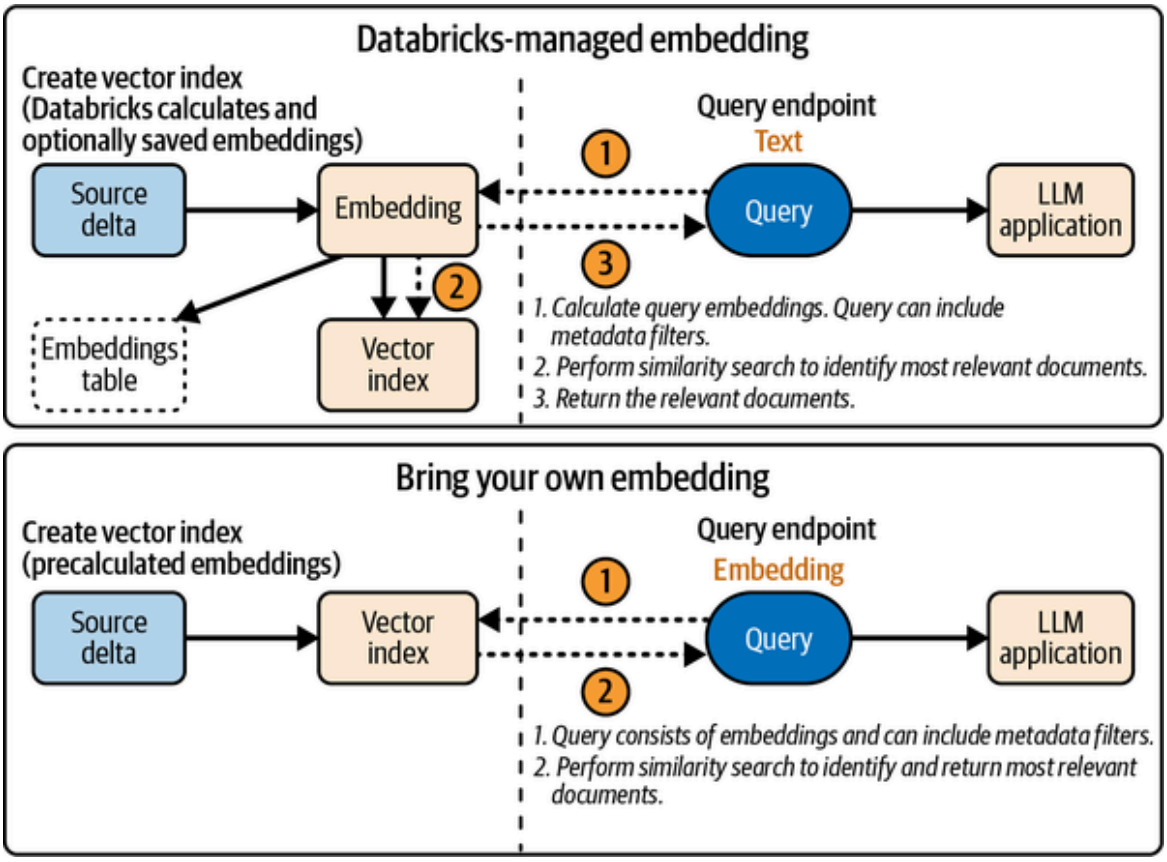


Figure 7-7. Databricks-managed versus self-managed embeddings, adapted from the Databricks documentation

Querying and retrieval quality

Databricks AI Search supports natural language queries. For the `similarity_search()` function, we need to provide the following arguments:

columns

These are the fields to return in the results.

query_text or query_vector

Here, we designate whether we want to make queries as a text or as embedding if we choose the “bring your own embedding” option.

filters

These are SQL-like filters that restrict results by metadata (e.g., specific years or authors). Filter operators like `=`, `>=`, `<=`, `IN`, `LIKE`, `NOT`, and `OR` are supported. Syntax varies by endpoint type.[13]

num_results

This is the number of chunks to retrieve.

score_threshold

This is where we define a minimum similarity score for results. Any document with a score below this threshold will be excluded, even if it ranks within the top *K* results.

query_type (ANN or HYBRID)

Databricks AI Search uses the HNSW algorithm for its approximate nearest neighbor (ANN) searches and the L2 distance metric to measure embedding vector similarity.[14] The hybrid keyword-similarity option matches exact words in the query while also using a vector-based similarity search to capture the semantic relationships and context of the query. This method is particularly useful in RAG applications where source data has unique keywords that are not well suited to pure similarity search.

reranker

> *Reranking* is a technique that improves retrieval quality by evaluating the retrieved documents to identify the ones that are semantically most relevant. Databricks has developed a reranker that you can use right away, and you can also use it to specify columns containing additional context that is relevant for reranking, such as `paper.summary` and `title`.

The function also accepts operational arguments that control its runtime behavior: how verbose its logging is (the debugging level), how many times it retries on failure, the backoff factor between those retries, and whether warnings will be suppressed.

This is how we can use the `similarity_search()` function:

```
from databricks.vector_search.reranker import DatabricksReranker

results = index.similarity_search(
   query_text="Chunking strategies for document processing",
   columns=["text", "id"],
   filters={"year": "2026"},
   num_results=5,
   query_type = "hybrid",
   reranker=DatabricksReranker(columns_to_rerank=["text", "title", "summary"])
   )
```

Optimizing search quality

To help you understand the impact of configuration changes, we need to establish evaluation metrics and a test dataset.

For a test dataset, you can use existing labeled query-answer pairs, generate synthetic test cases from your documents, or use LLM judges to assess quality without labels. Even a small synthetic dataset of 50 queries is enough to tell you whether hybrid search improves quality or whether reranking helps.

When it comes to evaluation metrics, you should pick them based on what failure looks like for your application. Once you have metrics aligned with

your use case, you can systematically test configurations and determine which changes actually improve retrieval quality.

Key metrics include the following:

Recall@K

> This measures how many relevant items were successfully retrieved. It's critical for legal and financial use cases.

Precision@K

> This measures what percentage of retrieved items are actually relevant. It's critical for technical support where only the best result matters.

F1-score@K

> This balances precision and recall.

The Databricks guide on vector search retrieval quality outlines eight main steps AI engineers should consider to improve quality:[15]

1. *Enable hybrid search*. One-line code change may improve recall by catching both semantic and keyword matches and improve precision for queries with specific terms.
2. *Use metadata filtering*. This reduces the search space, improves recall within the filtered subset, and improves precision by eliminating irrelevant results.
3. *Add reranking*. Databricks' built-in reranker is a great solution, but it is not suitable for high-QPS applications (those with > 5 QPS without additional scaling) or applications with low latency requirements. In those scenarios, it may be a good idea to build a custom reranker that utilizes a Model Serving endpoint.
4. *Improve data preparation.* Refine document parsing and chunking strategies.

5. *Optimize queries*. Generate multiple query variations to improve recall.
6. *Optimize prompts*. Use automatic prompt optimization techniques to improve prompts that are used for data preparation or query rewriting.
7. *Implement adaptive retrieval strategies*. Build agents that can reformulate queries or decide if retrieval is necessary.
8. *Fine-tune embedding models***.** This is a final option if standard models do not provide satisfactory results, particularly for non-English languages.

So far, we've created an AI Search endpoint, built an AI Search index, and tested querying it. While vector search is perfect for retrieving relevant context based on similarity, our agent also needs tools for analytical tasks, such as aggregations and statistical queries. For those requirements, Databricks Genie is a great solution.

Genie Agent (Formerly, Genie Space)

Genie is a Databricks tool that lets business users get data insights by asking questions in natural language. It translates a question into a SQL query, runs it on a SQL warehouse, and returns an answer. Rather than relying on a single large model, Genie uses several specialized components to handle the steps from question to result.

How well Genie performs depends almost entirely on the context you give it. It draws on table schemas, column names and their descriptions, example SQL queries, custom instructions, and conversation history.

When that context is thin, or when Genie is pointed at large, raw, uncurated tables, accuracy suffers. This is because the query space is wide, column meanings are ambiguous, and the model has more room to generate the wrong query. The fix is to design that context deliberately by doing the following:

Curate the datasets you expose

> Genie chooses which tables and columns to use for each question, so a wide surface means more chances to join the wrong tables or pick a misleading column. Keep it scoped to the tables that actually answer your users' questions.

Aggregate upfront

> Against raw data, Genie has to reconstruct the grouping, time windows, and deduplication for every question, and small variations produce different numbers. Summary tables encode that logic once, so answers stay consistent.

Guide Genie with instructions and example SQL queries

> Examples show Genie the join paths, filters, and column choices you treat as correct, so Genie will reuse them instead of improvising. Instructions cover what an example cannot be, such as which records to exclude or what specific metrics mean in your business.

Genie was originally built for people working through a UI, but it is now generally available through the Conversation API, which makes it usable inside AI agents and applications.

Databricks AI Search versus Genie

Databricks AI Search and Genie serve complementary purposes. To determine which one would be the best fit for a specific task, consider how the nature of the query dictates the tool usage by asking yourself these questions:

What are recent approaches to efficient attention mechanisms?

> Databricks AI Search is the best fit here, as it retrieves semantically similar paper chunks based on meaning rather than structured counts.

How many papers were published in each category in 2026?

> Genie is ideal for this as it translates this to SQL against the metadata table.

Who are the most prolific authors in our collection?

> Genie can easily count and rank authors by querying the structured metadata.

For the arXiv pipeline, we will create a Genie agent and grant it access to the metadata table. Although it is possible to connect Genie directly to an AI Search index, keeping these components separate gives us finer control over how to use tools.

Provisioning the SQL warehouse

Before creating a Genie agent, we must provision a SQL warehouse, as Genie relies on a warehouse to query tables. We will create this warehouse using the Databricks SDK. In this example, we deliberately choose the smallest serverless warehouse size, configure auto-termination after five minutes of inactivity, and apply a usage policy so that costs can be tracked and allocated later.

While we used DABs to create a warehouse in Chapter 6, we are using the SDK here for a specific reason: at the time of writing this book, creating a Genie agent was not yet supported via DABs. To keep our provisioning logic consistent within this workflow, we'll manage both the warehouse and the Genie agent through the SDK:

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service import sql
from databricks.sdk.service.sql import CreateWarehouseRequestWarehouseType

w = WorkspaceClient()

created = w.warehouses.create(
   name="2XS_arxiv_warehouse",
   cluster_size="2X-Small",
```

```
    max_num_clusters=1,
    auto_stop_mins=5,
    warehouse_type=CreateWarehouseRequestWarehouseType("PRO"),
    enable_serverless_compute=True,
    tags=sql.EndpointTags(
        custom_tags=[sql.EndpointTagPair(key="project",
                                         value="arxiv_curator")])
    ,
).result()

warehouse_id=created.id
```

Creating the Genie agent

We can create a Genie agent via the UI, API, or Databricks SDK. Below is an example of how to create a Genie agent using SDK, with access to only one table (`arxiv_papers`) under the defined catalog and schema:

```
import json

from arxiv_curator.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")
catalog = cfg.catalog
schema = cfg.schema


serialized_space = {
    "version": 1,
    "data_sources": {
        "tables": [
            {
                "identifier": f"{catalog}.{schema}.arxiv_papers",
                "column_configs": [
                    {"column_name": "authors"},
                    {"column_name": "ingest_ts", "get_example_values": True},
                    {"column_name": "paper_id", "get_example_values": True},
                    {
                        "column_name": "pdf_url",
                        "get_example_values": True,
                        "build_value_dictionary": True,
                    },
                    {"column_name": "processed", "get_example_values": True},
                    {"column_name": "published", "get_example_values": True},
                    {
                        "column_name": "summary",
```

```
                        "get_example_values": True,
                        "build_value_dictionary": True,
                    },
                    {
                        "column_name": "title",
                        "get_example_values": True,
                        "build_value_dictionary": True,
                    },
                    {
                        "column_name": "volume_path",
                        "get_example_values": True,
                        "build_value_dictionary": True,
                    },
                ],
            }
        ]
    },
}

space = w.genie.create_space(
    warehouse_id=warehouse_id,
    serialized_space=json.dumps(serialized_space),
    title="arxiv-curator-space",
)
```

We can retrieve the serialized space definition for an existing Genie agent by using the `get_space()` function:

```
space_id = space.space_id

space = w.genie.get_space(space_id=space_id, include_serialized_space=True)
json.loads(space.serialized_space)
```

Interacting with Genie

We can interact with the Genie agent by using SDK. For example, we can start a conversation and ask it to find the last 10 published papers that are available in the metadata table:

```
conversation = w.genie.start_conversation_and_wait(
    space_id=space.space_id,
    content="Find the last 10 papers published")
```

We can then inspect what the returned object looks like (see Figure 7-8) and notice that Genie translated the original question to a SQL query.

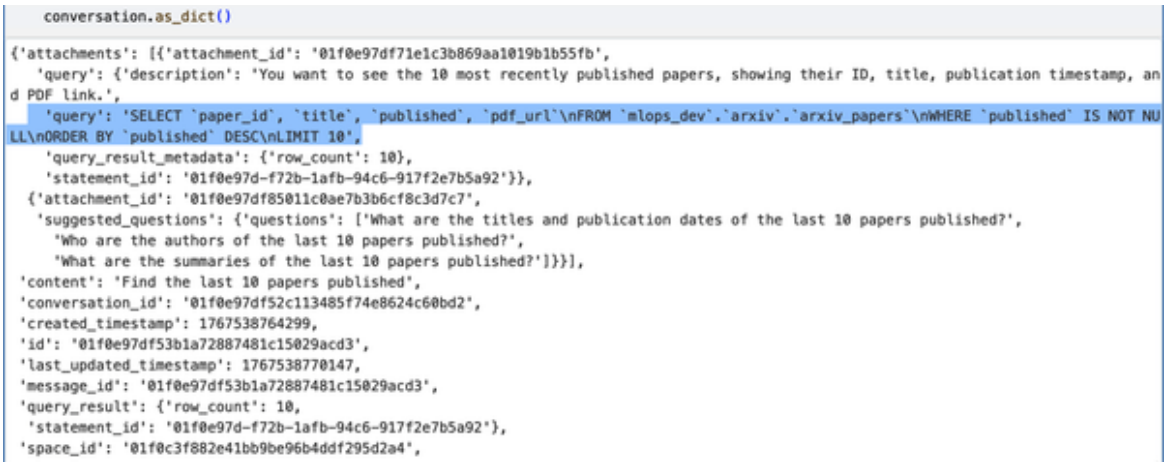


Figure 7-8. Genie conversation output

Now, let's ask a question within the same conversation (Genie keeps the message history within the conversation) and request to return the list of authors of the last 10 publications:

```
message = w.genie.create_message_and_wait(
    space_id=space.space_id,
    conversation_id=conversation.conversation_id,
    content="Return the list of authors of the last 10 papers published")
```

We'll see in the output that Genie was able to successfully answer the question (see Figure 7-9).

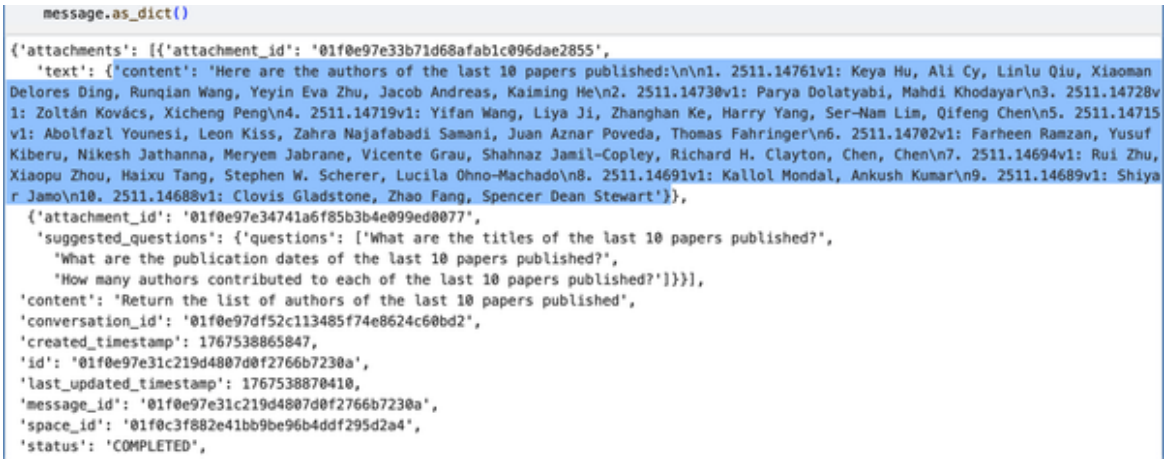


Figure 7-9. Genie message output

Having established our AI Search index for semantic retrieval and our Genie agent for analytical queries, we now have the two foundational *sources of context* for our agent.

The next challenge is exposing these capabilities to the agent in a standardized, scalable way. To do this, we will utilize Databricks-managed MCP servers. This approach allows the agent to interact with both structured and unstructured data through a unified protocol, which we will explore in the following section.

MCP Servers on Databricks and Tool Calling

The Model Context Protocol (MCP) is an open standard that was released by Anthropic in November 2024 to solve a fundamental problem: how to allow AI agents access to the data and tools they need in a standardized way without requiring custom integrations every time.[16]

Since MCP's release, its adoption has grown fast, and MCP has become a standard for AI development. Almost any tool you can think of has MCP integration today: Hugging Face, GitHub, Postman, and many more.

Managed MCP ecosystems on Databricks

You may have noticed that both the vector search index and the Genie agent have appeared under MCP servers (see Figure 7-10) without any configuration from our side, alongside the Unity Catalog functions we created in Chapter 4. That's because all these functionalities are treated as Databricks-managed MCP by default.

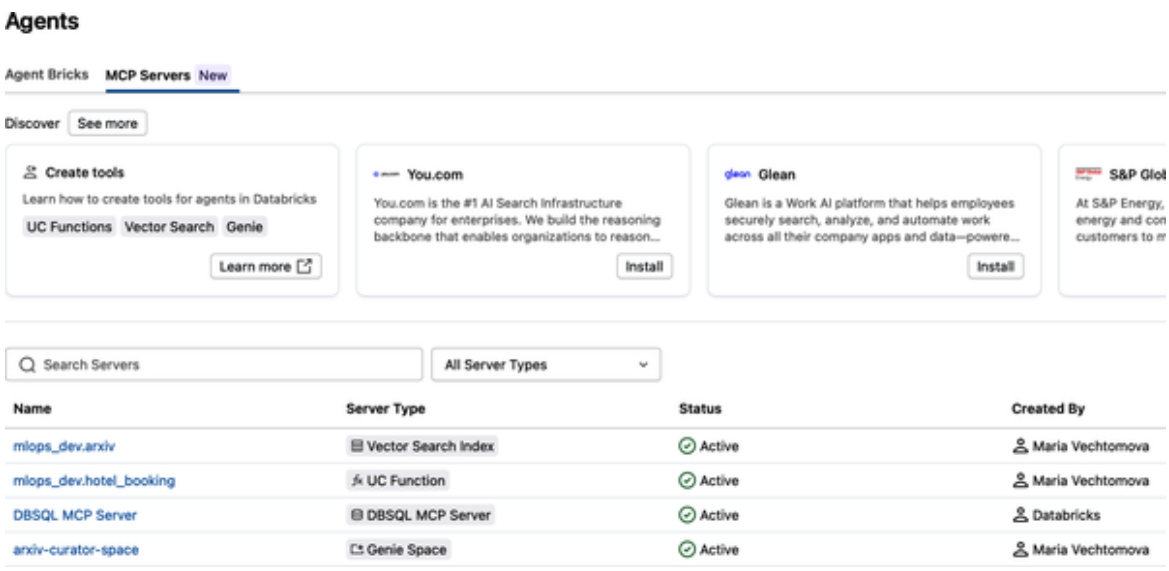


Figure 7-10. Managed MCP servers on Databricks

Table 7-1 outlines the available managed servers.

Table 7-1. Overview of Databricks-managed MCP servers

MCP server	URL pattern	Description	Available tools and parameters
AI Search index	`https://{host}/api/2.0/mcp/vector-search/{catalog}/{schema}`	This queries an AI Search index to find relevant chunks.	`{catalog}__{schema}__{index}: query (str)`
Genie agent	`https://{host}/api/2.0/mcp/genie/{genie_space_id}`	This queries Genie agent (formerly, Genie space).	`query_space_{space_id}: query (str), conversation_id (str); poll_response_{space_id}: conversation_id (str); message_id (str)`
Unity Catalog functions	`https://{host}/api/2.0/mcp/functions/{catalog}/{schema}`	This invokes a unity catalog function.	`{catalog}__{schema}__{fn}`; parameters are defined by users when functions get created.
DBSQL	`https://{host}/api/2.0/mcp/sql`	This runs AI-generated SQL.	`execute_sql: query (str); execute_sql_read_only: query(str); poll_sql_result: statement_id (str)`

One of the known limitations for AI Search index is that only indexes with Databricks managed embeddings are supported. Extra parameters such as query type, the reranker, and filters are not supported out of the box.

We can also set up an external MCP server on Databricks via Databricks Marketplace or HTTP connection, create a governance layer on top, and have it handle authentication and access.[17]

Implementing tool calling

To use the MCP servers on Databricks, we first need a consistent way to represent a tool to the agent. The following `ToolInfo` model holds the three things the agent needs for each tool: its `name`, the `spec` that describes the tool to the model, and the `exec_fn` that runs it when the model decides to call it:

```
from collections.abc import Callable
from pydantic import BaseModel

class ToolInfo(BaseModel):
   name: str
   spec: dict
   exec_fn: Callable
```

We can call tools we define in Databricks MCPs by using the `DatabricksMCPClient`. To initialize it, we need to provide a server URL and a `WorkspaceClient`. Here, we create a `create_managed_exec_fn()` function, which is a wrapper around tool calling:

```
from databricks.sdk import WorkspaceClient
from databricks_mcp import DatabricksMCPClient

def create_managed_exec_fn(
   server_url: str, tool_name: str, w: WorkspaceClient
) -> Callable:
   def exec_fn(**kwargs):
       client = DatabricksMCPClient(server_url=server_url, workspace_client=w)
       response = client.call_tool(tool_name, kwargs)
       return "".join([c.text for c in response.content])
   return exec_fn
```

Finally, we implement the `create_mcp_tools()` function, which takes a list of MCP server URLs and a WorkspaceClient as arguments and returns the list of `ToolInfo` objects we defined earlier. We must declare the function as async because `DatabricksMCPClient` exposes an asynchronous API. MCP clients are asynchronous by design because the problems they solve—like tool use, context gathering, and model coordination—are inherently I/O bound, multistep, and latency sensitive. Asynchronicity isn't an implementation detail; it's what makes MCP practical at scale:

```
async def create_mcp_tools(w: WorkspaceClient, url_list: list[str]) ->
list[ToolInfo]:
   tools = []
   for server_url in url_list:
       mcp_client = DatabricksMCPClient(server_url=server_url,
workspace_client=w)
       mcp_tools = mcp_client.list_tools()
       for mcp_tool in mcp_tools:
           input_schema = mcp_tool.inputSchema.copy() if mcp_tool.inputSchema
else {}
           tool_spec = {
               "type": "function",
               "function": {
                   "name": mcp_tool.name,
                   "parameters": input_schema,
                   "description": mcp_tool.description or f"Tool:
{mcp_tool.name}",
               },
           }
           exec_fn = create_managed_exec_fn(server_url, mcp_tool.name, w)
           tools.append(ToolInfo(name=mcp_tool.name, spec=tool_spec,
exec_fn=exec_fn))
   return tools
```

This code is defined under the `mcp` module of the `arxiv_curator` package. We'll use the `create_mcp_tools()` function later in Chapter 8.

We've covered how agents access external data and how MCP servers expose tools in a standardized way. But there's a third piece: maintaining state across turns within a session.

Agent Memory with Lakebase

When you deploy an agent behind a serving endpoint, you decouple the agent from its caller. Each request arrives independently, and the agent has no built-in way to know this is turn five of an ongoing conversation. Yet the user expects continuity. They ask about their recent order and then follow up with “Can you change the shipping address?” The agent needs to know which order they mean.

Session persistence solves this problem. The caller passes a session identifier with each request, and the agent uses it to retrieve and update the conversation state. This keeps the agent stateless at the infrastructure level while maintaining conversational context where it matters.

This is where Lakebase comes in. We already used it in Chapter 4 for Feature Serving and Model Serving with feature lookup, but it’s not limited to ML workflows. Lakebase is a fully managed Postgres database, so it’s a natural fit for storing agent session state.

The core pattern is straightforward: you store messages and retrieve them by session ID. From there, you can evolve the design. For example, instead of replaying the full conversation every time, you can introduce a *summarization step* that periodically compresses earlier turns into a shorter representation. This reduces token usage, improves latency, and keeps the agent focused on the most relevant context while still preserving continuity.

For simplicity’s sake, we’ll leave summarization out of scope for this book. Instead, let’s focus on the basic pattern: storing messages and retrieving them by session ID.

Lakebase Autoscaling for agent workloads

In Chapter 4, we introduced Lakebase for Feature Serving and created a Lakebase project using the Feature Engineering client. However, we did not explore Lakebase Autoscaling’s capabilities in detail.

Its key features include the following:

Autoscaling

Compute resources scale dynamically with demand, so you don’t need to provision for peak traffic in advance.

Scaling to zero

Inactive compute scales to 0 after timeout to minimize costs.

Branches

These allow you to create isolated database branches for development, testing, and experimentation.

Read replicas

You can use dedicated read-only replicas for read queries. By separating reads from writes, you reduce load on the primary database and improve overall throughput and latency.

Instant restore

You can create a new branch from any point in time within your restore window, so it’s easier to recover data or debug issues.

Lakebase Autoscaling project structure

Understanding the Lakebase Autoscaling project structure helps you organize and manage your resources effectively. A *project* is the top-level organizational unit in Lakebase Autoscaling, and it acts as the primary container for all your resources. If you’ve worked with Lakebase Provisioned before, you can think of a project as playing a role similar to that of an instance.

A project groups together your databases, branches, computes, and related configuration. It also defines defaults such as compute settings, restore windows, and update policies that apply across all branches within the project.

At the top level, a project contains one or more branches. Each branch is isolated and includes its own compute, roles, and databases. This structure allows you to manage environments cleanly, experiment without impacting production, and scale workloads independently within a single project:

```
Project
└── Branches (main, development, staging, etc.)
      ├── Computes (endpoints)
      ├── Roles (Postgres roles)
      └── Databases (Postgres databases)
```

Creating a Lakebase Autoscaling project

We'll start by creating a Lakebase Autoscaling project programmatically, using the Postgres API, which is part of the Databricks SDK. This approach allows you to define your IaC and integrate project creation directly into your workflows.

In this example, we configure a new project with autoscaling settings, including minimum and maximum compute capacity and an automatic suspend timeout to enable scale-to-zero behavior. This ensures the system can scale with demand while avoiding unnecessary costs when idle. Here, we also associate the project with a usage policy for cost tracking:

```
import json
import urllib.parse
from uuid import uuid4

import psycopg
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.postgres import (
   PostgresAPI,
   Project,
   ProjectDefaultEndpointSettings,
   ProjectSpec,
)
from google.protobuf.duration_pb2 import Duration
from loguru import logger

from arxiv_curator.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")
```

```
w = WorkspaceClient()
pg_api = PostgresAPI(w.api_client)

project_id = "arxiv-agent-lakebase"

try:
   project = pg_api.get_project(name=f"projects/{project_id}")
except Exception:
   project = pg_api.create_project(
       project_id=project_id,
       project=Project(
           spec=ProjectSpec(
               display_name=project_id,
               usage_policy_id=cfg.usage_policy_id,
               default_endpoint_settings=ProjectDefaultEndpointSettings(
                   autoscaling_limit_min_cu=1,
                   autoscaling_limit_max_cu=4,
                   suspend_timeout_duration=Duration(seconds=300),
               ),
           ),
       ),
   ).wait()
```

Now that we have set up the Lakebase project, let's create a connection, define a table, and try to insert and read data.

Interacting with Lakebase

To create a connection, we need a host. Due to Lakebase's multilevel structure, we first retrieve the branch (which is, by default, the production branch). Each branch has an associated compute endpoint, which provides the host required to establish the connection:

```
default_branch = next(iter(pg_api.list_branches(parent=project.name)))
endpoint = next(iter(pg_api.list_endpoints(parent=default_branch.name)))
host = endpoint.status.hosts.host
```

We then can use the Postgres API again to generate the database credentials and construct the connection string:

```
user = w.current_user.me()
pg_credential = pg_api.generate_database_credential(endpoint=endpoint.name)
```

```
username = urllib.parse.quote_plus(user.user_name)
conn_string = (
    f"postgresql://{username}:{pg_credential.token}@{host}:5432/"
    "databricks_postgres?sslmode=require"
)
```

Because Lakebase is a Postgres database, we can work with it using standard Python libraries. We use psycopg3, the modern PostgreSQL adapter for Python, to open a connection, and then, we use that connection to create a table for storing session-based conversation messages:

```
import psycopg

with psycopg.connect(conn_string) as conn:
    # Create session_messages table
    conn.execute("""
        CREATE TABLE IF NOT EXISTS session_messages (
            id SERIAL PRIMARY KEY,
            session_id TEXT NOT NULL,
            message_data JSONB NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    """)
    conn.execute("""
        CREATE INDEX IF NOT EXISTS idx_session_messages_session_id
        ON session_messages(session_id)
    """)
```

After we create the table, we can insert messages into it:

```
test_session_id = f"test-session-{uuid4()}"
test_messages = [
    {"role": "user", "content": "Hello, what can you help me with?"},
    {"role": "assistant", "content": "I can help you find research papers."},
    {"role": "user", "content": "Find papers about LLM reasoning"},
]

with psycopg.connect(conn_string) as conn:
    for msg in test_messages:
        conn.execute(
            "INSERT INTO session_messages (session_id, message_data) "
            "VALUES (%s, %s)",
            (test_session_id, json.dumps(msg)),
        )
```

And then, we can successfully read them back:

```
with psycopg.connect(conn_string) as conn:
    result = conn.execute(
        """
        SELECT message_data, created_at FROM session_messages
        WHERE session_id = %s
        ORDER BY created_at ASC
        """,
        (test_session_id,),
    ).fetchall()

    logger.info(f"Loaded {len(result)} messages:")
    for row in result:
        logger.info(f"  [{row[1]}] {row[0]}")
```

The agent can use this mechanism to load the messages from Lakebase when it receives a request and store messages before it generates output.

So far, we have worked with a single database connection. That is fine for demonstration purposes, but it does not scale. In a real application and especially in an agent system that handles concurrent requests, opening and closing a connection for every request adds unnecessary overhead. We solve this with a connection pool.

Utilizing a connection pool

A *connection pool* maintains a set of reusable database connections. Instead of opening a new connection each time, requests borrow a connection from the pool and return it when done. This reduces latency, improves throughput, and allows the system to handle multiple requests more efficiently.

In the memory module of our custom package, we define a `LakebaseMemory` class that uses a connection pool to interact with Lakebase and manage session persistence. Let’s walk through the implementation step by step.

The class takes a host and instance name during initialization, but it doesn’t immediately establish connections. Instead, the connection pool is created

lazily, only when it's first needed:

```
import json
import os
import urllib.parse
from typing import Any

import psycopg
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.postgres import PostgresAPI
from loguru import logger
from psycopg_pool import ConnectionPool


class LakebaseMemory:
    """Handles session message persistence using Lakebase (PostgreSQL)."""

    def __init__(
        self,
        project_id: str,
    ) -> None:
        self.project_id = project_id
        self._pool: ConnectionPool | None = None
```

Connecting to Lakebase requires exchanging Databricks credentials for a short-lived database token. This class supports two authentication modes: a *service principal* for production use and *user credentials* for local development and testing.

In production, we must add the service principal to the Lakebase project and grant it the necessary permissions to read from and write to the database:

```
def _get_connection_string(self) -> str:
    """Build connection string for Lakebase.

    Supports two authentication modes:
    - SPN (production): Needs LAKEBASE_SP_CLIENT_ID,
      LAKEBASE_SP_CLIENT_SECRET, LAKEBASE_SP_HOST
    - User (local testing): Uses default WorkspaceClient auth
    """
    # Use dedicated Lakebase SPN env vars to avoid overriding
    # the default WorkspaceClient auth used by MCP tools
```

```
    client_id = os.environ.get("LAKEBASE_SP_CLIENT_ID")
    client_secret = os.environ.get("LAKEBASE_SP_CLIENT_SECRET")
    host = os.environ.get("LAKEBASE_SP_HOST")

    if client_id and client_secret and host:
        w = WorkspaceClient(
            host=host,
            client_id=client_id,
            client_secret=client_secret,
        )
    else:
        w = WorkspaceClient()

    pg_api = PostgresAPI(w.api_client)
```

Notice that our implementation uses *Lakebase-specific environment variables* (`LAKEBASE_SP_CLIENT_ID`, `LAKEBASE_SP_CLIENT_SECRET`, and `LAKEBASE_SP_HOST`) instead of the standard Databricks variables (`DATABRICKS_HOST`, `DATABRICKS_CLIENT_ID`, and `DATABRICKS_CLIENT_SECRET`). This avoids conflicts with other components, particularly MCP resources that rely on the default authentication chain. We'll revisit this distinction later, when logging and registering the agent.

The connection pool is created on the first use with the `_get_pool()` function. Since database tokens are short lived, connections can fail when credentials expire. The `_reset_pool()` method handles this. If we hit an `OperationalError`, we close the pool so the next request gets fresh tokens:

```
def _get_pool(self) -> ConnectionPool:
    """Get or create connection pool."""
    if self._pool is None:
        conn_string = self._get_connection_string()
        self._pool = ConnectionPool(conninfo=conn_string, min_size=1,
max_size=5)
    return self._pool

def _reset_pool(self) -> None:
    """Reset pool to force new credentials on next use."""
    if self._pool is not None:
```

```
            self._pool.close()
            self._pool = None
```

The core functionality is straightforward: we load and save messages by session ID. When loading, we retrieve all messages for a session ordered by timestamp:

```
    def load_messages(self, session_id: str) -> list[dict[str, Any]]:
        """Load previous messages for a session."""
        try:
            with self._get_pool().connection() as conn:
                self._ensure_messages_table(conn)
                result = conn.execute(
                    """
                    SELECT message_data FROM session_messages
                    WHERE session_id = %s
                    ORDER BY created_at ASC
                    """,
                    (session_id,),
                ).fetchall()
                return [row[0] for row in result]
        except psycopg.OperationalError:
            self._reset_pool()
            raise
        except Exception as e:
            logger.warning(f"Failed to load session messages: {e}")
            return []
```

When saving, we append new messages to the session:

```
    def save_messages(
        self, session_id: str, messages: list[dict[str, Any]]
    ) -> None:
        """Append messages to a session."""
        try:
            with self._get_pool().connection() as conn:
                self._ensure_messages_table(conn)
                for msg in messages:
                    conn.execute(
                        "INSERT INTO session_messages (session_id, message_data) "
                        "VALUES (%s, %s)",
                        (session_id, json.dumps(msg)),
```

```
                )
            except psycopg.OperationalError:
                self._reset_pool()
                raise
            except Exception as e:
                logger.warning(f"Failed to save session messages: {e}")
```

We'll use the `LakebaseMemory` class when defining the agent in Chapter 8.

Conclusion

In this chapter, we built up the context an agent needs, step by step. We began with the different ways to use foundation models on Databricks. We then processed raw arXiv PDFs into parsed, chunked text and indexed it in Databricks AI Search so it could be retrieved, and we set up a Genie agent for natural-language queries over structured data. Finally, we saw how both Databricks AI Search and Genie can be exposed to an agent as tools through MCP.

So far, we have assembled the context and tools an agent can draw on, but not the agent itself. We have not yet defined its logic: how it decides which tool to call, when to retrieve, and how it sequences those steps into a response. That logic is the subject of the next chapter, along with two pieces that make the agent operational: MLflow flavors for agents, which give us a standard way to package and serve it, and MLflow tracing, which lets us see what the agent did on each request.

---

1 You can find the definition in the OpenAI SDK guides.

2 See the Databricks documentation on AI functions.

3 Consult your cloud-specific documentation for more details. For example, see the AWS documentation.

4 You can find supported models in the documentation.

5 For further details, consult the community discussion.

6 For further details, consult the documentation on benchmarking.

7 For further details, consult the documentation on networking.

8 Databricks solution architects Sara Dooley and Austin Zaccor wrote a blog that explains how to achieve the Custom Provider solution.

9 Notice that we refer to model families rather than specific versions. Individual releases change quickly, and many are retired within months, so for the current options on Databricks, consult the foundation model overview documentation.

10 Note that we use the `time.sleep(3)` command to suspend the execution for three seconds to avoid reaching API limits. For more information, see *https://oreil.ly/Na6hZ*.

11 For details on AI Search, consult the Databricks documentation.

12 Consult Vector Search Python SDK documentation for exact details.

13 For more details, consult the documentation on querying the AI Search index.

14 This blog by Redis explains how this algorithm can improve search.

15 For more information, consult the Databricks documentation.

16 Anthropic, “Introducing the Model Context Protocol,” November 25, 2024, *https://oreil.ly/Hh0UK*.

17 For more information, consult the Databricks documentation on external MCPs.

Chapter 8. MLflow for GenAI

MLflow for GenAI can be thought of as a *distinct layer* on top of the MLflow concepts we've seen so far. While familiar building blocks still apply, such as experiment tracking for logging runs and registering models, the application of these features shifts significantly in the context of generative AI.

In GenAI workflows, we still log and register models, but we rely on the PyFunc flavor with a different abstraction. Instead of extending *PythonModel*, GenAI applications typically use classes such as *ResponsesAgent*, which are designed to represent agents rather than traditional predictive models.

Evaluation is also fundamentally different. Rather than focusing on standard ML metrics, GenAI evaluation centers on behavioral qualities and agent decisions. To support this, MLflow introduces *traces* to provide fine-grained observability into everything that happens inside an agent, including tool calls, intermediate steps, and execution flow in real time.

In the following sections, we'll explore MLflow's GenAI flavors and the fundamentals of MLflow tracing. From there, we'll define a custom agent with tracing enabled and progressively build toward evaluation, logging, and registering the agent.

MLflow Flavors

In Chapter 3, we explored logging and registering ML models using MLflow, and we discussed different built-in flavors like scikit-learn (sklearn), LightGBM, and others. In the same way, MLflow provides specialized flavors for agentic frameworks, including DSPy, LangChain, and LlamaIndex. These libraries simplify the orchestration of AI agents by

providing high-level abstractions for components like tool use, memory, and LLM calling.

For even greater control, you can define your own custom orchestration logic and utilize `mlflow.pyfunc.ResponsesAgent`, which follows OpenAI Responses API. This approach gives you more flexibility than using built-in flavors, and it can be great for fast prototyping. Figure 8-1 summarizes these various flavors and illustrates the expected payload format when these models are deployed to a Mosaic AI Model Serving endpoint.

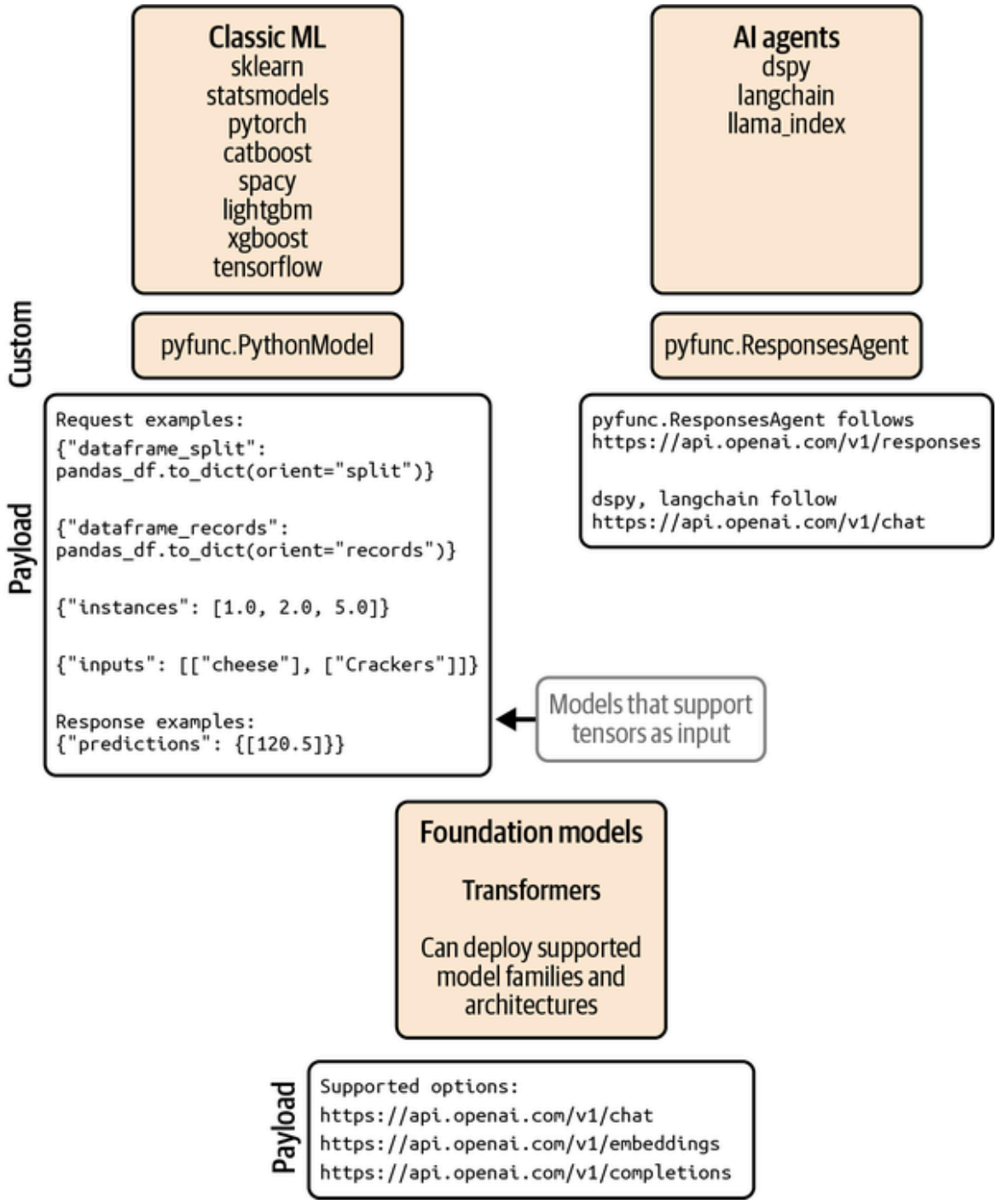


Figure 8-1. Model flavors and payload

In this chapter, we'll focus on building a custom agent. However, before we jump in, let's take a step back and look at the theory behind MLflow tracing, since we'll rely on the *@mlflow.trace* decorator when building the agent.

MLflow Tracing

MLflow tracing gives you visibility into what your LLM application is actually doing under the hood. It's an OpenTelemetry-compatible observability layer that records execution traces for every request. By capturing inputs, outputs, and metadata at each intermediate step (including model calls, tool invocations, and control-flow decisions), it transforms the "black box" of an agent into a transparent process.

With tracing enabled, you're no longer flying blind. Instead of receiving an incorrect final answer and guessing what went wrong, you can see the exact path the agent took to reach that response. This makes it much easier to debug failures, troubleshoot unexpected behavior, and assess performance as your application inevitably changes.

Let's examine the specific components that make up an MLflow trace.

Anatomy of a Trace

At a high level, a trace is conceptually similar to an MLflow run (which we discussed in Chapter 3). Both belong to an MLflow experiment. The difference is that traditional experiments track model training cycles, whereas GenAI experiments use traces to capture live or simulated agent executions.

Each trace consists of two main parts: trace info and trace data (see Figure 8-2).

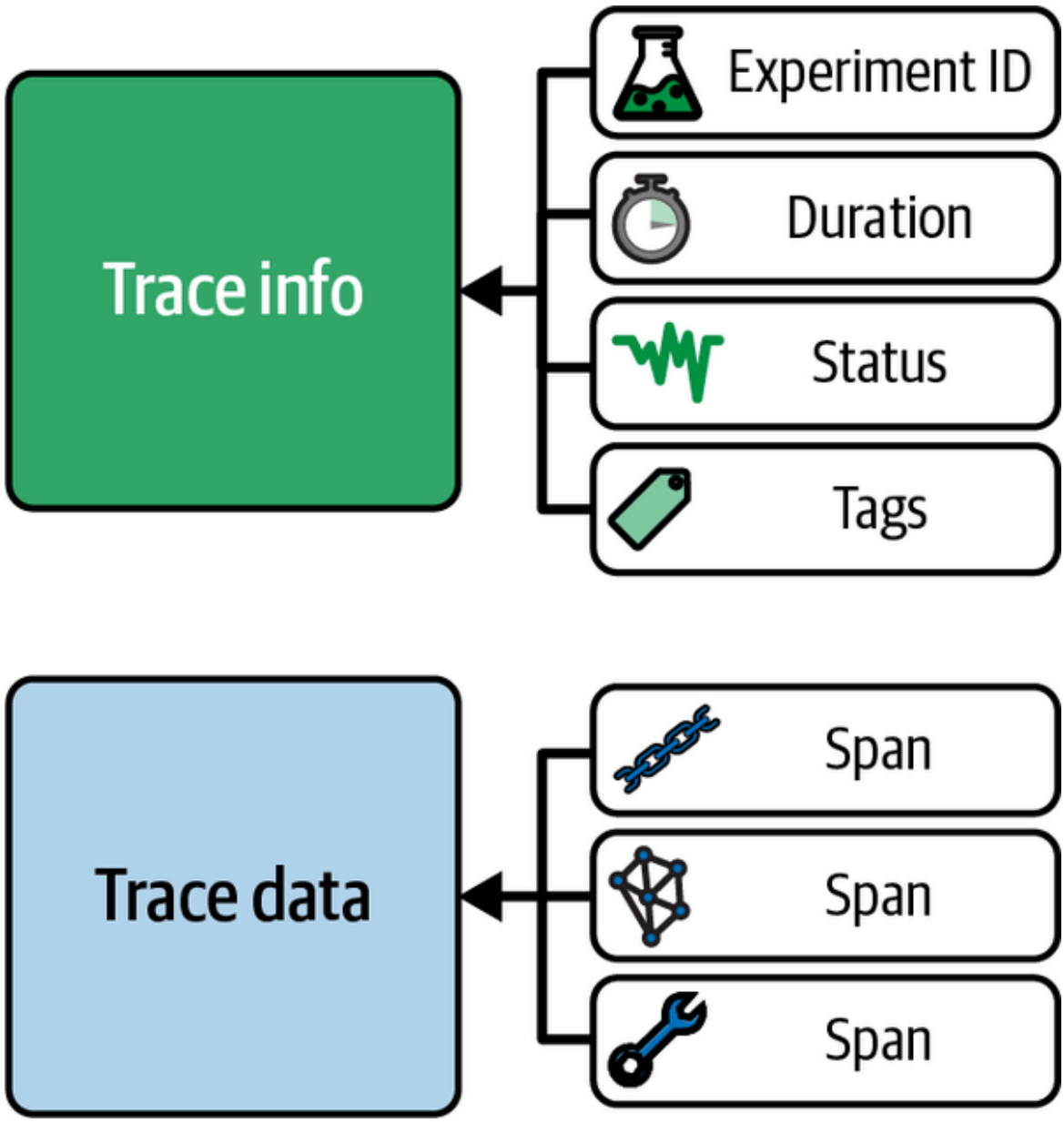


Figure 8-2. Trace info and trace data, adapted from MLflow documentation

Trace info contains high-level metadata about an execution. This includes the experiment ID, the trace status (for example, success or failure), and the total execution latency. It also supports *tags*, which add contextual information such as the user, session ID, and developer-defined key-value pairs. These tags are particularly useful for searching, filtering, and organizing traces during analysis, monitoring, and debugging.

Trace data stores the actual execution payload. It contains the instrumented *span objects* that capture the step-by-step behavior of your application from

input to output. Spans record critical operations such as LLM invocations, tool calls, and intermediate processing steps, to provide a detailed, end-to-end view of how a GenAI application executes in practice.

MLflow's span object is designed to be fully compatible with the OpenTelemetry span specification. It is implemented as a dataclass, and it closely mirrors the OpenTelemetry span model, while adding convenience accessors and helper methods that are tailored specifically to GenAI workloads. Each span records essential attributes such as inputs and outputs, start and end times, the execution status, the span ID, the trace ID, and an optional parent ID for hierarchical traces.

MLflow supports a rich set of span types that record common operations within GenAI applications, including the following:

LLM

: This records language model inference.

TOOL

: This records the execution of external tools, functions, and APIs.

RETRIEVER

: This records the retrieval of relevant context, for example, from an AI search index.

MEMORY

: This records instances of reading from or writing to conversational memory.

EMBEDDING

: This records the generation of vector embeddings from text or other inputs.

RERANKER

> This records the reranking of retrieved results based on relevance.

As with traditional ML experiment tracking, MLflow can automatically log traces. However, autologging is not always sufficient if you want to attach custom metadata and domain-specific context. To gain full control, you must use the MLflow tracing APIs.

MLflow Tracing APIs

The simplest way to trace a function is by using the `@mlflow.trace()` decorator. When we apply it, it creates a span for the duration of the decorated function and automatically captures the function's inputs and outputs. For example, the following code produces a span named `my_func` that records input arguments *x* and *y*, along with the function's return value. Note that we first set an experiment to define the location of the traces and set the tracking URI in the same way we did in Chapter 3:

```
import mlflow
from arxiv_curator.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()
mlflow.set_experiment('/Shared/tracing-demo')

@mlflow.trace
def my_func(x, y):
   return x + y
```

We can do the same with the `mlflow.start_span()` function, which manually defines the span and its scope:

```
def my_func(x, y):
   return x + y

with mlflow.start_span("my_func") as span:
   x = 1
   y = 2
   span.set_inputs({"x": x, "y": y})
```

```
    result = my_function(x, y)
    span.set_outputs({"output": result})
```

We can also add more context to a trace, even when we're using the `@mlflow.trace()` decorator. For example, we may want to attach a session ID and request ID to the trace and later pass the same information to the agent endpoint so we can link the two. We can do this by using the `mlflow.update_current_trace()` function. The following example shows how to add a session ID, Git SHA, request ID, and tags related to the Model Serving endpoint:

```
import random
from datetime import datetime

timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
session_id = f"s-{timestamp}-{random.randint(100000, 999999)}"
request_id = f"req-{timestamp}-{random.randint(100000, 999999)}"

git_sha = "abcd"

@mlflow.trace
def my_func(x, y):
    mlflow.update_current_trace(
    metadata={
        "mlflow.trace.session": session_id,
    },
    tags={"model_serving_endpoint_name": "arxiv-agent-endpoint",
          "model_version": 1,
          "git_sha": git_sha},
    client_request_id=request_id
)
    return x + y
```

We can search generated traces by using a workflow that's similar to MLflow experiment run searches. We can search traces by nearly any field. For example, the code below demonstrates how to search for traces using the `git_sha` tag:

```
mlflow.search_traces(filter_string=f"tags.git_sha = '{git_sha}'")
```

Now that we understand how to work with MLflow tracing and have already discussed MLflow flavors in the context of LLM applications, we can move on to defining a custom agent, applying the `@mlflow.trace` decorator, and passing additional context.

Defining a Custom Agent

The `arxivAgent` class inherits from the `mlflow.pyfunc.ResponsesAgent`. This base class requires us to implement prediction logic that produces `ResponsesAgentResponse` objects (or streams of response events).

Initialization and Async Handling

To initialize the class, we provide the Databricks Model Serving endpoint name, a system prompt, and the MCP tools we defined earlier. During this stage, we use the `WorkspaceClient` to retrieve an OpenAI-compatible client for Model Serving.

This setup ensures that the agent has everything it needs to orchestrate LLM calls and tool invocations. Notice that we set the MLflow experiment here and also use the `create_mcp_tools()` function we defined earlier. Here, we use the nest_asyncio package, which allows async code to run in environments where an event loop is already running. Python does not allow nested event loops, and calling `asyncio.run()` would raise an error. The `nest_asyncio.apply()` function patches the existing loop, enabling async functions like `create_mcp_tools()` to execute from synchronous code:

```
from mlflow.entities import SpanType
from mlflow.models.resources import (
   DatabricksGenieSpace,
   DatabricksServingEndpoint,
   DatabricksSQLWarehouse,
   DatabricksTable,
   DatabricksVectorSearchIndex,
)
from mlflow.pyfunc import ResponsesAgent
```

```python
from mlflow.types.responses import (
    ResponsesAgentRequest,
    ResponsesAgentResponse,
    ResponsesAgentStreamEvent,
    output_to_responses_items_stream,
    to_chat_completions_input,
)

from arxiv_curator.config import ProjectConfig
from arxiv_curator.mcp import create_mcp_tools
from arxiv_curator.memory import LakebaseMemory

# Additional imports omitted for brevity - consult the book code


class ArxivAgent(ResponsesAgent):
    def __init__(
        self,
        llm_endpoint: str,
        system_prompt: str,
        catalog: str,
        schema: str,
        genie_space_id: str | None = None,
        lakebase_project_id: str | None = None,
    ):
        """Initializes the Arxiv Agent."""
        nest_asyncio.apply()

        self.system_prompt = system_prompt
        self.llm_endpoint = llm_endpoint
        self.workspace_client = WorkspaceClient()
        self.model_serving_client = (
            self.workspace_client.serving_endpoints.get_open_ai_client()
        )

        # Initialize Lakebase memory if configured
        self.memory: LakebaseMemory | None = None
        if lakebase_project_id:
            self.memory = LakebaseMemory(
                project_id=lakebase_project_id,
            )

        # Create tools from config
        host = self.workspace_client.config.host
        tools = asyncio.run(create_mcp_tools(
            w=self.workspace_client,
            url_list=[
                f"{host}/api/2.0/mcp/vector-search/{catalog}/{schema}",
```

```
                f"{host}/api/2.0/mcp/genie/{genie_space_id}",
            ],
        ))
        self._tools_dict = {tool.name: tool for tool in tools}
```

If a Lakebase project ID is provided, the `LakebaseMemory` class gets initialized so that the agent can retrieve and store messages using a session ID.

Core Agent Logic

The `get_tool_specs()` method returns the tool specifications in the format that's expected by OpenAI-style chat completions. Each `ToolInfo` object exposes a spec that describes the tool's name, parameters, and purpose. These specifications are then passed to the LLM so that it can decide when and how to call a tool during generation:

```
def get_tool_specs(self) -> list[dict]:
    """Returns tool specifications in the format OpenAI expects."""
    return [tool_info.spec for tool_info in self._tools_dict.values()]
```

This method executes a tool by name, with the arguments provided by the model. It looks up the corresponding `ToolInfo` object and calls its execution function. (While you may notice tracing annotations here, we'll intentionally skip explaining them for now and revisit them in the next section.) Conceptually, this method forms the bridge between the LLM's structured tool calls and your application logic. You'll also notice that we use the `@mlflow.trace` decorator to trace the tool execution:

```
@mlflow.trace(span_type=SpanType.TOOL)
def execute_tool(self, tool_name: str, args: dict) -> Any:
    """Executes the specified tool with the given arguments."""
    return self._tools_dict[tool_name].exec_fn(**args)
```

The `call_llm` method invokes the LLM endpoint and streams its output, yielding partial response chunks as they arrive. It uses exponential backoff

to retry requests in case of rate-limiting errors. The method converts internal messages into the chat completions format that's expected by the OpenAI-compatible API, attaches tool specifications, and enables streaming.

For observability, we manually create an MLflow span by using `mlflow.start_span()` instead of relying on `@mlflow.trace`. This allows us to capture additional information, such as the request ID from the response headers, and explicitly set span outputs after the stream completes. We've included the request ID so you can trace which MLflow span corresponds to a specific LLM call, which makes debugging and monitoring much easier:

```
@backoff.on_exception(backoff.expo, openai.RateLimitError)
def call_llm(
    self,
    messages: list[dict[str, Any]],
) -> Generator[dict[str, Any], None, None]:
    with warnings.catch_warnings():
        warnings.filterwarnings(
            "ignore", message="PydanticSerializationUnexpectedValue"
        )
        stream = self.model_serving_client.chat.completions.create(
            model=self.llm_endpoint,
            messages=to_chat_completions_input(messages),
            tools=self.get_tool_specs(),
            stream=True,
        )
        with mlflow.start_span(name="call_llm", span_type=SpanType.LLM) as
span:
            span.set_inputs({"messages": messages})
            last_chunk: dict[str, Any] = {}
            for chunk in stream:
                chunk_dict = chunk.to_dict()
                last_chunk = chunk_dict
                yield chunk_dict
            llm_request_id = stream.response.headers.get("x-request-id")
            outputs: dict[str, Any] = {
                "model": last_chunk.get("model"),
                "usage": last_chunk.get("usage"),
            }
            if llm_request_id:
```

```
                outputs["llm_request_id"] = llm_request_id
            span.set_outputs(outputs)
```

When the model decides to call a tool, the `handle_tool_call` method handles the execution flow. The tool arguments are parsed from JSON, and the corresponding tool gets executed. The tool output is wrapped into a function-call output item and gets appended to the running message history. The `ResponsesAgentStreamEvent` is then returned, to complete the tool call:

```
def handle_tool_call(
    self, tool_call: dict[str, Any], messages: list[dict[str, Any]]
) -> ResponsesAgentStreamEvent:
    """
    Execute tool calls, add them to the running message history,
    and return a ResponsesStreamEvent w/ tool output
    """
    args = json.loads(tool_call["arguments"])
    result = str(self.execute_tool(tool_name=tool_call["name"], args=args))

    tool_call_output = self.create_function_call_output_item(
        tool_call["call_id"], result
    )
    messages.append(tool_call_output)
    return ResponsesAgentStreamEvent(
        type="response.output_item.done", item=tool_call_output
    )
```

The `arxivAgent` class provides dedicated methods for loading and saving messages to memory. While this may seem redundant, it ensures that each operation is properly traced via the `mlflow.trace` decorator. By isolating memory access in these methods, both loading and saving are captured as separate trace spans, to improve observability and make debugging easier:

```
@mlflow.trace(span_type=SpanType.RETRIEVER, name="memory_load")
def load_memory(self, session_id: str) -> list[dict[str, Any]]:
    """Load previous messages from Lakebase memory."""
    if self.memory:
        return self.memory.load_messages(session_id)
```

```
        return []

    @mlflow.trace(span_type=SpanType.CHAIN, name="memory_save")
    def save_memory(
        self, session_id: str, messages: list[dict[str, Any]]
    ) -> None:
        """Save new messages to Lakebase memory."""
        self.memory.save_messages(session_id, messages)
```

The `call_and_run_tools()` method implements the core agent loop. It builds the message history from the system prompt, optional previous messages (loaded from memory), and the current request, to enable stateful interactions.

It then runs `_run_tool_loop()`, which alternates between executing tool calls and invoking the LLM until a final response is produced or a maximum number of steps is reached. For simplicity, we omit the implementation of `_run_tool_loop()` here, but it's available in the full book code.

The method is traced with `@mlflow.trace` and enriched via `mlflow.update_current_trace()`, which adds tags such as `git_sha`, `model_version`, and `model_serving_endpoint_name`. These environment-based tags ensure that every run is traceable and make it clear whether the code was executed locally or behind a serving endpoint.

Finally, if memory is enabled, the method saves both the input and generated outputs, to allow future calls to continue the conversation seamlessly:

```
    @mlflow.trace(span_type=SpanType.CHAIN)
    def call_and_run_tools(
        self,
        request_input: list[dict[str, Any]],
        previous_messages: list[dict[str, Any]] | None = None,
        request_id: str | None = None,
        session_id: str | None = None,
    ) -> list[ResponsesAgentStreamEvent]:
        messages: list[dict[str, Any]] = [
            {"role": "system", "content": self.system_prompt},
```

```
]
if previous_messages:
    messages.extend(previous_messages)
messages.extend(request_input)

mlflow.update_current_trace(
    tags={
        "git_sha": os.getenv("GIT_SHA", "local"),
        "model_serving_endpoint_name": os.getenv(
            "MODEL_SERVING_ENDPOINT_NAME", "local"),
        "model_version": os.getenv("MODEL_VERSION", "local"),
    },
    metadata=(
        {"mlflow.trace.session": session_id} if session_id else {}),
    client_request_id=request_id,
)

events = self._run_tool_loop(messages)

if session_id and self.memory and request_input:
    self.save_memory(
        session_id,
        request_input + self._extract_output_items(events),
    )
return events
```

Prediction and Streaming

The `predict()` method implements the nonstreaming interface required by `ResponsesAgent`. It simply calls `predict_stream()`, collects the emitted events, and returns the final outputs as a ResponsesAgentResponse. This makes it suitable for synchronous or batch use cases.

The `predict_stream()` method serves as the main entry point for execution. It extracts the `session_id` and `request_id` from the request, loads any previous messages from memory, and converts the incoming inputs into the internal message format.

The `predict_stream()` method delegates all core execution logic to `call_and_run_tools`, which we discussed earlier. It then yields the resulting events as they are produced:

```
def predict(self, request: ResponsesAgentRequest) -> ResponsesAgentResponse:
    events = list(self.predict_stream(request))
    return ResponsesAgentResponse(
        output=self._extract_output_items(events),
        custom_outputs=request.custom_inputs,
    )

@mlflow.trace(span_type=SpanType.AGENT)
def predict_stream(
    self, request: ResponsesAgentRequest
) -> Generator[ResponsesAgentStreamEvent, None, None]:
    custom = request.custom_inputs or {}
    session_id = custom.get("session_id")
    request_id = custom.get("request_id")

    previous_messages = (
        self.load_memory(session_id)
        if session_id and self.memory else []
    )

    request_input = [i.model_dump() for i in request.input]
    events = self.call_and_run_tools(
        request_input=request_input,
        previous_messages=previous_messages,
        request_id=request_id,
        session_id=session_id,
    )
    yield from events
```

The agent is defined as part of the package in the `arxiv_curator.agent` module. Let's now use this module and see how the agent works.

Testing the Agent

Let's take a look at the *chapter8_2.test_agent.py* notebook. Here, we start by importing the project configuration and initializing the agent. Key settings—such as the MLflow experiment name, the Lakebase project ID, which LLM endpoint to use, and the system prompt—are defined in the *project_config.yml* file:

```
import random
from datetime import datetime
```

```
import mlflow

from arxiv_curator.agent import ArxivAgent
from arxiv_curator.config import ProjectConfig
from arxiv_curator.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()

cfg = ProjectConfig.from_yaml("../project_config.yml")
mlflow.set_experiment(cfg.experiment_path)

agent = ArxivAgent(
   llm_endpoint=cfg.llm_endpoint,
   system_prompt=cfg.system_prompt,
   catalog=cfg.catalog,
   schema=cfg.schema,
   genie_space_id=cfg.genie_space_id,
   lakebase_project_id=cfg.lakebase_project_id
)

mlflow.models.set_model(agent)
```

We want to verify that the agent behaves as expected when custom inputs are provided, and that its memory is updated correctly. To do this, we call the `predict()` method twice, using the same `session_id` for both requests.

This allows us to simulate a multiturn interaction, in which the second request can build on the context of the first. By reusing the same session, we can confirm that previous messages are successfully stored and retrieved from memory, to enable the agent to respond in a context-aware manner:

```
timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
session_id = f"s-{timestamp}-{random.randint(100000, 999999)}"

first_request = {
   "input": [
       {"role": "user",
        "content": "What are recent papers about LLMs and reasoning?"}
   ],
   "custom_inputs": {
       "session_id": session_id,
       "request_id": f"req-{timestamp}-1",
```

```
    },
}

result_1 = agent.predict(first_request)

second_request = {
    "input": [
        {"role": "user",
         "content": "Rewrite that post focusing on chain-of-thought
prompting."}
    ],
    "custom_inputs": {
        "session_id": session_id,
        "request_id": f"req-{timestamp}-2",
    },
}

result_2 = agent.predict(second_request)
```

After we invoke the `predict` functions, we'll see that an MLflow trace has appeared under the defined experiment (see Figure 8-3):

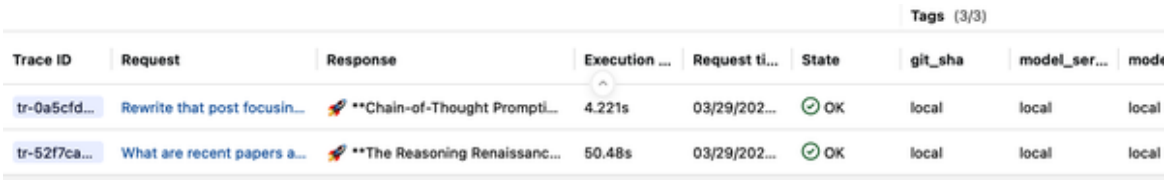


Figure 8-3. Traces from the agent invocation

We can inspect the trace in the UI to view a detailed breakdown of the execution and the final output. By opening a trace, we can also confirm that the `session_id` we passed in the code is captured in the trace metadata (see Figure 8-4).

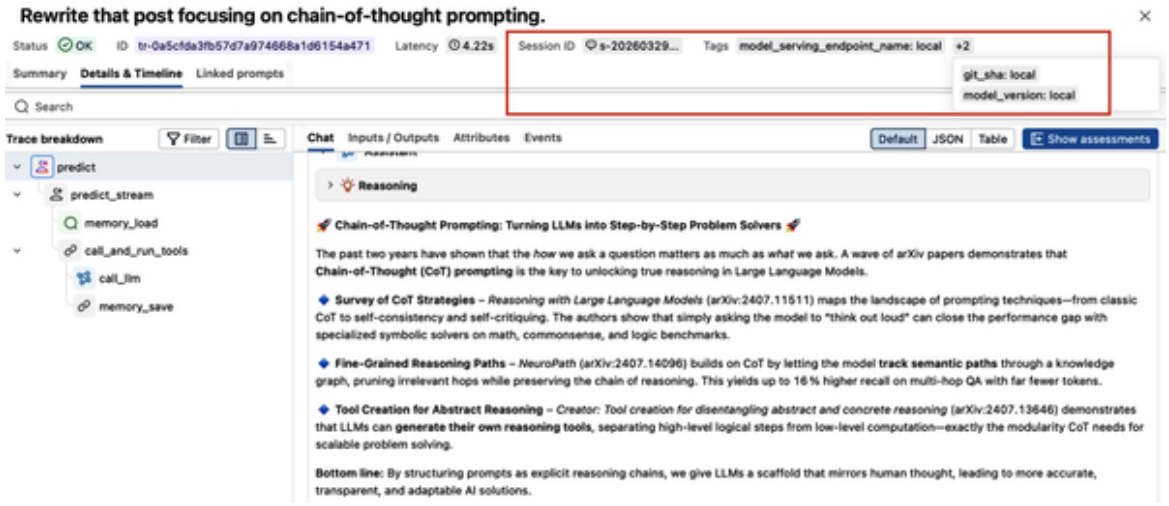


Figure 8-4. Inspecting the trace

LLM System Evaluation

Before we take a deep dive into the code, let's talk about what evaluating an LLM system means. Simply put, *evaluation* is the systematic process of measuring a system's quality. In traditional ML systems, we make this judgment by relying on well-established evaluation metrics, such as accuracy or the F1 score for classification and RMSE or R^2 for regression. LLM systems, however, are much harder to evaluate, which is why teams often treat *evals* (a term that practitioners commonly use to refer to LLM evaluations) as an afterthought.

During the LLM system development phase, teams can get pretty far with manual testing and going with their gut feelings. However, once the system reaches production and real users are using it, building without evals quickly becomes unsustainable. And after you receive complaints and make changes to the system, there will be no reliable way to know whether other behaviors have not silently regressed.

Just as we wouldn't skip evaluation in a standard ML workflow, we shouldn't skip it here. Evals are most valuable when implemented early, for several reasons:

Explicit behavior

They force you to define exactly what “good behavior” looks like for your agent.

Measurable impact

They allow you to measure how changes to orchestration logic, prompts, or the underlying model affect performance.

System optimization

In many ways, choosing a prompt or a temperature setting is like tuning hyperparameters in classic ML; evals provide the data you need to make those choices.

How to Evaluate LLM Systems

If you’re starting with evals early, the best approach is a simple, iterative loop. Experts Aishwarya Naresh Reganti and Kiriti Badam, in their materials on AI Evals,[1] suggest this workflow:

1. Create examples: start small; 10–20 high-quality examples are better than 100 mediocre ones.
2. Run test cases: process them through your current agent.
3. Evaluate with the business: determine whether the output has met expectations.
4. Identify error patterns: explain exactly why an output failed.
5. Decide on key metrics: determine what you are actually measuring.
6. Iterate and expand: refine your prompts or tools and repeat.

Anthropic’s “Demystifying evals for AI agents” article adds a few useful principles:

Product requirements naturally translate into test cases

If a requirement states that *“The agent must cite its sources,”* you can turn that into a test that verifies whether responses include references to the retrieved documents.

The “expert agreement” rule

A good test case is one in which two domain experts independently reach the same pass/fail verdict. If they disagree, the example is too noisy.

Test negative behaviors

Don’t forget to test both cases in which a behavior should occur and where it shouldn’t. Imbalanced datasets are just as problematic in agent evaluation as they are in traditional ML.

Table 8-1 illustrates how test cases for a technical support agent can be processed through an LLM system and evaluated alongside business stakeholders.

Table 8-1. Examples of test cases and outputs from an evaluation session

Input	System output	Expected response	Satisfactory	Reas
“Same issue as last month”	“Can you describe the problem?”	Reference previous tickets and confirm if there is a related issue.	No	Previ ticke not re and t was f repea conte
“Just a quick question about settings”	“Sure, which settings are you looking at? I can walk you through it.”	Respond proportionally and ask clarifying questions.	Yes	-
“Tried everything you suggested”	“Have you tried restarting?”	Acknowledge previous attempts and escalate or try a new approach.	No	An al tried sugge was r and t conv conte not main

You can identify errors and potential causes after the initial evaluation session. For example, if the user says, “Same issue as last month,” and if the agent fails to retrieve the issue from the history, then the root cause may be missing access to ticket data. This isn’t a quick fix: it requires you to add

a tool and integration, unlike with issues that you can address by changing the prompt alone.

As a next step, you should identify metrics. A *metric* represents a *behavior or quality you want to track over time*. Examples include tool usage correctness (for example, whether the agent attempted to retrieve data from history), tone appropriateness, number of tool calls, latency, and cost. For systems that produce structured output, common metrics include structured output correctness and simple quantitative measures such as output length and character count.

After you define metrics, you choose graders to measure them. A *grader* (also called a *scorer*) is the mechanism you use to evaluate a metric. In practice, agent evaluations typically rely on three types of graders:

Code-based graders

> These are deterministic checks that are implemented directly in code. Examples include string and regular expression (regex) matching, character and token counts, output structure validation (for example, using Pydantic, as shown earlier), tool call verification, latency measurement, and cost tracking. Code-based graders are fast, inexpensive, objective, and reproducible, so they're ideal to introduce early and run continuously. Their main limitation is a lack of nuance: they are poorly suited to evaluating subjective or context-dependent behaviors.

Model-based graders

> These are often referred to as *LLM judges*, and they use a language model to assess qualities that are difficult to capture with code alone, such as adherence to guidelines, correctness of reasoning, and appropriateness of text based on tone and context. These graders are more flexible and can handle open-ended outputs, but they are more

expensive and nondeterministic. To make them reliable, you typically need to calibrate them against human judgment.

Human graders

Using these is unavoidable if you want to have an LLM system that truly meets business expectations. While slow and expensive, human evaluation, ideally performed by subject matter experts, serves as the gold standard. Human graders are essential for defining evaluation criteria, calibrating model-based graders, and reviewing edge cases. In production, they are often used for spot checks to ensure the system continues to behave as intended as user behavior and real-world conditions evolve.

It's also important to keep in mind that this is still a relatively young field. While terminology and naming conventions may vary across tools and platforms, the underlying principles remain consistent. Each framework introduces its own vocabulary for similar concepts.

With that context in mind, let's take a closer look at the MLflow GenAI evaluation framework.

MLflow GenAI Evaluation Framework

Scorers serve as the core components of the MLflow GenAI evaluation framework. To help you get started, MLflow provides several *predefined model-based scorers* that are designed to assess different aspects of model or agent behavior.

At a minimum, every evaluation requires scorers to have access to two things: *inputs* (what was provided to the system) and *outputs* (what the system produced). These correspond to the first and second columns of our earlier test cases in Table 8-1. In addition, you must also specify the *evaluation model* that will act as the judge. For some scorers, this information is sufficient: metrics like *RelevanceToQuery, Completeness*,

Fluency, and *Safety* can evaluate a response using only the inputs and outputs, without requiring any additional annotations.

Other scorers rely on *user-provided evaluation criteria*. For example, *Correctness* and *Equivalence* require explicit annotations, such as reference answers, expected facts, or task-specific expectations. This is similar to the information shown in column 3 of Table 8-1.

Finally, some scorers operate on agent behavior rather than just text output. *Tool-calling* and *retrieval-related scorers* need access to traces so they can inspect tool usage, parameters, and execution order. In addition, certain scorers that measure conversation guidelines or role adherence are *session based*, meaning they evaluate the entire conversation or interaction sequence instead of individual input-output pairs.

Table 8-2 summarizes MLflow's predefined model-based scorers.

Table 8-2. MLflow scorers[a]

Scorer	What it evaluates	Does it require expectations | traces | a session?
RelevanceToQuery	Does the response directly address the user's input?	No | No | No
Correctness	Are the expected facts supported by the response?	Yes | No | No
Completeness	Does the agent address all questions in a single user prompt?	No | No | No
Fluency	Is the response grammatically correct and naturally flowing?	No | No | No
ExpectationsGuidelines	Does the response meet specific expectations and guidelines?	Yes | No | No

Scorer	What it evaluates	Does it require expectations | traces | a session?
Safety	Is the response free of harmful or toxic content?	No | No | No
Equivalence	Is the response equivalent to the expected output?	Yes | No | No
RetrievalGroundedness	Is the response grounded in retrieved information?	No | Yes | No
RetrievalRelevance	Are retrieved documents relevant to the user’s request?	No | Yes | No
RetrievalSufficiency	Do retrieved documents contain all necessary information?	Yes | Yes | No
ToolCallCorrectness	Are the tool calls and arguments correct for the user query?	No | Yes | No

Scorer	What it evaluates	Does it require expectations | traces | a session?
ToolCallEfficiency	Are the tool calls efficient without redundancy?	No | Yes | No
ConversationCompleteness	Does the agent address all user questions throughout the conversation?	No | Yes | Yes
ConversationalGuidelines	Do the assistant's responses comply with provided guidelines?	No | Yes | Yes
ConversationalRoleAdherence	Does the assistant maintain its assigned role throughout the conversation?	No | Yes | Yes
ConversationalSafety	Are the assistant's responses safe and free of harmful content?	No | Yes | Yes
ConversationalToolCallEfficiency	Was tool usage across the conversation	No | Yes | Yes

Scorer	What it evaluates	Does it require expectations | traces | a session?
	efficient and appropriate?	
KnowledgeRetention	Does the assistant correctly retain information from earlier user inputs?	No | Yes | Yes
UserFrustration	Is the user frustrated and was any frustration resolved?	No | Yes | Yes

[a] Adapted from MLflow documentation where you can find an overview of all available scorers.

Predefined scorers in MLflow evaluate model behavior by using instructions that are built directly into the library. They are useful for getting started quickly and for generating early evaluation signals, but on their own, they are rarely sufficient for rigorous or production-grade evaluation.

To support more tailored evaluation needs, MLflow allows you to define *custom scorers*. The simplest of these is the *Guidelines scorer*, which lets you specify general, natural language rules that apply to all tasks without requiring per-example expectations. Because guidelines are evaluated as

pass/fail conditions, this scorer provides a fast and accessible way to customize evaluation behavior.

For more nuanced and flexible evaluation, MLflow offers *template-based LLM scorers*. These scorers enable richer evaluation logic by allowing the judge to reason over additional context, such as expectations, conversation history, and execution traces.

Figure 8-5 compares the different types of LLM-based scorers and their complexity levels.

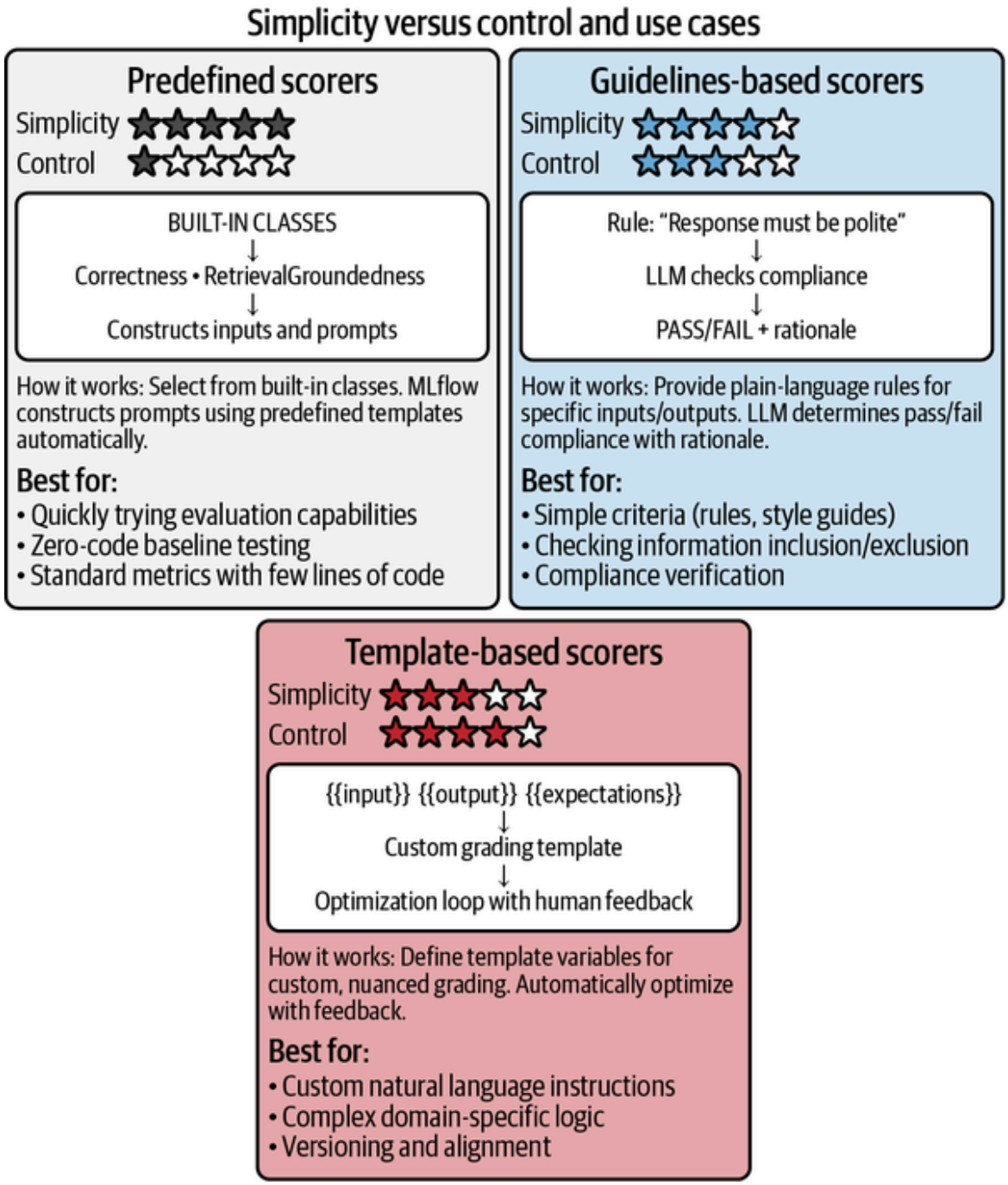


Figure 8-5. MLflow LLM scorer approaches

The differences between guidelines-based and template-based LLM scorers are not always obvious at first glance. To make these distinctions concrete, we'll compare the Guidelines scorer with a template-based LLM scorer we'll create by using the `make_judge` function. We'll start with a simple example:

```
import mlflow
from mlflow.genai.scorers import Guidelines

mlflow.set_experiment("/Shared/guidelines-example")
escalation_guidelines = Guidelines(
   name="escalation_handling",
   guidelines=[
       "When a user indicates previous attempts failed, the response must "
       "acknowledge their efforts and either escalate or offer a new approach"
   ],
   model="databricks:/databricks-gpt-oss-120b"
)

data = [
   {
       "inputs": {"message": "Tried everything you suggested"},
       "outputs": "Have you tried restarting?",
   },
   {
       "inputs": {"message": "Tried everything you suggested"},
       "outputs": (
           "I understand you've already tried the previous suggestions "
           "without success. Let me escalate this to our senior support "
           "team who can look into this more deeply. In the meantime, "
           "could you share any error logs you've encountered?"
       ),
   },
]

mlflow.genai.evaluate(data=data, scorers=[escalation_guidelines])
```

This evaluation session creates traces that show the first example failed the guideline and the second passed it (see Figure 8-6).

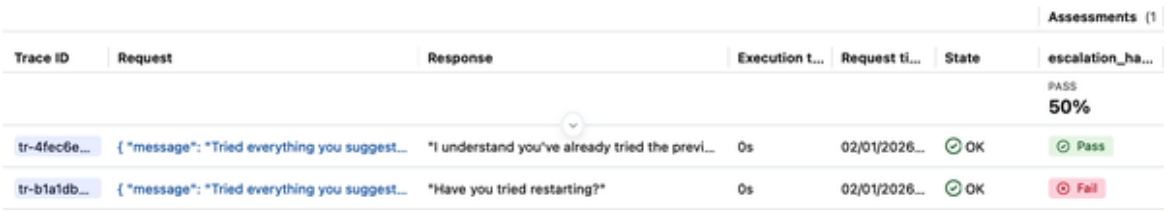


Figure 8-6. Traces: escalation handling based on guidelines

The Guidelines scorer produces binary outcomes (pass/fail). When you need more expressive feedback, such as a 1–5 rating, a continuous score, or structured diagnostic output, you should use a template-based LLM scorer.

While this is just one example of when an LLM scorer is the better choice, it represents a common and practical scenario.

In the following code snippet, we create a judge that produces an escalation quality score (from 1 to 5) instead of a binary output. Notice that the `make_judge()` function takes an instruction argument that contains template variables. Template-based LLM scorers support `{{ inputs }}`, `{{ outputs }}`, and `{{ expectations }}` template variables:

```
import mlflow
from mlflow.genai.judges import make_judge

# Example using make_judge for escalation quality evaluation (1-5 scale)
mlflow.set_experiment("/Shared/make-judge-example")
escalation_judge = make_judge(
   name="escalation_quality",
   instructions=(
       "Evaluate how well the response in {{ outputs }} handles the user's "
       "message in {{ inputs }} when they indicate previous attempts failed. "
       "Score from 1 to 5:\n"
       "1 - Ignores context, repeats already-tried suggestions\n"
       "2 - Acknowledges frustration but offers no new solutions\n"
       "3 - Offers a new approach but lacks empathy or clarity\n"
       "4 - Acknowledges efforts and offers a reasonable new approach\n"
       "5 - Empathetic, escalates appropriately or provides creative solution"
   ),
   model="databricks:/databricks-gpt-oss-120b",
   feedback_value_type=int,
)

mlflow.genai.evaluate(data=data, scorers=[escalation_judge])
```

This evaluation session creates traces that show scores of 5 and 1 (respectively) for the two examples we provided (see Figure 8-7).



Trace ID	Request	Response	Execution t...	Request ti...	State	Assessments (1 escalation_qu...
						STRING 2 values 1 50% 5 50%
tr-2404e1...	{ "message": "Tried everything you suggest...	"I understand you've already tried the previ...	0s	02/01/2026...	OK	5
tr-448bf4...	{ "message": "Tried everything you suggest...	"Have you tried restarting?"	0s	02/01/2026...	OK	1

Figure 8-7. Traces: escalation quality based on a template-based LLM scorer

One of the key benefits of using a template-based LLM scorer is that it lets you better align automated evaluation with human judgment. You can add human evaluations directly through the MLflow UI and then combine those annotations with trace data and LLM-based judge evaluations to align the scorer.

In this example, we use the Stochastic Introspective Mini-Batch Ascent (SIMBA) optimizer from the DSPy library. It enables an LLM to analyze its own performance and derive improvement rules by sampling mini-batches, identifying challenging examples with high output variability, and either generating self-reflective rules or incorporating successful examples as demonstrations. Custom optimizers are also supported.

To effectively align the judge, it is recommended to have at least 10 traces available for optimization:[2]

```
experiment_id = mlflow.search_experiments(
   filter_string="name='/Shared/make-judge-example'")[0].experiment_id

from mlflow.genai.judges.optimizers import SIMBAAlignmentOptimizer

# Retrieve traces with both judge and human assessments
traces_for_alignment = mlflow.search_traces(
   experiment_ids=[experiment_id],
   return_type="list"
)

# Filter for traces with both judge and human feedback
valid_traces = []
for trace in traces_for_alignment:
   feedbacks = trace.search_assessments(name="escalation_quality")
   has_judge = any(f.source.source_type == "LLM_JUDGE" for f in feedbacks)
   has_human = any(f.source.source_type == "HUMAN" for f in feedbacks)
```

```
    if has_judge and has_human:
        valid_traces.append(trace)

optimizer = SIMBAAlignmentOptimizer(
    model="databricks:/databricks-gpt-oss-120b"
)

aligned_judge = escalation_judge.align(optimizer, valid_traces)
```

MLflow also supports *Agent-as-a-Judge*, for which we can use the same `make_judge()` function and pass an instruction with the `{{ trace }}` template variable. Here is an example of an Agent-as-a-Judge that uses trace to evaluate performance of the system:

```
from typing import Literal

performance_judge = make_judge(
    name="performance_analyzer",
    instructions=(
        "Analyze the {{ trace }} for performance issues.\n\n"
        "Check for:\n"
        "- Operations taking longer than 2 seconds\n"
        "- Redundant API calls or database queries\n"
        "- Inefficient data processing patterns\n"
        "- Proper use of caching mechanisms\n\n"
        "Rate as: 'optimal', 'acceptable', or 'needs_improvement'"
    ),
    feedback_value_type=Literal["optimal", "acceptable", "needs_improvement"],
    model="databricks:/databricks-gpt-oss-120b",
```

The key difference between an LLM-as-a-Judge and an Agent-as-a-Judge is that the latter has access to tools for inspecting traces stored in the MLflow backend. This allows it to retrieve high-level trace metadata, inspect individual spans, and search for patterns across spans using regular expressions.

In addition to complex evaluation mechanisms, MLflow supports *code-based scorers*, which allow you to implement custom evaluation logic directly in Python. These are best suited for deterministic checks such as structural validation, tool usage verification, latency or cost thresholds, and

other objective criteria that are difficult or unnecessary to express in natural language.

Beyond evaluation, scorers also provide the objective metrics needed for prompt optimization.[3] MLflow supports several optimization techniques, including Multiprompt Instruction PRoposal Optimizer Version 2 (MIPROv2) from the DSPy library.

MIPROv2 jointly refines task instructions and few-shot examples by bootstrapping candidate examples, proposing alternative instructions that reflect different task strategies, and selecting the best combination using Bayesian optimization. The optimizer can be used to tune both instructions and few-shot examples together or to optimize instructions alone in a zero-shot setting.

Prompt optimization is conceptually similar to hyperparameter tuning in traditional machine learning: it is an iterative process that you should revisit periodically as your system, prompts, and data evolve.

Evaluating, Logging, and Registering the Agent

To see evaluation in practice, we'll create a code-based scorer and a custom guideline. We are intentionally keeping these examples straightforward; the goal here isn't to build the "perfect" agent but to demonstrate the specific tools you need to operate an agent effectively. Just as we did with traditional models in Chapter 3, we'll log these evaluation results directly to your MLflow experiment for easy tracking and comparison.

Finally, after we evaluate the agent, we can log it and make it ready for serving by registering it in Unity Catalog.

Evaluation in Practice

Before we can evaluate an agent, we need to provide evaluation examples. They can range from simple inputs to more detailed pairs that include both

inputs and expectations. In our case, we'll go for a simple approach by providing only the inputs. Our test cases are stored in an *eval_inputs.txt* file that contains the following records:

```
What is so special about the last paper on RAG?
Summarize the key findings from the last paper on reasoning.
What problem does the last paper on multi-agent systems solve?
Tell me about the last paper on prompt engineering.
What are the main contributions of the last paper on LLM fine-tuning?
Explain the methodology in the last paper on chain-of-thought prompting.
What results did the last paper on hallucination detection achieve?
Describe the last paper on LLM benchmarks.
What approach does the last paper on code generation propose?
What conclusions does the last paper on embedding models draw?
```

In the *chapter8_4.evaluate_log_register_agent.py* notebook, we initialize the agent by following the same process as before, define the predict function used for evaluation, and load these records into memory:

```
import mlflow
from mlflow.genai.scorers import Guidelines

from arxiv_curator.agent import ArxivAgent
from arxiv_curator.config import ProjectConfig
from arxiv_curator.evaluation import (
    hook_in_post_guideline,
    polite_tone_guideline,
    word_count_check,
)

from arxiv_curator.utils.common import set_mlflow_tracking_uri

set_mlflow_tracking_uri()

cfg = ProjectConfig.from_yaml("../project_config.yml")
mlflow.set_experiment(cfg.experiment_path)

agent = ArxivAgent(
    llm_endpoint=cfg.llm_endpoint,
    system_prompt=cfg.system_prompt,
    catalog=cfg.catalog,
    schema=cfg.schema,
    genie_space_id=cfg.genie_space_id,
    lakebase_project_id=cfg.lakebase_project_id,
```

```
)

# Load evaluation inputs
with open("../eval_inputs.txt") as f:
    eval_data = [{"inputs": {
        "question": line.strip()}} for line in f if line.strip()]


def predict_fn(question: str) -> str:
    """Predict function that wraps the agent for evaluation."""
    request = {"input": [{"role": "user", "content": question}]}
    result = agent.predict(request)
    return result.output[-1].content
```

Scorers are defined under the evaluation module of our package. Here, we specify guidelines (we want the LinkedIn post to have an engaging hook and a polite, professional tone). We also define a code-based scorer that evaluates that the length of the post must be no longer than 350 words:

```
import mlflow
from mlflow.genai.scorers import Guidelines

polite_tone_guideline = Guidelines(
    name="polite_tone",
    guidelines=[
        "The response must use a polite and professional tone throughout",
        "The response should be friendly and helpful without being
condescending",
        "The response must avoid any dismissive or rude language"
    ],
    model="databricks:/databricks-gpt-oss-120b"
)

hook_in_post_guideline = Guidelines(
    name="hook_in_post",
    guidelines=[
        "The response must start with an engaging hook that captures
attention",
        "The opening should make the reader want to continue reading",
        "The response should have a compelling introduction before diving into
details"
    ],
    model="databricks:/databricks-gpt-oss-120b"
)
```

```
# Define code-based scorer for word count
@mlflow.genai.scorer
def word_count_check(outputs: list) -> bool:
    """Check that the output is under 350 words."""
    word_count = len(outputs[0]["text"].split())
    return word_count < 350
```

With the scorers and evaluation data in place, we can now run the evaluation:

```
mlflow.set_experiment("/Shared/genai-arxiv-agent")

results = mlflow.genai.evaluate(
    predict_fn=predict_fn,
    data=eval_data,
    scorers=[word_count_check,
             polite_tone_guideline,
             hook_in_post_guideline]
)
```

Once the evaluation is complete, you can retrieve the results directly from the `EvaluationResults` object, where they are stored within the metrics attribute. For a more visual deep dive, you can inspect the performance data in the MLflow tracing UI (see Figure 8-8). This view is particularly helpful as it provides a clear breakdown of how frequently the agent's responses passed your defined assessments.

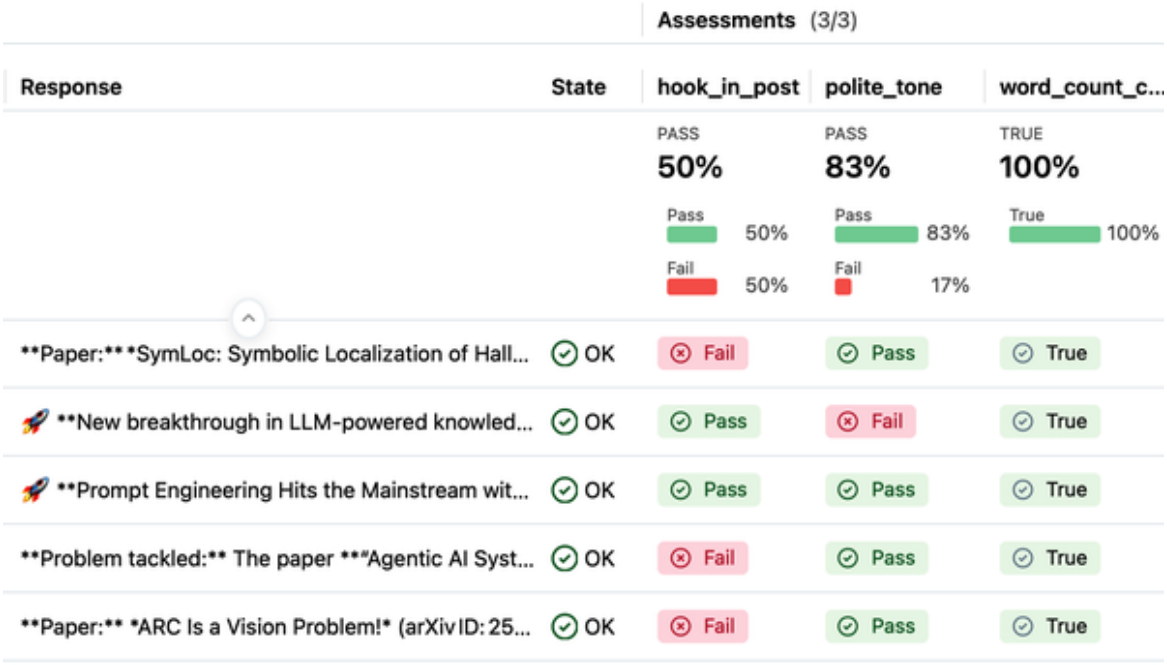


Figure 8-8. Evaluation results on the MLflow tracing UI

We can use the logic we defined to run evaluation before logging the model, as well as to monitor the system once it is in production. However, before we can explore those monitoring workflows in detail, we need to log and register the agent.

Agent Logging

Custom agents that inherit from the `mlflow.pyfunc.ResponsesAgent` class can be logged as code.[4] This means that we must specify the agent, along with all its resources, in a Python file. In our project, the *arxiv_agent.py* file is located in the root of the repository. While the code is very similar to what we used for agent evaluation, there are a few key differences.

For instance, we specify `ModelConfig`, which is used for development purposes. You may notice that we are repeating the values from the *project_config.yml* file (except that we've replaced the prompt). Think of these values as placeholders; the actual values from the *project_config.yml* file will be passed in the model logging step. We also must use the `set_model()` function from the `mlflow.models` module, which allows

MLflow to package the agent as code. Here is the structure of the `arxiv_agent.py` script:

```
import mlflow
from mlflow.models import ModelConfig

from arxiv_curator.agent import ArxivAgent

config = ModelConfig(
   development_config={
       "catalog": "mlops_dev",
       "schema": "arxiv",
       "genie_space_id": "01f0e97a42981382b3d16f3f1899fdb5",
       "system_prompt": "prompt placeholder",
       "llm_endpoint": "databricks-gpt-oss-120b",
       "lakebase_project_id": "arxiv-agent-lakebase",
   }
)

agent = ArxivAgent(
   llm_endpoint=config.get("llm_endpoint"),
   system_prompt=config.get("system_prompt"),
   catalog=config.get("catalog"),
   schema=config.get("schema"),
   genie_space_id=config.get("genie_space_id"),
   lakebase_project_id=config.get("lakebase_project_id"),
)
mlflow.models.set_model(agent)
```

Once you've defined the agent as code, we can proceed with model logging. Before doing so, however, you must define the model's *resource dependencies* and *configuration*. Make sure to define all underlying dependencies, such as the tables Genie needs to access:

```
from datetime import datetime

import mlflow
from mlflow.models.resources import (
   DatabricksGenieSpace,
   DatabricksServingEndpoint,
   DatabricksVectorSearchIndex,
   DatabricksTable,
   DatabricksSQLWarehouse
)
```

```
from arxiv_curator.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")

model_name = f"{cfg.catalog}.{cfg.schema}.arxiv_agent"

resources = [
   DatabricksServingEndpoint(endpoint_name=cfg.llm_endpoint),
   DatabricksGenieSpace(genie_space_id=cfg.genie_space_id),
   DatabricksVectorSearchIndex(
       index_name=f"{cfg.catalog}.{cfg.schema}.arxiv_index"),
   DatabricksTable(table_name=f"{cfg.catalog}.{cfg.schema}.arxiv_papers"),
   DatabricksSQLWarehouse(warehouse_id=cfg.warehouse_id),
   DatabricksServingEndpoint(endpoint_name="databricks-bge-large-en"),
]

model_config = {
       "catalog": cfg.catalog,
       "schema": cfg.schema,
       "genie_space_id": cfg.genie_space_id,
       "system_prompt": cfg.system_prompt,
       "llm_endpoint": cfg.llm_endpoint,
       "lakebase_project_id": cfg.lakebase_project_id,
   }
```

We can log the agent by using the same approach we used for the custom model we introduced in Chapter 3. In this example, we start by providing placeholder values for `git_sha` and the Databricks `run_id` job, along with the path to the Python model relative to the notebook.

When the job executes, the script receives the actual values for these parameters at runtime, following the same configuration pattern we used in Chapter 5:

```
git_sha = "abc"
run_id = "unset"

mlflow.set_experiment("/Shared/genai-arxiv-agent")
ts = datetime.now().strftime('%Y-%m-%d')
with mlflow.start_run(
   run_name=f"arxiv-mcp-agent-{ts}",
   tags={"git_sha": git_sha, "run_id": run_id}
) as run:
```

```
    model_info = mlflow.pyfunc.log_model(
        name="agent",
        python_model="../arxiv_agent.py",
        resources=resources,
        input_example=test_request,
        model_config=model_config
    )
```

Registering the Agent and Preparing for Serving

Once the model is logged, we can proceed with registering it:

```
model_name = f"{cfg.catalog}.{cfg.schema}.arxiv_agent"

registered_model = mlflow.register_model(
    model_uri=model_info.model_uri,
    name=model_name,
    tags={"git_sha": git_sha, "run_id": run_id},
    env_pack="databricks_model_serving"
)
```

Notice that we pass the `env_pack` argument with the `databricks_model_serving` value. This indicates that we will use the express deployment option (previously, serverless optimized deployment),[5] which significantly simplifies the handling of custom dependencies.

Although the agent defined in the `arxiv_agent.py` script imports the custom `arxiv_curator` package, we do not log this package explicitly, unlike with the approach we used in Chapter 2. The *serverless environment packaging* mechanism handles dependency resolution automatically. For this to work correctly, both the model logging and registration steps must run in the *same serverless environment*.

Simplifying custom packaging is not the primary benefit of this approach. The main advantage is significantly *faster agent deployment and autoscaling*. Keep in mind that model registration takes slightly longer than usual in this case. Another important constraint to keep in mind is *model size*: Databricks serverless optimized deployments support a maximum

model size of *1 GB*, compared to *4 GB* when environment packaging is not used.[6]

The environment packaging stores all Python dependencies together with the model. We can find them under the *_databricks* folder in the agent Artifacts tab (see Figure 8-9).

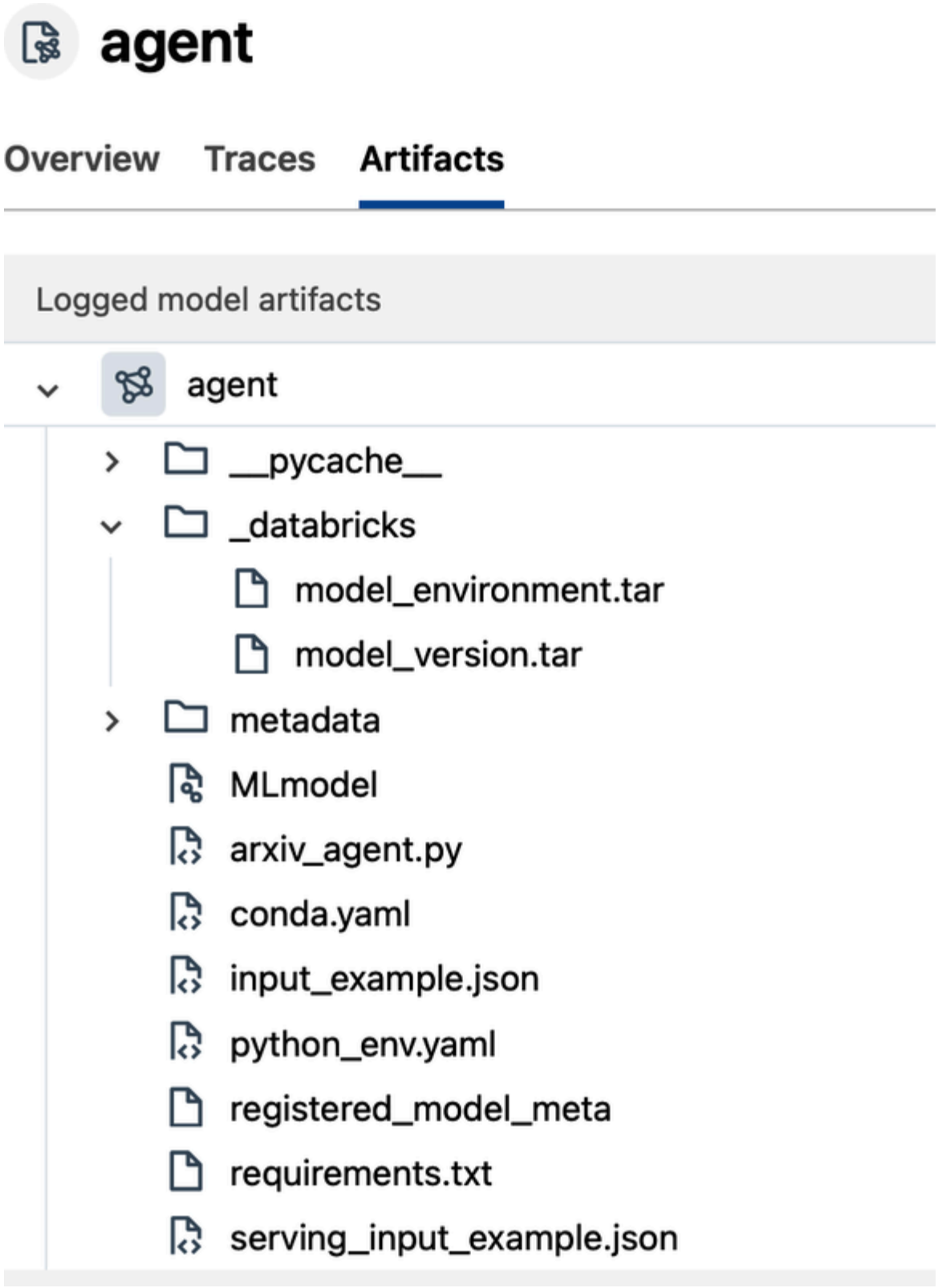


Figure 8-9. _databricks folder with project dependencies

After we register the agent, we can set the latest-model alias so that we can easily retrieve the latest version of the agent:

```
from mlflow import MlflowClient

client = MlflowClient()
client.set_registered_model_alias(
   name=model_name,
   alias="latest-model",
   version=registered_model.version,
)
```

Now that we've successfully logged and registered the agent, let's talk about deployment.

Conclusion

In this chapter, we took the agent from a definition to a managed, observable model. We explored the MLflow flavors used to represent an agent, and then, we logged and registered it as a versioned model in Databricks Unity Catalog. We evaluated it with MLflow's GenAI evaluation tools to measure its quality and compare versions. And we added MLflow tracing, which records each step the agent takes: every retrieval, tool call, and model response.

A logged, evaluated, and traced agent is ready to leave the development environment. The next chapter covers deployment: serving the agent behind an endpoint and monitoring it in production, so the evaluation and tracing we set up here keep working once real traffic arrives.

---

1 Check out their repository.

2 For more details, check out the Databricks documentation on judge alignment.

3 Consult the MLflow documentation for more details on prompt optimization.

4 You can consult the Mlflow documentation on models from code.

[5] For Model Serving endpoints, consult the documentation on express deployments.

[6] CPU workload sizes of 8 GB and 16 GB are also available with regular deployments, per the documentation.

Chapter 9. Deploying in Monitoring LLM-Based Systems

In this chapter, we explore how to deploy an agent and the infrastructure around it. There is no single “correct” architecture as the right approach depends on the use case, latency requirements, and governance constraints.

Many agent use cases are naturally batch oriented. Agents can enrich newly ingested documents with summaries or metadata, classify support tickets, and generate offline reports. In these scenarios, running the agent as part of a scheduled batch job, much like how we tested it earlier, is often sufficient, without requiring a real-time endpoint.

Another pattern is embedding the agent directly into an application codebase. While convenient at first, this can create separation-of-concerns issues by tightly coupling application logic with agent orchestration and configuration. Over time, this makes versioning, monitoring, and governance more difficult, especially when different teams own the agent and application layers.

A more robust, production-ready approach is to deploy the agent behind a Model Serving endpoint. This introduces a clear separation between the agent and the consuming application, which makes it easier to evolve the agent independently, manage versions, and support multiple clients.

In this chapter, we follow this approach and deploy the agent using a serving endpoint.

Deploying a Databricks Model Serving Endpoint

In Chapter 4, we deployed several Model Serving endpoints using Databricks SDK. While we could use the same approach for agents, that would require us to do additional setup, such as provisioning a service principal with access to the MLflow experiment and configuring environment variables (such as `DATABRICKS_CLIENT_ID`, `DATABRICKS_CLIENT_SECRET`, and `DATABRICKS_HOST`) for authentication. This can conflict with the resource authentication setup we defined earlier, and it adds unnecessary complexity.

To avoid this, we'll use the Databricks agent framework. It handles deployment, tracing, and authentication out of the box, so we can focus on the agent logic rather than infrastructure.

Unity AI Gateway and Guardrails

The agent framework enables the inference table as part of the AI gateway configuration, so that we can log the payload that goes through the Model Serving endpoint. At the time of writing, payload logging was the only supported configuration available for the AI agents. More advanced features like rate limiting and cost tracking are reserved for external model endpoints and provisioned throughput and pay-per-token foundation model APIs. Because these features evolve rapidly, it is always a good idea to consult the Databricks documentation for the latest updates. Table 9-1 summarizes the AI gateway features that are currently supported and not supported.

Table 9-1. Supported AI gateway features by endpoint type

Feature	External model endpoint	Foundation Model API provisioned throughput endpoint	Foundation Model API pay-per-token endpoint	Cust endp *mlflo Pyth*
Permission and rate limiting	Supported	Supported	Supported	Supp
Payload logging	Supported	Supported	Supported	Supp
Usage tracking	Supported	Supported	Supported	Supp
AI Guardrails	Supported	Supported	Supported	*Not s*
Fallbacks	Supported	*Not supported*	*Not supported*	*Not s*
Traffic splitting	Supported	Supported	*Not supported*	Supp

Guardrails on the LLM level

It is worth noting that when we're deploying an AI agent, AI guardrails are not supported at the gateway level of a serving endpoint. However, we can still apply guardrails at the level of the underlying LLM used by the agent.

In our case, we use the Databricks-hosted `gpt-oss-120b` model. On top of it, we can enable guardrails for PII (blocking or redaction), jailbreak

detection, hallucination detection, and unsafe content, or we can define our own custom guardrails (see Figure 9-1).[1]

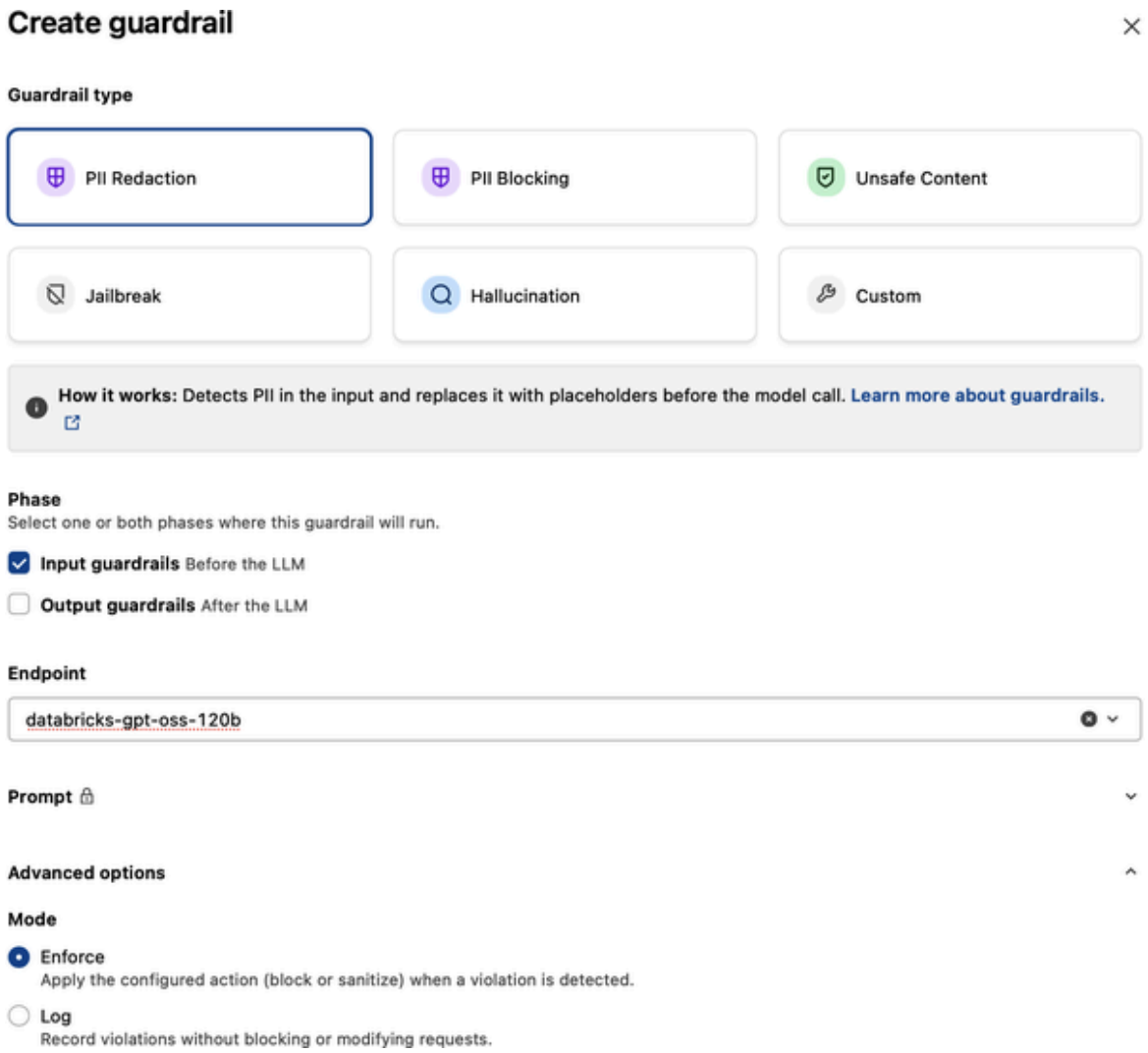


Figure 9-1. AI gateway settings for creating custom guardrails

Each guardrail can run in enforce or log mode. In *enforce mode*, the guardrail acts on a violation: it blocks the request or redacts the offending content before the response reaches the user. In *log mode*, it takes no action and only records that the violation occurred. Log mode lets you see how often a guardrail would fire against real traffic without affecting anyone, so a common pattern is to start in log mode, confirm that the guardrail behaves as expected, and switch to enforce once you trust it.

While these guardrails operate at the model level rather than the endpoint gateway, they still play an important role in ensuring safe and compliant agent behavior.

Guardrails as part of the agent logic

It's also worth noting that guardrails can (and should) be incorporated directly into the agent logic. While LLMs are inherently flexible, this flexibility can lead to undesired behavior if left unchecked. For example, in a recipe-generation app, you may want to prevent users from asking unrelated questions, such as by making Python programming queries. You can address this by adding custom validation logic, for instance, by using `mlflow.genai.evaluate()` to enforce domain constraints.

MLflow also supports *Guardrails AI scorers*. Guardrails AI provides a wide range of validators for safety, PII detection, content quality, and more. Importantly, these validators are *not always LLM-based*—many rely on rule-based approaches such as regex or keyword matching. This makes them faster, cheaper, and more predictable, which is especially useful in production systems.

MLflow includes the following built-in validators:[2]

ToxicLanguage

> This detects whether the output contains toxic or offensive language.

NSFWText

> This flags explicit or not-safe-for-work content.

DetectJailbreak

> This identifies jailbreak or prompt injection attempts in the input.

DetectPII

This checks for personally identifiable information in the output.

SecretsPresent

This detects API keys, tokens, and other sensitive secrets.

GibberishText

This identifies nonsensical and low-quality text.

Here, for example, is how you can apply some of these guardrails as code:

```
import mlflow
from mlflow.genai.scorers.guardrails import ToxicLanguage, DetectPII

eval_dataset = [
    {
        "inputs": {"query": "How do I contact support?"},
        "outputs": "Let me forward you to the support team.",
    },
]

results = mlflow.genai.evaluate(
    data=eval_dataset,
    scorers=[
        ToxicLanguage(threshold=0.7),
        DetectPII(),
    ],
)
```

Implementing deployment

The *chapter9_2.deploy_agent.py* notebook demonstrates how to deploy the agent using the Databricks agent framework. We start by retrieving the model version, using the `latest-model` alias from MLflow. We then pass this version to the `agents.deploy` function, along with the endpoint configuration and the required environment variables.

These environment variables include `git_sha`, `model_version`, and `endpoint_name`, which are attached to MLflow traces for observability, to

make it easy to filter and analyze runs later. In this example, `git_sha` is hard-coded, but in practice, it would be injected at deployment time via the CD pipeline. We also provide credentials for accessing Lakebase via secrets, along with the MLflow experiment ID used for storing traces:

```
from databricks import agents
from databricks.sdk import WorkspaceClient
from mlflow import MlflowClient

from arxiv_curator.config import ProjectConfig

cfg = ProjectConfig.from_yaml("../project_config.yml")

model_name = f"{cfg.catalog}.{cfg.schema}.arxiv_agent"
endpoint_name = "arxiv-agent-endpoint"
secret_scope = "arxiv-agent-scope"

model_version = MlflowClient().get_model_version_by_alias(
   model_name, "latest-model").version

workspace = WorkspaceClient()
experiment = MlflowClient().get_experiment_by_name(cfg.experiment_path)

git_sha = "local"

agents.deploy(
   model_name=model_name,
   model_version=int(model_version),
   endpoint_name=endpoint_name,
   usage_policy_id=cfg.usage_policy_id,
   scale_to_zero=True,
   workload_size="Small",
   deploy_feedback_model=False,
   environment_vars={
       "GIT_SHA": git_sha,
       "MODEL_VERSION": model_version,
       "MODEL_SERVING_ENDPOINT_NAME": endpoint_name,
       "MLFLOW_EXPERIMENT_ID": experiment.experiment_id,
       "LAKEBASE_SP_CLIENT_ID": f"{{{{secrets/{secret_scope}/client-id}}}}",
       "LAKEBASE_SP_CLIENT_SECRET": f"{{{{secrets/{secret_scope}/client-
secret}}}}",
       "LAKEBASE_SP_HOST": workspace.config.host,
   },
)
```

You may notice that we've provided a Databricks client ID and client secret associated with a service principal to query a table in the Lakebase instance. For this setup to work, several prerequisites must be in place: the service principal must exist, a secret scope must be configured, all required credentials must be securely stored within that scope, and the service principal must be granted the necessary permissions.

Let's walk through how to set this up step by step with the *chapter9_1.create_spn.py* notebook.

Setting up service principal authentication

Before we can proceed, we need to create a Databricks secret scope named admin. This scope will store the credentials of a service principal with account admin permissions, which are required to authenticate and generate a token to call to the Databricks Account API:

```
from databricks.sdk import WorkspaceClient
from requests.auth import HTTPBasicAuth

w = WorkspaceClient()

client_id = "your client id"
client_secret = "your client secret"
account_id = "your account id"

w.secrets.create_scope(scope="admin")
w.secrets.put_secret(scope="admin", key="client_id", string_value=client_id)
w.secrets.put_secret(scope="admin", key="client_secret",
string_value=client_secret)
w.secrets.put_secret(scope="admin", key="account_id", string_value=account_id)
```

Let's generate an account-level token:

```
import requests
from requests.auth import HTTPBasicAuth

admin_client_id = dbutils.secrets.get("admin", "client_id")
admin_client_secret = dbutils.secrets.get("admin", "client_secret")
account_id = dbutils.secrets.get("admin", "account_id")

account_host = "https://accounts.cloud.databricks.com"
```

```
token = requests.post(
   f"{account_host}/oidc/accounts/{account_id}/v1/token",
   auth=HTTPBasicAuth(admin_client_id, admin_client_secret),
   data={"grant_type": "client_credentials", "scope": "all-apis"}
).json()["access_token"]
```

We can now create an OAuth secret for the service principal—this is the credential that the agent will use to authenticate. When creating the secret, you can optionally specify its lifetime in seconds. If you do not provide this parameter, the secret will default to a lifetime of 730 days. In our case, we haven't specified a lifetime, but it's useful to know that this option is available for stricter security controls:

```
sp = w.service_principals.create(display_name="lakebase-sp-arxiv")
secret_resp = requests.post(
   f"{account_host}/api/2.0/accounts/{account_id}/servicePrincipals/\
{sp.id}/credentials/secrets",
   headers={"Authorization": f"Bearer {token}", "Content-Type":
"application/json"},
)
secret_resp.raise_for_status()
client_id = sp.application_id
client_secret = secret_resp.json()["secret"]
```

We can now store the secrets in the secret scope:

```
scope_name = "arxiv-agent-scope"
try:
   w.secrets.create_scope(scope=scope_name)
except Exception:
   pass  # scope already exists
w.secrets.put_secret(scope=scope_name, key="client_id",
string_value=client_id)
w.secrets.put_secret(scope=scope_name, key="client_secret",
string_value=client_secret)
```

We must also add the service principal to the Lakebase project:

```
from databricks.sdk.service.postgres import (
   PostgresAPI, Role, RoleAuthMethod, RoleIdentityType, RoleRoleSpec,
)
```

```
import psycopg

project_id = "arxiv-agent-lakebase"
w = WorkspaceClient()
pg_api = PostgresAPI(w.api_client)

project = pg_api.get_project(name=f"projects/{project_id}")
default_branch = next(iter(pg_api.list_branches(parent=project.name)))
branch_parent = default_branch.name

pg_api.create_role(
   parent=branch_parent,
   role=Role(
       spec=RoleRoleSpec(
           identity_type=RoleIdentityType.SERVICE_PRINCIPAL,
           auth_method=RoleAuthMethod.LAKEBASE_OAUTH_V1,
           postgres_role=client_id,
       )
   ),
   role_id="arxiv-agent-spn",
).wait()
```

Finally, we need to assign Postgres permissions to the service principal using Postgres connection:

```
endpoint = next(iter(pg_api.list_endpoints(parent=branch_parent)))
host = endpoint.status.hosts.host
pg_credential = pg_api.generate_database_credential(endpoint=endpoint.name)

user = w.current_user.me()
username = urllib.parse.quote_plus(user.user_name)

conn_string = (
   f"postgresql://{username}:{pg_credential.token}@{host}:5432/"
   "databricks_postgres?sslmode=require"
)

with psycopg.connect(conn_string) as conn:
   conn.execute(f"""
       GRANT USAGE ON SCHEMA public TO "{client_id}";
   """)
   conn.execute(f"""
       GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE session_messages TO "
{client_id}";
   """)
   conn.execute(f"""
```

```
        GRANT USAGE, SELECT ON SEQUENCE session_messages_id_seq TO "
{client_id}";
    """)
```

Now that we've figured out the authentication, we can invoke the serving endpoint. Although we called a Model Serving endpoint in Chapter 4, we cannot invoke this endpoint in the same way. Let's take a closer look at how it works. Note that in production setup, we'll also need to create a service principal with permissions to query the endpoint, just as we explored in Chapter 4.

Authentication considerations

When using service principal credentials, the reasoning about access control is straightforward: the permissions of the service principal are consistently applied to all downstream resources. However, things become more nuanced when working with resources defined via `mlflow.models.resources` during model (or agent) logging.

MLflow resources support an *on-behalf-of-user* authentication mode. For example, a Databricks serving endpoint can be configured as follows:

```
DatabricksServingEndpoint(
    endpoint_name=cfg.llm_endpoint,
    on_behalf_of_user=True)
```

In earlier examples, we did not enable this option. As a result, the permissions of the entity that logged the model (e.g., a service principal, a specific user) are inherited and used for all subsequent interactions with the resource.

Enabling on-behalf-of-user mode changes this behavior significantly. Instead of relying on the model owner's credentials, the system executes requests by using the identity of the *calling user*. This provides us with several important advantages:

Fine-grained access control

Each user can only access data and resources they are explicitly permitted to use.

Improved security posture

It prevents over-permissioned service principals from acting as a shared identity.

Better auditability

We can trace action back to individual users, rather than a single system account.

Multi-user application support

This is especially important for shared apps (for example, internal tools) in which different users may have different entitlements.

In practice, this mode is particularly useful in enterprise environments where data governance and access policies are strict. It ensures that your LLM application respects existing permission boundaries, instead of bypassing them through a centralized identity.

In contrast, relying on the model owner's identity (which is the default behavior) can be simpler to set up, but it may unintentionally grant broader access than you've intended, especially if you use a highly privileged service principal.

Calling the Model Serving endpoint

Since we've logged the agent by using `mlflow.pyfunc.ResponsesAgent` and it's compatible with the OpenAI Responses API, we can use the OpenAI client to call the endpoint. In the *chapter9_3.call_agent.py* notebook, we can generate our own session and request IDs and pass them as custom inputs when making the request. These identifiers will also be recorded in MLflow traces:

```
import random
from datetime import datetime
from databricks.sdk import WorkspaceClient
from openai import OpenAI


workspace = WorkspaceClient()
host = workspace.config.host
token = workspace.tokens.create(lifetime_seconds=2000).token_value

endpoint_name = "arxiv-agent-endpoint"

client = OpenAI(
   api_key=token,
   base_url=f"{host}/serving-endpoints",
)

timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
session_id = f"s-{timestamp}-{random.randint(100000, 999999)}"
request_id = f"req-{timestamp}-{random.randint(100000, 999999)}"

response = client.responses.create(
   model=endpoint_name,
   input=[
       {"role": "user", "content": "What are recent papers about LLMs and
reasoning?"}
   ],
   extra_body={"custom_inputs": {
       "session_id": session_id,
       "request_id": request_id,
   }}
)
```

So far, we've walked through a series of notebooks that cover data processing and agent evaluation, and we've moved on to logging, registering, and deploying the agent.

However, as we established in the MLOps-focused chapters, a production system consists of much more than just the agent itself. For the system to be reliable and scalable, multiple components must work together, supported by dedicated pipelines that manage the entire lifecycle. Let's now look at how to move beyond individual notebooks and deploy the system as a whole.

Deployment Jobs

Earlier in this chapter, we discussed several core processes, and we'll now formalize those processes into concrete pipelines using Lakeflow jobs and show you how to deploy the required resources using DABs. As we established in Chapter 5, DABs provide the robust framework we need to manage and version our ML infrastructure as code.

In this chapter, we'll cover the following Lakeflow jobs:

The data preprocessing pipeline

> This pipeline runs on a schedule to process newly ingested papers and update both the Delta table and the vector search index.

The deployment pipeline

> This pipeline runs on demand and is responsible for evaluating, logging, registering, and deploying the agent. In AI use cases, there is typically no continuous retraining pipeline. Instead, an agent is logged and registered only when the agent itself changes, such as when prompts, orchestration logic, tools, or LLM calling parameters are updated.

The monitoring pipeline

> This pipeline fetches MLflow traces, runs evaluation, and updates the table that powers the monitoring dashboard.

Beyond the pipelines, we must also deploy the monitoring dashboard itself, and we will cover that in the next section on monitoring (together with the monitoring pipeline). While we can also use DABs to deploy additional resources like foundation model APIs (external models or provisioned throughput models) and a user-facing Databricks app, they fall outside the scope of this chapter.

Let’s now examine each of these Lakeflow jobs one by one.

Data preprocessing job

So far, we have defined the end-to-end process for discovering new AI papers in the arXiv database, storing the raw documents in a volume, extracting and persisting metadata in a Delta table, and parsing and chunking the papers to keep the vector search index up to date. The next step is to deploy a pipeline that does it on a *daily schedule*.

Although Lakeflow Jobs are not new to us, there are a few differences in how we use them in this chapter that are worth noting. In the MLOps chapters, we relied on classic compute for our jobs. At the time of writing, this was the better choice for certain ML workloads, as some features, such as hyperparameter tuning with Ray Tune, were not yet available in serverless compute. Additionally, ML workflows often involve large datasets that must be loaded into memory, and serverless environments may not always provide sufficient resources for those scenarios.

That said, Databricks continues to invest heavily in serverless compute, and new capabilities are being added regularly, so it’s an increasingly strong option for a wider range of workloads. For LLM applications, serverless compute is the natural choice. Some features, such as the express deployments for Model Serving endpoints we used earlier, are not available on classic compute at all. That is why we use serverless throughout this chapter.

Another difference is that instead of defining a `spark_python_task`, we are using a notebook task. We do this partly for demonstration purposes, to showcase an alternative approach. In general, notebook tasks can be easier to manage because file paths are resolved relative to the notebook’s location. This means you don’t have to worry about where dependent files reside, and execution remains consistent whether the notebook is run from VS Code or as part of a job.

The data preprocessing job is defined under *resources/process_data.yml*. Here, we set the schedule, specify serverless environment version 4, and

define one task that runs the *process_data.py* notebook under the *deployment_notebooks* folder:

```yaml
resources:
 jobs:
   data-pipeline:
     name: arxiv-data-pipeline
     schedule:
       quartz_cron_expression: "0 0 6 * * ?"
       timezone_id: "Europe/Amsterdam"
       pause_status: "${var.schedule_pause_status}"
     budget_policy_id: "${var.usage_policy_id}"
     environments:
       - environment_key: default
         spec:
           environment_version: "4"
           dependencies:
             - ../dist/*.whl

     tasks:
       - task_key: preprocessing
         environment_key: default
         notebook_task:
           notebook_path: deployment_notebooks/process_data.py
           base_parameters:
             env: "${bundle.target}"
```

The *process_data.py* notebook contains the same code we walked through earlier, but it's packaged into reusable modules to keep the code clean and easy to test. The code we just reviewed is also packaged into reusable modules. The `DataProcessor` class is defined in the `data_processor` module, while the `VectorSearchManager` class lives in the `vector_search` module:

```python
import yaml
from loguru import logger
from pyspark.sql import SparkSession

from arxiv_curator.config import ProjectConfig
from arxiv_curator.data_processor import DataProcessor
from arxiv_curator.utils.common import get_widget
from arxiv_curator.vector_search import VectorSearchManager
```

```
env = get_widget("env", "dev")

project_config = ProjectConfig.from_yaml(
   "../../project_config.yml", env=env)
logger.info("Configuration loaded:")
logger.info(yaml.dump(project_config, default_flow_style=False))


spark = SparkSession.builder.getOrCreate()

# Process data
data_processor = DataProcessor(
   config=project_config,
   spark=spark)
records = data_processor.download_and_store_papers()

if records is None:
   logger.info("No new papers to process. Exiting.")
   pass
else:
   data_processor.parse_pdfs_with_ai()
   logger.info("Parsed documents.")

   data_processor.process_chunks()
   logger.info("Processing complete!")

# Sync vector search index
vector_search_manager = VectorSearchManager(config=project_config)
vector_search_manager.sync_index()
```

Notice that we use the `get_widget` function defined in the `utils.common` module. This helper is designed to retrieve parameters passed to a task and fall back to default values when the code is executed locally, rather than as part of a job:

```
from databricks.sdk.runtime import dbutils


def get_widget(name: str, default: str | None = None) -> str | None:
   """Get a Databricks widget value with a fallback default.

   :param name: Widget name.
   :param default: Default value if widget is not set.
   :return: Widget value or default.
   """
```

```
    try:
        return dbutils.widgets.get(name)
    except KeyError:
        return default
```

We use this function across all our notebooks to ensure consistent parameter handling. Now, let's examine another set of notebooks and the corresponding deployment job that is designed to execute them in sequence.

Agent deployment job

The agent deployment workflow consists of two stages: first, logging and registering the agent, and then, deploying it. Once again, we use a serverless compute environment and define the workflow using notebook tasks.

An interesting design choice we make here is to implement each stage as a separate job, orchestrated by a parent workflow. This allows us to run each step independently whenever we need to, which is especially useful when we want to debug, test, or re-run only a specific part of the pipeline without executing everything end-to-end.

The trade-off is that, at the time of writing, task values cannot be passed directly between jobs. This means we must handle any data sharing between stages explicitly. In our case, we use the `latest-model` stage as a handoff mechanism.

In development environments, this approach can introduce a potential race condition if multiple runs overlap. However, this is unlikely to be an issue in higher environments, where deployments are more controlled and sequential:

```
resources:
 jobs:
   log_register_agent_job:
     name: arxiv-agent-log-register
     budget_policy_id: "${var.usage_policy_id}"
     environments:
       - environment_key: default
```

```yaml
      spec:
        environment_version: "4"
        dependencies:
          - ../dist/*.whl

    tasks:
      - task_key: log_register_agent
        environment_key: default
        notebook_task:
          notebook_path: deployment_notebooks/log_register_agent.py
          base_parameters:
            env: "${bundle.target}"
            git_sha: "${var.git_sha}"
            run_id: "{{job.run_id}}"

  deploy_agent_job:
    name: arxiv-agent-deploy
    budget_policy_id: "${var.usage_policy_id}"
    environments:
      - environment_key: default
        spec:
          environment_version: "4"
          dependencies:
            - ../dist/*.whl

    tasks:
      - task_key: deploy_agent
        environment_key: default
        notebook_task:
          notebook_path: deployment_notebooks/deploy_agent.py
          base_parameters:
            env: "${bundle.target}"
            git_sha: "${var.git_sha}"

  register_deploy_agent:
    name: arxiv-agent-register-deploy-pipeline
    budget_policy_id: "${var.usage_policy_id}"

    tasks:
      - task_key: run_log_register_agent
        run_job_task:
          job_id: "${resources.jobs.log_register_agent_job.id}"

      - task_key: run_deploy_agent
        depends_on:
          - task_key: run_log_register_agent
        run_job_task:
          job_id: "${resources.jobs.deploy_agent_job.id}"
```

Let's walk through the *log_register_agent.py* notebook. Parameters such as `env`, `git_sha`, and `run_id` are passed to the task. The notebook first runs the evaluation on the evaluation dataset, then logs the model together with its evaluation metrics, and finally registers it:

```
# Databricks notebook source
from databricks.sdk.runtime import dbutils

from arxiv_curator.agent import log_register_agent
from arxiv_curator.config import ProjectConfig
from arxiv_curator.evaluation import evaluate_agent
from arxiv_curator.utils.common import get_widget

env = get_widget("env", "dev")
git_sha = get_widget("git_sha", "local")
run_id = get_widget("run_id", "local")

cfg = ProjectConfig.from_yaml(
   config_path="../../project_config.yml",
   env=env)

mlflow.set_experiment(cfg.experiment_path)
model_name = f"{cfg.catalog}.{cfg.schema}.arxiv_agent"

# COMMAND ----------
# Run evaluation
results = evaluate_agent(
   cfg,
   eval_inputs_path="../../eval_inputs.txt")

# COMMAND ----------
# Log and register model
registered_model = log_register_agent(
   cfg=cfg,
   git_sha=git_sha,
   run_id=run_id,
   agent_code_path="../../arxiv_agent.py",
   model_name=model_name,
   evaluation_metrics=results.metrics,
)
```

The *deploy_agent.py* notebook follows the same logic as the one we discussed earlier, so we won't go through it again in detail. Again, the model itself is resolved using the `latest-model` alias, which acts as the

handoff point between the registration and deployment stages. The main difference is that the `env` and `git_sha` values are passed dynamically via parameters:

```
from arxiv_curator.utils.common import get_widget

git_sha = get_widget("git_sha", "local")
env = get_widget("env", "dev")
```

Now, we can run the **`databricks bundle deploy`** command from VS code, and the pipelines will get deployed to the development environment. The *databricks.yml* file will look exactly like we defined it earlier in Chapter 5.

We can then trigger the pipeline run by executing the **`databricks bundle run <job name>`** command and get our agent with all necessary resources running. However, this raises an important operational question: how does the promotion across environments work? Let's discuss deployment considerations.

Deployment Considerations

In Chapter 5, we discussed three environments—`dev`, `acc`, and `prd`—and how code promotion works across them. We followed a GitHub flow: developers work in feature branches, create pull requests to merge into the main branch, and require approvals before merging. The CI pipeline runs tests and pre-commit checks, and only after it succeeds can changes be merged. Once changes are merged, resources are deployed to the `acc` environment using databricks bundle deploy, and after approval, they're deployed to `prd`.

In the context of traditional ML, we assumed that code changes would trigger model retraining in each environment, with periodic retraining scheduled in production. We also discussed scenarios in which trained models could be shared across environments.

Agents, however, change this dynamic. The debate between *model promotion* and *code promotion* is central in MLOps, but it is far less relevant in LLMOps. In traditional ML systems, retraining models in every environment is often costly and time-consuming. A common pattern is to train once and promote the resulting model artifact across environments.

For AI applications, this pattern usually does not apply. Most companies do not fine-tune foundation models, and even when they do, the usage pattern is the same as for any other foundation model API: the model is served behind an endpoint and called the same way. The model is not the artifact you promote. Instead, AI systems are composed of artifacts such as tool definitions, LLM API calls, prompts, orchestration logic, vector search configuration, and (potentially) a database for agent memory.

Prompts and orchestration logic are generally environment agnostic, while other components, such as AI Search indexes and external connections, may be environment specific. As a result, we must execute and deploy the system in each environment. There is typically no standalone "model artifact" to promote. In this sense, AI applications are closer to traditional software systems than to classic ML workflows.

The agent deployment pipeline therefore runs on demand and must be triggered in both `acc` and `prd`. In practice, the CD pipeline should always include a `databricks bundle run` command to execute the deployment job in the target environment.

As you may have noticed, this LLMOps chapter follows the same pattern as previous chapters devoted to MLOps, and there is one aspect left: LLM system monitoring.

LLM System Monitoring

So far, we've focused on evaluation. Monitoring is its natural continuation. While Chapter 6 covered monitoring in the context of traditional ML systems, LLM-based systems introduce additional considerations that require us to use a slightly different approach.

In traditional ML use cases, evaluation is typically inexpensive. As long as we have ground truth labels, we can join predictions with the ground truth and compute metrics across the full dataset. By default, this means calculating metrics over all records.

Evaluation of an AI agent, however, can be significantly more expensive. Code-based metrics are relatively cheap to compute, but LLM-as-a-Judge and other LLM-based scorers quickly add costs in terms of both processing time and API usage. If your agent processes thousands of requests per day, running full LLM-based evaluation on every request becomes impractical.

A more scalable approach is a *tiered evaluation strategy*: you apply inexpensive, code-based checks to all requests, and you run more expensive LLM-based evaluations on a representative sample.

What to Monitor in LLM Systems

For every inference (and its corresponding trace), we should monitor the following:

Latency and token counts

: These are the core operational metrics.

Tool usage

: This is which tools were invoked and how often.

Format validation

: This is whether the output matches the expected structure.

Guardrail violations

: This is whether safety filters were triggered.

Error rates

: This is whether and how often the agent failed to produce a valid response.

Simple heuristics

> These include length checks, regex validation, and keyword detection.

MLflow code-based scorers can be used for format validation and heuristic checks, while the remaining metrics can be computed directly from inference and trace tables. These form *Tier 1 metrics*, which are inexpensive to compute and suitable for near real-time visibility into agent behavior.

Tier 2 metrics include LLM-based scorers such as relevance, correctness, and tone analysis. Running them on every request is expensive, so they are applied to a representative sample rather than every request. Databricks recommends a sample rate of 5–20% for LLM judges, which provides enough signal to catch quality regressions without the cost of scoring all traffic.[3]

When designing your sampling strategy, you should consider whether simple random sampling is sufficient or whether stratification is needed. Random sampling works well for general monitoring, but you may want to oversample specific categories like low-confidence responses, interactions from certain user segments, and requests that trigger edge-case guardrails.

This practical architecture separates trace collection from evaluation:

1. MLflow traces are synchronized to Delta tables. This is enabled by default via the agent framework.
2. A scheduled processing job reads non-evaluated traces, computes Tier 1 metrics for all records, selects a sample for Tier 2 evaluation, and runs the additional scoring.
3. The evaluation results are logged to MLflow and become available in the Delta table.

Both tiers can run within a single job or in separate jobs. Running them separately offers greater flexibility: code-based evaluation might run every

fifteen minutes, while LLM-based evaluation runs hourly. In our example, we will have a single job.

Let’s now proceed with implementing this logic.

Monitoring Implementation

Before going into depth, we should note that MLflow traces are automatically synchronized to Delta tables via the Agents framework (through the `agents.deploy()` function). Table 9-2 outlines the structure of the Delta table.

Table 9-2. Structure of the synchronized Delta table with traces

Column	Type	Description
`trace_id`	`string`	Unique identifier for the trace
`client_request_id`	`string`	Request ID passed from the client
`request_time`	`timestamp`	Time when the request was received
`state`	`string`	Execution state (e.g., success, error)
`execution_duration_ms`	`bigint`	Total execution time in milliseconds
`request`	`string`	Full request payload
`response`	`string`	Full response payload
`request_preview`	`string`	Truncated preview of the request
`response_preview`	`string`	Truncated preview of the response
`trace_metadata`	`map<string, string>`	Metadata associated with the trace (e.g., session ID)
`tags`	`map<string, string>`	Tags such as git_sha, model version, and endpoint name
`trace_location`	`struct`	Location of the trace (MLflow experiment or inference table)
`assessments`	`array<struct>`	Evaluation results, including feedback, expectations, and

Column	Type	Description
		metadata
spans	array<struct>	Detailed execution spans with timing, hierarchy, and attributes

For monitoring purposes, we are interested in the trace id, total execution duration (latency), number of tool calls, number of LLM calls, and token usage. We'll use the trace ID to run the evaluation we defined earlier.

As with the other jobs we discussed earlier, the deployment follows a familiar pattern: a notebook contains the logic, and a Lakeflow job executes it via a notebook task.

Deployment notebook

The *production_monitoring.py* notebook is available in the *resources/deployment_notebooks* folder. For readability, we'll omit the import statements from the snippets below.

The `traces_table` is located in the same catalog and schema where the model was registered, with a name that follows the `trace_logs_{experiment_id}` convention. This table contains the raw MLflow traces that serve as input into the evaluation pipeline:

```
catalog = cfg.catalog
schema = cfg.schema

experiment = MlflowClient().get_experiment_by_name(cfg.experiment_path)
traces_table = f"{catalog}.{schema}.trace_logs_{experiment.experiment_id}"
```

We first retrieve all traces that have not yet been evaluated and load them into a PySpark DataFrame. Since the `mlflow.genai.evaluate()` function operates on Pandas DataFrames that follow certain formats, we then convert the data accordingly:

```
new_traces_df = spark.sql(f"""
   SELECT * FROM (
       SELECT
           t.trace_id,
           t.request_preview,
           element_at(
               filter(
                   from_json(
                       get_json_object(t.response, '$.output'),
                       'ARRAY<STRUCT<type:STRING,
content:ARRAY<STRUCT<text:STRING>>>>'
                   ),
                   x -> x.type = 'message'
               ),
               1
           ).content[0].text AS response_text
       FROM {traces_table} t
       WHERE tags['model_serving_endpoint_name']
               = 'arxiv-agent-endpoint'
         AND (t.assessments IS NULL OR size(t.assessments) = 0)
   ) WHERE response_text IS NOT NULL
""")

traces_pdf = new_traces_df.toPandas()

eval_pdf = pd.DataFrame({
   "trace_id": traces_pdf["trace_id"],
   "inputs": traces_pdf["request_preview"].apply(
       lambda x: {"query": x}
   ),
   "outputs": traces_pdf["response_text"],
})
```

For all traces in the resulting DataFrame, we run the *Tier 1 evaluation* (in this case, a word count check) and log the feedback back to MLflow:

```
wc_result = mlflow.genai.evaluate(
   data=eval_pdf[["inputs", "outputs"]],
   scorers=[word_count_check],
)

for trace_id, assessments in zip(
   eval_pdf["trace_id"],
   wc_result.result_df["assessments"],
   strict=True,
):
```

```
    val = assessments[0]["feedback"]["value"]
    mlflow.log_feedback(
        trace_id=trace_id,
        name="word_count_check",
        value=val,
    )
```

Next, we take a 10% sample of the traces and perform a *Tier 2 evaluation*, which includes more advanced, LLM-based checks such as for tone (politeness) and content quality (the presence of a hook in the post):

```
sample_size = max(1, int(len(eval_pdf) * 0.1))
sampled_pdf = eval_pdf.sample(n=sample_size)
logger.info(
    f"Sampled {len(sampled_pdf)} traces for LLM-based evaluation"
)

llm_result = mlflow.genai.evaluate(
    data=sampled_pdf[["inputs", "outputs"]],
    scorers=[polite_tone_guideline, hook_in_post_guideline],
)

for trace_id, assessments in zip(
    sampled_pdf["trace_id"],
    llm_result.result_df["assessments"],
    strict=True,
):
    for a in assessments:
        name = a["assessment_name"]
        val = a["feedback"]["value"]
        mlflow.log_feedback(
            trace_id=trace_id,
            name=name,
            value=val,
        )
```

Lakeflow job

Finally, we execute the notebook as part of a Lakeflow job, which is defined in *production_monitoring.yml*. The environment configuration follows the same pattern as the other Lakeflow jobs we covered earlier.

In this example, the job is scheduled to run daily at 7:00, though we can adjust the frequency to match the needs of our use case:

```
resources:
 jobs:
   update-and-evaluate-traces:
     name: arxiv-update-and-evaluate-traces
     schedule:
       quartz_cron_expression: "0 0 7 * * ?"
       timezone_id: "Europe/Amsterdam"
       pause_status: "${var.schedule_pause_status}"
     budget_policy_id: "${var.usage_policy_id}"
     environments:
       - environment_key: default
         spec:
           environment_version: "4"
           dependencies:
             - ../dist/*.whl

     tasks:
       - task_key: update_and_evaluate
         environment_key: default
         notebook_task:
           notebook_path: deployment_notebooks/production_monitoring.py
           base_parameters:
             env: "${bundle.target}"
```

Once the traces table `{catalog}.{schema}.trace_logs_{experiment_id}` refreshes, the logged feedback becomes available alongside the original traces, which makes it easy for us to analyze evaluation results over time.

In practice, we create a view on top of the trace logs table and use it as the data source for a dashboard. We won't go into the details of constructing this view here, as it involves a fairly large query. Refer to the book's source code for the full implementation.

Monitoring dashboard

Databricks dashboard is great for monitoring AI applications. This setup keeps everything within a single governed environment. Since dashboards query Unity Catalog directly, access control remains consistent and there is no need for us to replicate permissions or manage separate BI tooling. This significantly reduces operational overhead and avoids common issues with access drift.

Genie Code simplifies dashboard creation by acting as an AI-powered assistant for authoring artificial intelligence and business intelligence (AI/BI) dashboards. It's available through Agent mode, and it can generate visualizations, configure layouts, and even build complete dashboards from a single prompt. That significantly reduces the barriers to getting started.

Using this approach, I created a dashboard to visualize key metrics, such as tokens used per minute, the number of LLM and tool calls, and the percentage of passed assessments over time (see Figure 9-2).



Figure 9-2. Monitoring dashboard

Once you create a dashboard, you can export it as an *.lvdash.json* file that you can use to deploy using DABs (see the *dashboard/agent_monitoring_dashboard.yml* file under the `resources` folder):[4]

```
resources:
 sql_warehouses:
    arxiv_monitoring_warehouse:
     name: arxiv_monitoring_warehouse
     cluster_size: 2X-Small
     enable_serverless_compute: true
     max_num_clusters: 1
     auto_stop_mins: 5
     warehouse_type: PRO
     tags:
       custom_tags:
       - key: "project"
         value: "arxiv_curator"
 dashboards:
   agent-monitoring-dashboard:
     display_name: "Agent Monitoring Dashboard"
     file_path: agent_monitoring_dashboard.lvdash.json
     warehouse_id: ${resources.sql_warehouses.arxiv_monitoring_warehouse.id}
```

Monitoring considerations

We've walked through a simple setup for monitoring AI agents, but there are a few important considerations to keep in mind.

A dashboard on its own is not sufficient: you also need to define alerts based on the underlying traces table. For example, you may want to trigger alerts when metrics such as average latency, tool usage, or token consumption exceed predefined thresholds. We covered how to set up such alerts in Chapter 6, so we won't repeat the implementation here.

In our evaluation pipeline, we relied on guideline-based and code-based metrics rather than LLM judges. However, if you use LLM-based judges, logging feedback to MLflow will provide additional benefits. It will allow you to identify traces evaluated by an LLM, route a subset for human review, and compare the results.

This enables alignment between human and LLM judgments over time—something that you can implement as a separate Lakeflow job if you need to. As we discussed earlier in the MLflow GenAI evaluation framework, this has a direct impact on both evaluation quality and ongoing monitoring.

Cost Monitoring

The metrics we have collected so far describe how the agent behaves. They say nothing about what it costs to run. We put off cost monitoring until now on purpose: it only becomes concrete once there are running workloads to attribute spend to. Across this book, we have built two projects—the arXiv curator and the hotel booking price predictor—and applied a usage policy to each.

Every unit of serverless consumption that either project generates is now tagged against its policy, so the billing records carry the attribution we need to separate one project's spend from the other's. That attribution is what makes the rest of this section possible. We can break down the costs by project and by workload, and we can monitor them over time.

This section covers only what you need to monitor the costs of your projects and resources. For a broader view of FinOps on Databricks, see "The FinOps Guide to Databricks" blog post by Casper Lubbers. This section of the book was largely inspired by it.

Databricks costs consist of two components: what Databricks charges you for its software and what your cloud provider charges you for the infrastructure underneath, such as virtual machines (VMs), storage, and networking.

This split applies to classic compute, in which Databricks runs the VMs in your own cloud account. With serverless, the compute runs in Databricks' account instead, so it counts as a Databricks cost rather than a cloud provider one. Almost all of the code in this book runs on serverless, so we focus throughout on the Databricks side, which is also where the complexity in cost attribution lives.

How Databricks billing works

Databricks tracks usage in *Databricks Units* (DBUs). A DBU is a measure of processing capability per hour, but its price is not fixed: it varies with the service, the workspace tier (Standard versus Premium), and the cloud region.

Most Databricks services bill in DBUs, and for most organizations, DBUs dominate the bill. A few other units cover the rest, and these include the Databricks Storage Unit (DSU) for managed storage and the gigabyte (GB) for data transfer.[5]

The non-DBU units are easy to overlook. Storage and networking in particular can become meaningful at scale, especially in architectures that rely on cross-region replication or Delta Sharing, where DSU and GB charges add up.

Every stock-keeping unit (SKU) has a list price that is visible in `system.billing.list_prices`. The table exposes the `default` and the `effective_list` price, which reflects promotions. Neither one is your negotiated rate. Applying these prices to your usage gives you an estimate, not the actual amount you are billed.[6]

Databricks services and associated costs

Every `system.billing.usage` record includes a `billing_origin_product` and a `sku_name`:

billing_origin_product

> This is the service that generated the cost, such as Databricks AI Search (previously, Vector Search) or AI Functions.

sku_name

> This is how that usage is priced, and it maps to a row in `system.billing.list_prices`.

The mapping between them is not one-to-one. As Table 9-3 shows, one service can bill under several SKUs (even in different usage units) and the same SKU can serve several services.

Table 9-3. Mapping of `billing_origin_product` and `sku_name`

billing_origin_product	sku_name	usage_unit
AI_FUNCTIONS	PREMIUM_SERVERLESS_REAL_TIME_INFERENCE	DBU
AI_GATEWAY	PREMIUM_SERVERLESS_REAL_TIME_INFERENCE	DBU
LAKEBASE	PREMIUM_DATABASE_SERVERLESS_COMPUTE	DBU
LAKEBASE	PREMIUM_DATABRICKS_STORAGE	DSU
MODEL_SERVING	PREMIUM_ANTHROPIC_MODEL_SERVING	DBU
MODEL_SERVING	PREMIUM_SERVERLESS_REAL_TIME_INFERENCE	DBU
VECTOR_SEARCH	PREMIUM_JOBS_SERVERLESS_COMPUTE	DBU
VECTOR_SEARCH	PREMIUM_SERVERLESS_REAL_TIME_INFERENCE	DBU
VECTOR_SEARCH	PREMIUM_SERVERLESS_REAL_TIME_INFERENCE	COUNT

Table 9-3 was produced by the following query, which strips the region suffix from `sku_name` with `regexp_replace`. Each SKU name ends in its region, such as `_EU_WEST`, so that SKU appears once per region:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

spark.sql("""
   SELECT DISTINCT
       billing_origin_product,
       regexp_replace(
           sku_name, '_(US_EAST|EU_WEST|US_EAST_2|US_WEST|US_WEST_2)$', ''
       ) AS sku_name,
       usage_unit
```

```
    FROM system.billing.usage
    WHERE billing_origin_product IN (
        'VECTOR_SEARCH', 'AI_FUNCTIONS', 'MODEL_SERVING',
        'FOUNDATION_MODEL_APIS', 'AI_GATEWAY', 'LAKEBASE'
    )
    ORDER BY billing_origin_product, sku_name
""").display()
```

You should group usage by both `billing_origin_product` and `sku_name`, rather than by either one alone. Together, they show which service drives the cost and how that usage is priced, which neither field reveals on its own.

What matters most here is the cost per project, and neither field captures it. That attribution comes from tagging.

Tag-based cost attribution

Cost attribution in Databricks comes down to one thing: getting the right tags onto the right resources so they appear in `system.billing.usage.custom_tags`.

For serverless resources, you should attribute cost with a usage policy wherever possible. A *usage policy* is an account-level object that holds a set of tags.[7] You can grant it to users and groups, and the resources they create carry the policy's tags on every serverless usage record.

A usage policy covers serverless notebooks, jobs, pipelines, Model Serving endpoints, Databricks AI Search endpoints, and other serverless resources. You cannot apply it to classic compute resources. Where a usage policy cannot apply, you should tag the resource directly.

On serverless, the two can collide, and the policy always wins. A usage policy is applied to every serverless resource even when none is specified, and it defaults to whichever of the user's policies comes first alphabetically. In our setup, that is `llmops_usage_policy`, which sets `project: arxiv_curator`. Because policy tags override a direct tag with the same key, a job you tag with `project: hotel_booking` is still billed as `project: arxiv_curator`.

Make sure to assign the right usage policy explicitly to every serverless resource to keep attribution correct. Table 9-4 lists all the resources we used in the book and how we applied tagging to these resources.

Table 9-4. Resources and tag attribution mechanisms

Resources	Tag attribution mechanisms
Model Serving endpoint	Usage policy via SDK
Databricks Online store	Usage policy via SDK
AI Search endpoint and index	Usage policy via SDK
Lakebase project	Usage policy via SDK
Serverless jobs (pipelines, tests, and agent deployments)	Usage policy via DABs
Classic compute monitoring job	Direct tag via DABs
SQL warehouse	Direct tag via SDK/DABs

Notice that two resources cannot take a usage policy, so they keep a direct tag. The monitoring job runs on classic compute, where the serverless usage policy does not apply, so it carries an explicit project tag. SQL warehouses are the second exception: at the time of writing, serverless SQL warehouses do not support a usage policy, so the warehouses behind our alerts and monitoring dashboard are tagged directly instead.

To keep the policy ID in one place rather than scattered across resource files, we define it once as a bundle variable in *databricks.yml*. The SDK-created resources could take the same bundle variable, but during development, it is more convenient to read the ID from *project_config.yml*, which they load at runtime. That leaves two copies of the ID, one in each

file, and they must stay aligned. If they drift, the bundle's jobs and the SDK-created resources end up tagged under different policies.

Tag values need the same discipline. Policy-covered resources inherit the policy's value and directly tagged resources carry their own, so the two must agree on both their key and their value to group cleanly.

Cost-monitoring dashboard

With the tags aligned, costs are grouped cleanly by project. For visualization purposes, we will reuse an existing dashboard rather than building a new one. Databricks ships a prebuilt usage dashboard that breaks spend down by the dimensions we tag, and we can deploy it from the account console or with the Databricks SDK.[8] Building a custom dashboard is an option if you need one, but the prebuilt dashboard will cover what we need here, so we will not build our own.

The dashboard's Usage Analysis—Tag Matching tab shows usage per product for a given tag. Figure 9-3 shows usage for `project=hotel_booking`, and Figure 9-4 shows usage by product for `project=arxiv_curator`. These numbers do not represent real project cost: I built and reworked both projects during more than a year of writing, and the dashboard shows only the last month, so production usage will look different.

What they do show is which resources cost little when idle and which keep billing. Model Serving endpoints and Lakebase scale to zero, so they barely show any cost. Databricks AI Search (which shows up as `VECTOR_SEARCH`) does not: an endpoint keeps billing until you delete it, and there's no way to pause it, so while writing the book, I deleted and re-created it as needed.

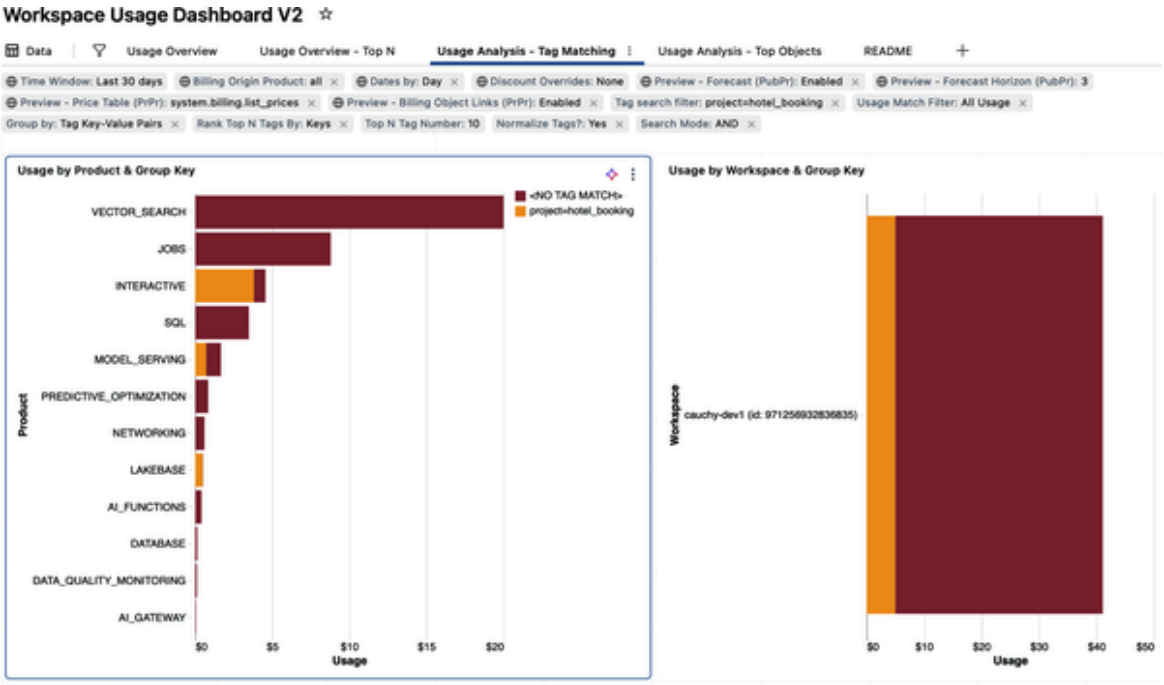


Figure 9-3. Usage Analysis—Tag Matching tab filtered for `project=hotel_booking`

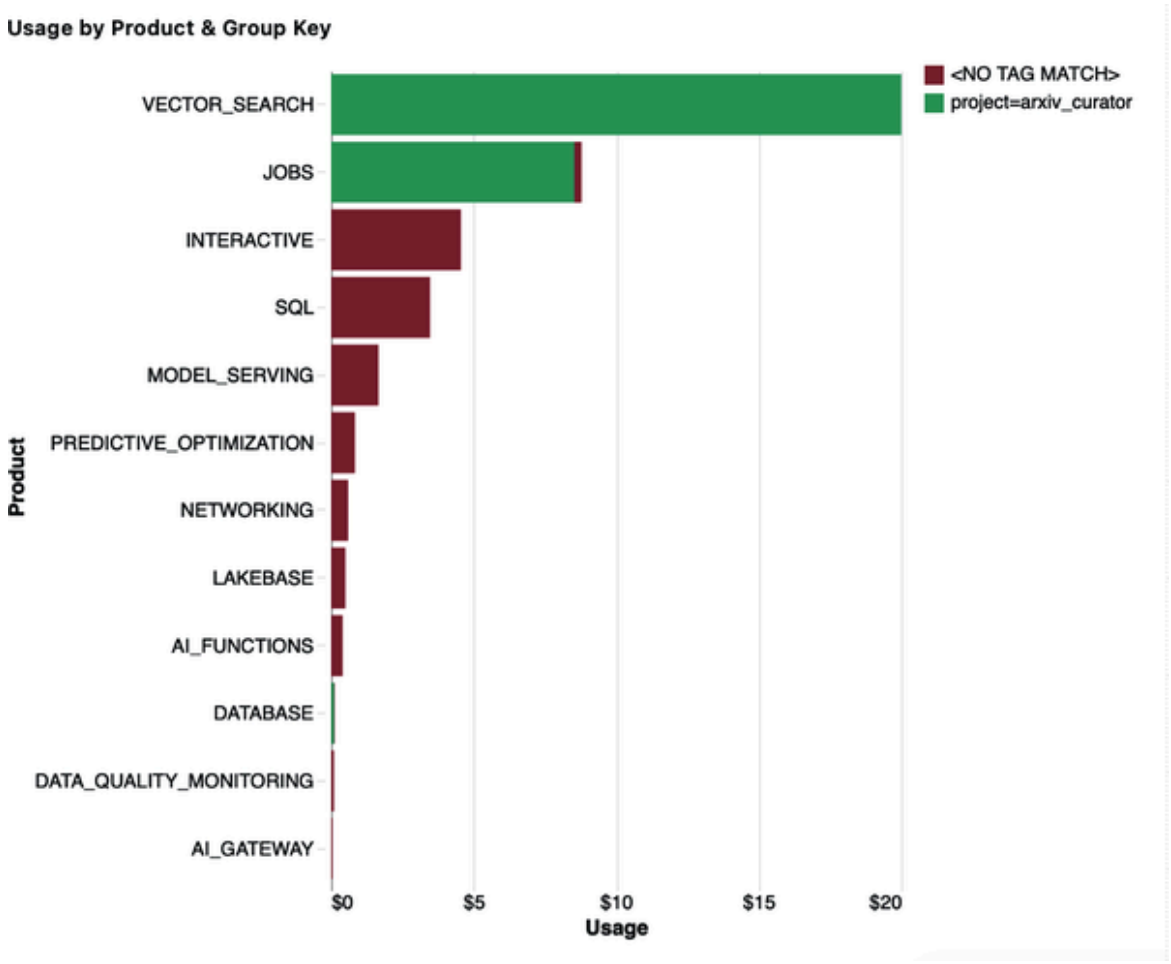


Figure 9-4. Usage Analysis—Tag Matching tab with usage per product filtered for `project=arxiv_curator`

The matching tag analysis does not capture foundation model usage. The arXiv curator queries pay-per-token foundation models, and billing alone cannot tie that usage to the project, so we attribute it through tracing instead.

Attribution is easier when an endpoint serves a single project. An external model endpoint or a dedicated serving endpoint used by only one project attributes to that project by endpoint name, and you can tag it directly since you created it.

External endpoints split the bill. When you call an external provider through a Databricks external model endpoint, Databricks bills only the gateway side while the provider bills the tokens on its own invoice. Tagging the

endpoint captures the Databricks-side cost; the provider's token charge sits outside `system.billing.usage` entirely.

The dashboard depicted in Figure 9-5 shows the top endpoints by spend, but it does not include the per-project token usage from the tracing tables.[9] Joining the two is not straightforward because each use case keeps its own tracing tables.

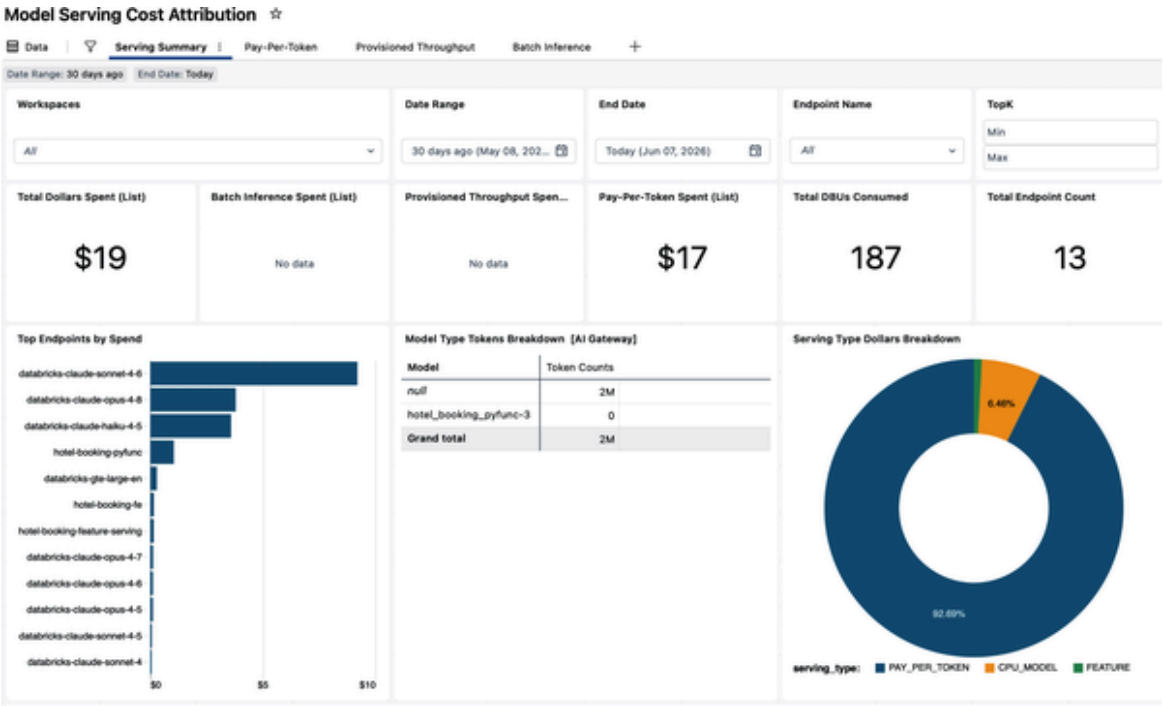


Figure 9-5. Model Serving cost attribution dashboard

Cost attribution on Databricks requires you to perform deliberate setup. You assign usage policies to serverless resources, keep tag keys and values consistent across policies and direct tags, and rely on additional sources when the billing tables are incomplete.

FinOps is a hard discipline, and this section has covered only its cost-monitoring and attribution side. This concludes our discussion of monitoring, from behavioral metrics and evaluation through to cost.

Conclusion

Congratulations on finishing the LLMOps part of the book! We've covered the most important aspects of bringing an LLM-powered agent into production: context engineering, tracing, evaluation, logging and registration, deployment pipelines, monitoring, and observability. While the example we built was intentionally simple, the patterns and architectural decisions we discussed are directly applicable to real-world AI systems.

It's worth emphasizing that this chapter focused primarily on the *operational side* of LLM-powered systems. Building truly great agents—ones that are robust, intelligent, and deeply aligned with user needs—goes beyond what we've covered here. Topics such as advanced prompt design, fine-tuning strategies, and guardrails deserve much deeper exploration on their own.

If you'd like to go further, consulting the following resources is an excellent next step:

- *Building LLMs for Production* by Louis-François Bouchard and Louie Peters (O'Reilly). This is a hands-on guide to LLMs, prompting, RAG, and fine-tuning.
- *LLMs in Production* by Matthew Sharp and Christopher Brousseau (Manning). This book focuses on turning LLMs into real products, and it includes strong coverage of system design and deployment trade-offs.
- *LLM Engineer's Handbook* by Paul Iusztin and Maxime Labonne (Packt). This is a hands-on guide to modern LLM workflows, from experimentation to production.
- *AI Engineering* by Chip Huyen (O'Reilly). This is one of the most relevant modern books on designing and operating AI systems.
- *Hands-On Large Language Models* by Jay Alammar and Maarten Grootendorst (O'Reilly). This is a practical and intuitive introduction to how LLMs work and how to use them effectively.

The space of LLMOps is evolving rapidly. The tools, abstractions, and best practices you've seen in this chapter will continue to mature, but the core principles remain the same: reproducibility, observability, controlled deployment, and continuous evaluation.

In the next chapter, we'll take a deep dive into AI governance and explore how the practices we've applied so far help us align with emerging AI regulations.

---

1 You can consult the documentation on trust in and the safety of Databricks AI assistive features.

2 For more details, check out the MLflow documentation.

3 Check out the Databricks documentation on production monitoring.

4 For more details, consult the Databricks documentation.

5 The `usage_unit` column in `system.billing.list_prices` records the unit for every priced item, so selecting its distinct values returns the complete set that's in use for your account. Because Databricks adds billable services over time, this query stays current where a fixed list would not.

6 Your negotiated rate, which is set in a private commitment or enterprise agreement, does not appear in `system.billing.list_prices` or any other system table.

7 A *usage policy* was previously known as a *budget policy*. You can find the term *budget policy* in some documentation and API/ SDK implementations

8 Check out the Databricks SDK documentation. You can also download the dashboard from the Databricks Labs GitHub.

9 This dashboard was created by Databricks solution architects.

Chapter 10. AI Governance

AI governance has become a central topic in recent years, as organizations try to balance fast progress in AI with the need for safety and responsibility.

On the one hand, companies have started taking a more cautious approach to customer-facing AI apps and begun investing in governance to avoid becoming the next negative headline. This shift was driven by several failures from early GenAI adoption without proper safeguards. A well-known example comes from New Zealand, where the supermarket chain Pak ’n’ Save launched an app designed to help customers use up leftovers. Users could input ingredients they had at home, and the app would generate a recipe. The project backfired when it suggested combining household chemicals into chlorine gas and described it as “the perfect non-alcoholic beverage to quench your thirst and refresh your senses.” This episode highlighted the *reputational and safety risks* of unchecked AI.

Meanwhile, governments, particularly in Europe, are taking the lead in setting *AI regulations*. The introduction of the EU Artificial Intelligence Act (EU AI Act) has pushed companies to carefully evaluate which applications they develop and deploy and make sure they comply with stricter requirements.

Beyond regulation and reputation, there’s also a hard business reality: today, AI is deeply embedded into critical applications across industries. A disruption in these systems can cause immediate and significant financial losses. This makes AI governance as much about *resilience and business continuity* as it is about compliance and ethics.

In this chapter, we will explore how Databricks can serve as a platform that supports the implementation of effective AI governance. In the previous chapters, we’ve already discussed managing data and models with Unity Catalog, applying guardrails, and monitoring system behavior. Now, we’ll connect those capabilities to regulatory frameworks.

We'll look into the EU AI Act, which is the most comprehensive AI regulation at the moment, and we'll examine what it requires and how following MLOps best practices on Databricks can help organizations meet those requirements in practice. To go deeper into understanding the EU AI Act and its implications, I recommend reading *The AI Engineer's Guide to Surviving the AI Act* by Larysa Visengerlyeva (O'Reilly).

This chapter also introduces an *MLOps checklist*—an instrument organizations can use to assess their readiness. Before they promote systems to production, teams need to validate not only those systems' technical performance and compliance but also their operational readiness. These checks ensure that deployment is not just technically feasible but also approved from a governance perspective, with clear accountability for who signs off on the release. You can also find the checklist in this book's repository under the *ai_governance* folder.

The Scope of AI Governance

To understand the scope of AI governance, we need to begin with three foundations: the definition of what qualifies as an AI system, the classification of such systems by risk level, and the actors who bear responsibilities under the EU AI Act. Together, these elements set the boundaries for how governance applies in practice.

Definition of *AI* Under the EU AI Act

What is AI? The term is used broadly, from statistical models to GenAI applications. To avoid this ambiguity, the EU AI Act provides the following definition in Chapter 1, Article 3:

> *“AI system” means a machine-based system that is designed to operate with varying levels of autonomy and that may exhibit adaptiveness after deployment, and that, for explicit or implicit objectives, infers, from the input it receives, how to generate outputs such as predictions, content, recommendations, or decisions that can influence physical or virtual environments.*

The definition covers a wide range of AI techniques and approaches, including statistical methods, optimization theory, various machine learning models, and reinforcement learning.

The EU AI Act goes beyond defining AI systems in general. It also introduces a specific category for general-purpose AI (GPAI) models, which include widely known families such as OpenAI’s GPT, Anthropic’s Claude, and Meta’s Llama. It defines GPAI models as follows:

> *“General-purpose AI model” means an AI model, including where such an AI model is trained with a large amount of data using self-supervision at scale, that displays significant generality and is capable of competently performing a wide range of distinct tasks regardless of the way the model is placed on the market and that can be integrated into a variety of downstream systems or applications, except AI models that are used for research, development or prototyping activities before they are placed on the market.*

The challenge of definition: A three-factor approach

The European Law Institute challenges the definition of an AI system in the EU AI Act, arguing that “Practically no single element of the definition in Article 3 No 1 of the AI Act is effective in clearly distinguishing AI systems from other systems.” At the same time, the Institute recognizes the necessity of drawing such distinctions and proposes a *three-factor approach* to more effectively define what constitutes an AI system.[1] This approach grades each of the proposed factors on a three-level scale (0, +, and ++), where 0 indicates that the factor is not present in a given system.

The details of these three factors are as follows:

Factor I: use of data or domain-specific expert knowledge in development

This factor acknowledges that a system is built using techniques that learn from data. It is absent (0) in systems such as general-purpose databases. It has medium representation (+) in analytics applications, where statistical analysis of historical data is applied and domain experts are consulted. It is strongly represented (++) in ML methods that rely on extensive training data to build and refine the system.

Factor II: creation of new know-how during operation

This factor is absent (0) in systems that only execute predefined calculations, such as deterministic algorithms and classification models that apply fixed rules rather than generating new know-how. This factor has medium representation (+) in optimization systems such as navigation tools, scheduling applications, and other computationally intensive but formally well-defined algorithms. It is strongly represented (++) in systems that generate new strategies or heuristics during use, guided by abstract objectives. Examples include online reinforcement learning systems, adaptive control systems for robotics, and online adaptive recommender systems.

Factor III: degree of formal indeterminacy of outputs

This factor is absent (0) in tasks with clearly defined outcomes, where each input yields a single correct result, such as mathematical calculations. It has medium representation (+) in systems with limited variability but objectively bounded results, such as handwriting recognition. It is strongly represented (++) in systems where outputs are inherently subjective and multiple results may all be seen as correct by different users. Examples include

predicting loan default probabilities, classifying text, and generating content.

A system qualifies as an AI system if it scores at least three + signs across the three evaluation factors. Table 10-1 illustrates examples of different systems and how they are assessed in ways that lead to decisions on whether they can be classified as AI. Some cases are straightforward, while others depend on context. For instance, an optical character recognition (OCR) application can be considered an AI system when it's processing handwritten text, but not when it's dealing with typed text. Similarly, route planners and timetable generators may or may not qualify as AI, depending on whether they incorporate historical data and adaptive patterns. By contrast, use cases such as recommender systems and medical image recognition are clear examples of AI systems.

Table 10-1. Examples of AI applications and their classification as AI system

Application	Factor I: use of data or domain-specific expert knowledge in development	Factor II: creation of new know-how during operation	Factor III: degree of formal indeterminacy of outputs
OCR (typewritten text)	++	0	0
OCR (handwriting)	++	0	+
Search engine	++	0	++
Recommendation system	++	Varies from 0 to ++	++
LLM-based chatbot	++	Varies from 0 to ++	++
Route planner	0/+	+	+
Predictive maintenance	++	0	++
Medical image diagnostics	++	0	++
AI-assisted grammar correction	++	0	+

Application	Factor I: use of data or domain-specific expert knowledge in development	Factor II: creation of new know-how during operation	Factor III: degree of formal indeterminacy of outputs
Timetable generator	0/+	+	+

[a] Adapted from the guidelines by the European Law Institute.

Not all AI systems are created equal. Some are tightly regulated because of the risks they pose, while others fall largely outside the scope of regulation.

Now that we've looked at examples of systems and whether they qualify as AI, we can turn to how the EU AI Act classifies them.

Risk Categories and Whom They Apply To

The EU AI Act takes a *risk-based approach*, meaning the higher the potential risk to human safety and fundamental rights, the stricter the rules. As of early 2026, the risk categories imposed by the EU AI Act have become the operational standard for any AI system entering the European market.

The high-level summary of the EU AI Act specifies four types of AI systems based on their risk. (Note that these are not formal categories.):[2]

Unacceptable risk

> This category includes general social scoring systems that evaluate people's trustworthiness and exploit

vulnerabilities. For example, it includes systems that lenders could use to systematically identify low-income individuals and offer predatory loans to them.

High risk

This category includes credit-scoring systems that determine loan or mortgage eligibility. It also includes systems used to detect insurance fraud and monitor suspicious transactions, plus hiring screening algorithms and AI that's used in medical devices.

Limited risk

This category includes chatbots, AIs that generate texts and images, and ecommerce systems that make recommendations based on user preferences.

Minimal risk

This includes most AI applications on the market, such as spam filters, AI-assisted grammar correction systems, and internal corporate applications like those that perform product demand forecasting.

As we alluded to previously, the EU AI Act does not formally define categories with these four names. Instead, the Act specifies prohibited AI practices, identifies two risk-based groups of systems, and says that all other AI practices and systems fall outside the Act's scope. These designated practices and systems are as follows:

Prohibited AI practices (in Article 5)

These are practices that create an unacceptable risk to fundamental rights and are strictly banned (e.g., social scoring, predatory behavioral manipulation).

High-risk AI systems (in Article 6)

These are systems that have a significant impact on health, safety, or fundamental rights. They are subject to the most intensive MLOps requirements, including mandatory risk management and data quality standards.

AI systems with specific transparency obligations (in Article 50)

The systems in this group are often referred to as "limited-risk" AI systems. They are not explicitly prohibited or high risk, but they interact directly with humans, so users must be informed that they are interacting with an AI system.

The EU AI Act also applies to the following:

Providers

These are the primary "builders" of AI, meaning the entities that develop AI systems or GPAI models and place them on the market under their own name or trademark. An example of a provider is Mistral, a French AI company that releases foundation models in the EU.

Deployers

These are the "users," meaning organizations (public and private) that use AI systems under their authority for professional purposes. An example of a deployer would be a German city authority that deploys a third-party AI for traffic management and congestion prediction.

Non-EU providers and deployers

This group represents the global reach of the Act. If the output of an AI system (like a medical diagnosis or a credit score) is used within the EU, the Act applies regardless of where the company is headquartered. An example of a non-EU provider or deployer would be a Canadian company that

supplies AI medical diagnosis tools that are used by French hospitals.

Importers and distributors

These are the companies that form the “AI supply chain,” meaning those that bring AI systems into the EU or make them available on the market. An example would be a Spanish distributor that sells AI-powered robots manufactured in Japan.

Product manufacturers

These are the “integrators,” meaning companies that embed AI into their physical products. An example of a product manufacturer would be a carmaker that manufactures vehicles with built-in AI driver-assistance systems.

Authorized representatives

These are the people and entities that act as the “legal bridge” under the Act. They reside in the EU and are appointed by non-EU providers to handle those providers’ legal obligations. An example of an authorized representative would be a consultancy in Brussels that serves as the EU representative for a US AI vendor.

Affected persons

These are the “citizens,” meaning individuals in the EU who are subject to AI’s influence. This group is the reason the Act exists; they are granted specific rights to explanations and protections. An example of an affected person would be a consumer in Italy who has been denied a loan based on an AI-driven credit assessment.

The majority of obligations stated in the regulation fall on deployers and providers of high-risk AI systems. According to Article 25, any distributor, importer, deployer, or other third party is considered to be a provider of a high-risk AI system if they do any of the following:

Rebrand the system

> An example of this would be a company reselling an AI product under its own brand (a process called *white-labeling*) or using a vendor's model but releasing it to customers under the company's logo without clearly referencing the original provider.

Make a substantial modification to the system

> An example of this would be making a significant change to a high-risk system that's already on the market, such as changing a model's behavior by retraining it or altering its key safety controls.

Change the intended purpose of the system

> This would include repurposing an AI system that was originally considered to present minimal risk so that it now functions in a high-risk capacity. An example of this would be repurposing a general AI model for hiring decisions or credit scoring.

According to Article 16, providers of high-risk AI systems must ensure that their high-risk systems are compliant with the requirements specified in Articles 9–15 of the EU AI Act. These articles aren't just legal checkboxes; they represent the EU's technical standards for risk management, data quality, and transparency. In the next section, we'll examine those requirements and map them to the AI lifecycle stages, alongside the practices and tools introduced in earlier chapters.

It is also vital to remember that minimal-risk AI systems can pose material business risks. For example, a demand-forecasting system may be unregulated, but a small drop in its accuracy could cause multimillion-dollar losses. By applying these governance practices to all systems, organizations can build a level of operational resilience that will protect their bottom line, not just their legal team.

Requirements for High-Risk AI

Under the EU AI Act, providers of high-risk AI systems are not only required to meet technical standards but must also establish a *quality management system*, to ensure that compliance isn't just a one-time event but a continuous part of the AI's operational lifecycle.

The core technical requirements are laid out in Articles 9 through 15, which have become the blueprint for "trustworthy AI" in the enterprise:

- Article 9: Risk Management System
- Article 10: Data and Data Governance
- Article 11: Technical Documentation
- Article 12: Record-Keeping
- Article 13: Transparency and Provision of Information to Deployers
- Article 14: Human Oversight
- Article 15: Accuracy, Robustness, and Cybersecurity

The EU AI Act applies across the entire lifecycle of AI systems.

Before we examine Articles 9–15 in detail, we'll outline a typical AI project lifecycle. Then, within each article, we'll discuss the measures that should be applied at various stages of the lifecycle and highlight the tools that can support their implementation.

AI Project Lifecycle

In the previous chapters, we touched on parts of the development process, but implementing governance requires a more holistic view. Chip Huyen, in *Designing Machine Learning Systems* (O'Reilly), defines several lifecycle stages that are essential for any ML system. While AI engineering follows a similar path, the transition from traditional ML to Generative AI introduces significant nuances. Here are the six stages of a project:

1. *Project scoping*: Developers define the project's scope, business objectives, and success criteria. They identify risks and constraints, secure stakeholder alignment, and allocate resources.
2. *Dataset/context engineering*: Developers prepare and process data for machine learning. This includes ingestion, cleaning, and data transformation (which we covered in Chapters 2 and 7). Data annotation can also be considered as part of data engineering.
3. *Model development/adaptation*: Developers build and refine models through feature engineering, algorithm selection, and hyperparameter tuning (which we covered in Chapter 3).
4. *Deployment*: Developers operationalize models so their predictions are accessible to end users (which we covered in Chapters 4, 5, and 9).
5. *Monitoring and continual learning*: Developers track deployed models for performance drift, detect any failures or bias, and adapt models as requirements and environments evolve (which we covered in Chapters 6 and 9).
6. *Business analysis*: Developers evaluate model outcomes against business goals, to ensure that the project generates business value. This stage is closely related to project scoping.

Figure 10-1 depicts these stages.

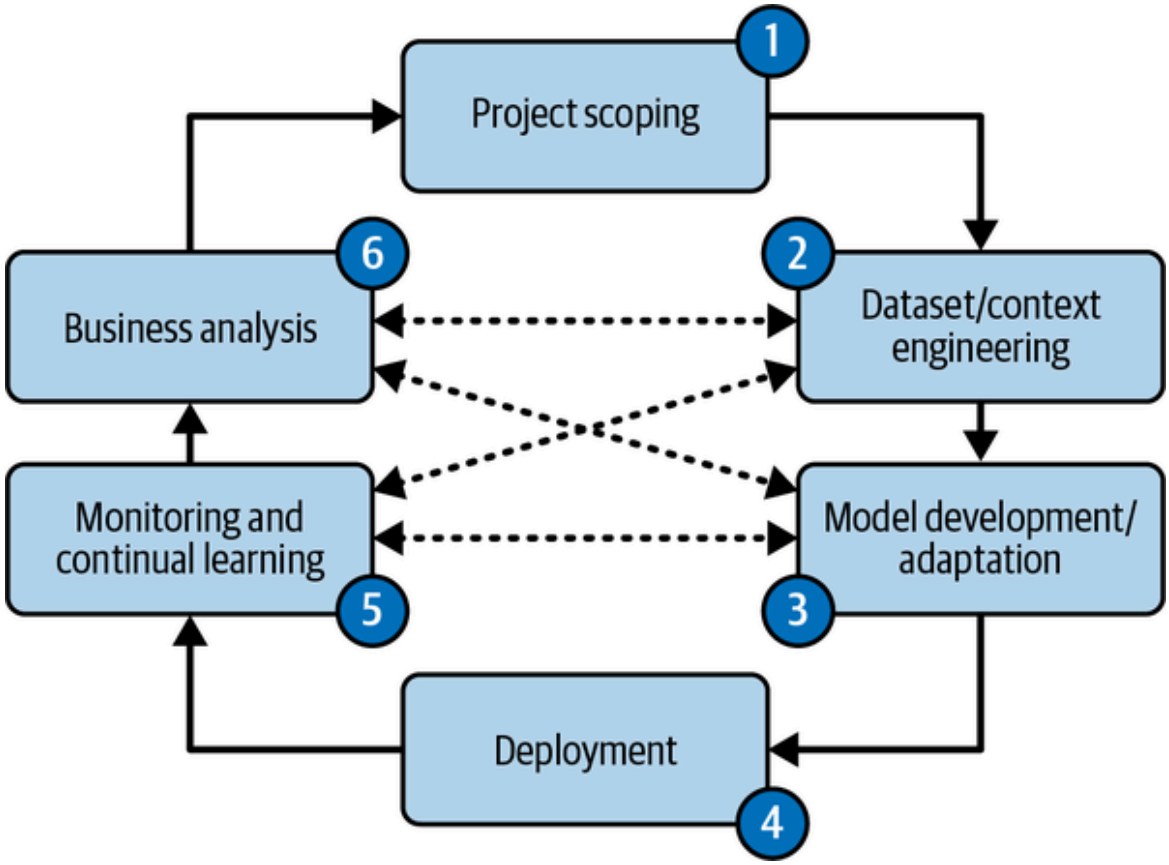


Figure 10-1. The iterative process for developing AI systems, adapted from Designing Machine Learning Systems by Chip Huyen

Model development is at the core of ML engineering. In AI engineering, however, you are working with a model that is already trained and adapting it to your own needs. Prompt-based techniques can adjust model output without updating model weights. Some adaptations aren't possible without changing model weights; for example, you can't adapt a model to a new task it wasn't exposed to during training. In such cases, you should look into model fine-tuning techniques.

The process of *dataset engineering* is relatively straightforward in traditional ML. In the most common scenario, it involves working with tabular data and predefined model outputs. For example, for hotel booking cancellation, the outputs would be *cancelled* or *not cancelled* within a month of the reservation date. In the case of AI agents, outputs are not restricted to a fixed set of labels, and annotating them often requires you to exercise subjective judgment or have domain expertise.

We've noticed that dataset (context) engineering in AI engineering would therefore potentially involve these additional steps, depending on the approach being used:

Prompt engineering

> In an ideal scenario, this requires a fixed benchmark dataset of inputs and expected outputs to measure and optimize prompt performance.

Retrieval-augmented generation (RAG)

> This focuses on preparing, chunking, and embedding domain-specific corpora.

Fine-tuning

> This applies in rare cases, and it emphasizes data collection, cleaning (removing duplicates and toxic or sensitive data), and annotation.

When it comes to *model deployment*, traditional ML deployment typically involves packaging a model into a REST API or embedding it in a business system. AI deployment introduces unique challenges, such as managing context length and retrieval efficiency, integrating guardrails, and orchestrating multiple models for fallback. Inference optimization is a concern in both ML and AI, but it is far more pressing for AI applications, where costs and latency can escalate quickly.

Monitoring in ML typically means tracking relevant model and business metrics. For AI systems, monitoring is harder because outputs are open ended and subjective, which means *human evaluation* often becomes necessary.

Business analysis in AI systems is broader and less clearly defined than in traditional ML. Instead of fixed metrics, teams must align on *what "good"*

looks like, often across subjective dimensions like helpfulness, correctness, and safety. Evaluation typically combines automated metrics, human judgment, and user behavior signals. Because requirements often evolve through real user interactions, business analysis becomes a *continuous, iterative process* that's closely tied to both evaluation and system improvement.

These stages are tightly interconnected and form a continuous, iterative lifecycle. While deployment and monitoring (stages 4 and 5) are the primary focus of MLOps, their success depends heavily on disciplined practices established in the earlier stages.

Next, we will identify which lifecycle stages each of the high-risk requirements (Articles 9–15) impacts and how the Databricks platform can automate compliance.

> **NOTE**
>
> The author is not a lawyer, and this content does not constitute legal advice. It is intended for technical and operational guidance only.

Article 9: Risk Management System

Article 9 of the EU AI Act requires providers of high-risk AI systems to establish, implement, document, and maintain a risk management system throughout the entire lifecycle of the system. The focus is on risks that can be reasonably mitigated or eliminated through the design, development, and operation of the system.

Risk management is not a one-and-done compliance check; it's a continuous and iterative process that mirrors the iterative nature of the AI lifecycle.

Project scoping (risk identification)

During *project scoping*, teams should do the following:

- Identify and analyze known and reasonably foreseeable risks to people’s health, safety, and fundamental rights.
- Consider whether the AI system may adversely impact vulnerable groups (e.g., minors) under the intended use.
- Evaluate risks not only under intended use but also under reasonably foreseeable misuse.

Development and testing (mitigation)

Before a team can put a high-risk system into service, it must perform testing to determine which risk mitigation measures are appropriate and verify that the system performs consistently for its intended purpose. To comply with the regulations, they must do the following:

- Test against the predefined evaluation metrics and probabilistic thresholds to ensure testing is objective and traceable.
- Perform real-world testing where appropriate.

Post-market monitoring (via a feedback loop)

Teams must continuously analyze data collected from real-world usage to detect new or unforeseen risks that were not identified during development. These insights must feed back into the risk management process to update mitigation measures as the system evolves.

Types of risks and mitigation strategies

Now, let’s walk through some examples of risks in AI systems and how to mitigate and eliminate them. They include the following:

- Data-related and model behavior risks
- Operational risks
- Security/cyber risks

Data-related and model behavior risks

Data risks are the most common source of AI failure, and they often lead to biased or unsafe model outcomes. Common risks include the following:

- Biased datasets (for example, medical data that predominantly represents male patients)
- Incorrect or inconsistent labels
- Missing, duplicated, or outdated data
- *Data drift*, in which the data no longer represents the real world

Data-related risks often become *model behavior risks*. Models trained on problematic datasets are likely to produce unsafe or discriminatory outcomes. To mitigate these risks, teams must continuously validate data quality.

Lakeflow SDPs allow teams to define data quality rules ("expectations") that automatically detect and quarantine invalid or unexpected records. In addition, teams can set up Databricks Data Quality Monitoring on input data, to compute profile and drift metrics over time.

Teams can also extend monitoring to inference data (logged predictions and features), as we discussed in Chapter 6, to enable detection of drift between training and serving. They can also configure alerts by defining SQL queries on the metrics tables, to notify teams when data deviates from expected patterns. This enables early detection of issues, before they propagate into model behavior.

Operational risks

These arise not from data or model behavior but from the processes used to move models into production. Examples of situations that create operational risks include the following:

- Broken or untested code being pushed directly to production
- Development and production environments being misaligned

- A deployment failing midway, leaving a system in an inconsistent state
- Teams accidentally deploying the wrong model version
- Teams making manual changes in production that bypass governance and auditability

These situations can lead to outages, incorrect predictions, and loss of traceability, all of which are unacceptable for high-risk AI systems.

To mitigate operational risk, deployment must follow controlled, automated, and auditable workflows. Code and configuration should be promoted to production only through CI/CD pipelines, not via manual changes in the production environment. Developers should not have write access to production, and promotion should require approval gates. We discussed best practices in detail in Chapter 5, and we also demonstrated how DABs help to simplify and streamline the deployment process to different environments.

Security and cybersecurity risks

Security risks arise when an AI system or its infrastructure can be accessed, manipulated, or misused by unauthorized parties. These risks apply not only to the model but also to data, serving endpoints, credentials, CI/CD pipelines, and even third-party dependencies. Examples of security risks include the following:

- Prompt injection or jailbreaking attempts in LLM applications
- Leakage of personal or sensitive data through logs or model outputs
- Exposure or theft of secrets (API keys and OAuth tokens) used within AI systems
- Compromise of a package dependency or CI/CD pipeline, which can lead to malicious code execution

To mitigate these risks, least-privilege access and controlled permission boundaries are essential. In Chapters 4 and 9, when discussing *OAuth authentication*, we highlighted the importance of using the principle of minimum permissions: each service principal should have only the exact access required; for example, they should be able to query a single Model Serving endpoint and nothing else.

In Chapters 4 and 9, we talked about the Unity AI Gateway, which helps reduce security and misuse risks by enabling guardrails (input/output filtering), rate limits, usage tracking, and inference logging. Usage is written into system tables, so teams can detect unusual traffic patterns or potential abuse. *Inference tables* provide detailed logs of every request, and *MLflow Tracing* captures all intermediate steps for LLM applications, enabling teams to audit and investigate suspicious activity.

The recommended best practice is to store credentials in *Databricks secret scopes* and reference them at runtime, to ensure that no secrets are embedded in code or configuration files. We used Databricks secret scopes in Chapter 9.

Security risks are not limited to application behavior. They can originate from the software supply chain itself. Attackers increasingly use or exploit the following:

Typosquatting

> This involves creating malicious packages with names that are very similar to popular libraries (e.g., installing "reqeusts" instead of "requests").

Malicious or compromised packages on PyPI

> A real-world example of such a package is the fastapi-toolkit package on PyPI. It was compromised after attackers gained access to the maintainer's account and published a new version containing malware that was designed to exfiltrate environment variables and credentials from build pipelines.

Malicious transitive dependencies

> For example, the LiteLLM library (which has about 100 million downloads per month)—a dependency for CrewAI, databricks-agents, and other Python packages—recently came under attack.

Repojacking/dependency hijacking

> In GitHub Actions, if an Action is referenced as `{owner}/{repo}@{tag}` or `{branch}` and attackers take control of the repository, they can inject malicious code. Then, when the workflow runs, the runner loads secrets into memory that malware can exfiltrate.

Users accidentally committing API keys or OAuth tokens to Git

> This is one of the most common and avoidable security incidents.

These are only a small number of examples. As AI adoption accelerates and introduces new attack surfaces in the coming years, we will inevitably see more frequent and sophisticated attacks. To mitigate these risks, companies can consider using *private package repositories* and allowing installation only from approved sources instead of the public PyPI. They can also avoid referencing GitHub Actions by branch or tag and instead pin them to a *commit SHA* to prevent repojacking.

These examples demonstrate how a small misconfiguration or dependency oversight (core or transitive) can lead to severe consequences. Code scanning across organizations can help to find and address vulnerabilities, to ensure best practices are followed.

Article 10: Data and Data Governance

While Article 9 of the EU AI Act requires a continuous risk management process, Article 10 focuses on one of the most critical risk areas: the *quality*

of data used to develop high-risk AI systems. If an AI system involves training a model, the provider must ensure that training, validation, and testing datasets are governed, managed, and controlled throughout the lifecycle.

Let's walk through the AI project lifecycle and requirements that teams should consider at each stage.

Project scoping

Compliance starts before any data is ingested. During project scoping, teams must understand the intended purpose of the AI system and establish a *data governance strategy* aligned with Article 10. This includes defining how data will be collected, what sources will be used, and whether the data is suitable for the model.

Article 10 requires that organizations do all of the following:

- Document the design choices that led to selecting specific datasets
- Record assumptions about what data represents
- Assess whether data is sufficiently available, representative, and suitable
- Consider the potential impact of data biases on individuals or groups

At this stage, governance is primarily about *planning and accountability*: identifying data sources, evaluating licensing and usage rights, and assessing whether the system may affect vulnerable populations.

Dataset/context engineering

In this phase, raw data is processed into structured and governed datasets. Article 10 mandates data governance practices and explicitly requires documentation of the following:

- Data collection processes and origin

- Data preparation operations, such as labeling, cleaning, and aggregation
- Analysis of potential biases and measures to mitigate them
- Identification of gaps in representativeness or quality and how they will be addressed

Datasets must be all of the following:

- Relevant
- Sufficiently representative (including geographic or contextual characteristics of the real-world deployment setting)
- To the greatest extent possible, free of errors
- Complete for the intended purpose

Databricks capabilities help streamline data quality enforcement. Built-in expectations allow you to automatically detect and isolate invalid records.

However, this functionality is limited to declarative pipelines. For standard PySpark-based workflows, an alternative is DQX by Databricks Labs. It is a data quality framework for Apache Spark that enables teams to define, monitor, and handle data quality issues directly in Python-based pipelines. While we do not explore DQX in detail in this book, it is a valuable project that teams working with PySpark-based data pipelines will find worth exploring.

Delta Lake provides data version control and lineage tracking: every transformation step becomes traceable and auditable. *Unity Catalog* supports best governance practices by providing the least-privilege access that's required for the purpose. We discussed best practices in Chapter 5.

Compliance is demonstrated by traceability (how the dataset was constructed) and justification (why it is suitable).

Model development/adaptation

Article 10 requires datasets that are used for model training, validating, and testing to meet the established quality criteria: governance, representativeness, and bias management.

During this phase, teams must do the following:

- Track the dataset versions they use to train each model iteration
- Validate that training, validation, and test sets represent the real-world population
- Test the model on relevant subgroups to verify fairness and prevent discriminatory outcomes

In Chapter 3, we demonstrated how to log evaluation results and datasets using MLflow and link it to the models registered in Unity Catalog.

Deployment

The EU AI Act requires that data within the deployment environment must remain consistent with the quality and representativeness standards we established earlier: datasets must remain relevant and appropriate to the intended purpose.

Practical safeguards include the following:

- Quality gates in ML pipelines that prevent model promotion if dataset or fairness metrics regress
- Controlled release processes that prevent the accidental use of outdated or unapproved datasets

Lakeflow Jobs allows for if-else statements, as shown earlier in Chapter 5, to enable model promotion only if certain quality thresholds are met.

Monitoring and continual learning

Article 10 does not end at deployment. It requires organizations to continuously evaluate whether the data flowing into their systems remains

representative over time. Real-world datasets change due to market shifts, user behavior changes, and seasonal effects. A model trained on compliant data may become noncompliant if the data context shifts.

Databricks supports this phase through the following:

- *Data Quality Monitoring*, which detects drift between inference data and a baseline, and between different windows of inference data
- Automated *alerting* when statistical properties deviate from expectations
- *Inference tables* and *MLflow Tracing*, which capture model inputs and outputs for audit and investigation
- Bias monitoring in Data Quality Monitoring (classification)

This fulfills the implicit requirement of Article 10: compliance is *ongoing*, not a one-time validation exercise.

Business analysis

There are rare circumstances in which processing of special-category personal data (e.g., race, ethnicity, health data) is required to detect or correct bias. Article 10.5 permits this only when all of the following are true:

- No alternative data can achieve the same objective.
- The data is strictly access-controlled and documented.
- The data is deleted once bias mitigation is completed.

Business analysis is responsible for justifying this exceptional processing and documenting why it is necessary. *Unity Catalog* permission controls, data retention policies, audit logging, and column masking support the strict safeguards mandated by Article 10.5.

Article 11: Technical Documentation

Article 11 requires providers of high-risk AI systems to prepare and maintain *technical documentation* that demonstrates compliance with the law. This documentation must be complete before a provider can place a system on the market or put it into service, and it must be kept continuously up to date as the system evolves.

The purposes of technical documentation are to prove that an AI system meets all applicable regulatory requirements (such as risk management, data governance, and transparency) and to enable authorities to perform independent assessments of the AI system.

Article 11 specifies that technical documentation must include all elements defined in the Annex IV reporting framework, at minimum. Annex IV requires the documentation of information such as the following:

- Intended purpose and system architecture
- Dataset characteristics and data governance measures
- Risk management strategy and mitigation actions
- Model performance evaluation, limitations, and monitoring processes
- System logs, versioning, and auditability records

Importantly, this is not technical documentation in the traditional engineering sense. It is a *compliance artifact* that describes how the system meets the Act's requirements and provides evidence to support that claim.

Technical documentation becomes the compliance backbone—the place where all actions taken under Article 9 (Risk Management) and Article 10 (Data Governance and Data Quality) are captured, evidenced, and made auditable.

With the growing adoption of AI-assisted software development, documentation is no longer just a compliance requirement; it is becoming a *technical necessity.* Tools that generate or modify code based on prompts,

architecture descriptions, or past decisions require context to operate correctly. Without documentation, there is no shared understanding of why decisions were made, what constraints exist, or what assumptions must be preserved.

Documentation should *live where the system lives: in the repository, under version control*, not in scattered Confluence pages or slide decks. One effective approach could be to store documentation as Markdown files in the same repository as the code. It could be updated iteratively, ideally in the same pull requests as code changes, and AI-assisted tools like Claude Code could help generate or refine these updates.

Keeping documentation in the repo provides several benefits like *traceability* (every change to code and documentation is tracked together), *discoverability* (other developers can easily find relevant documentation), and *governance* (documentation becomes part of code review, not an afterthought).

Article 12: Record-Keeping

Article 12 requires high-risk AI systems to automatically record events in logs for their entire operational lifetime. Logging is not an optional enhancement; it is a mandatory capability that must be built into the system from the beginning. The system must provide an audit trail that makes it possible to reconstruct what happened, why it happened, and who interacted with it.

Logs must capture the events that are necessary to the following:

Detect situations that may present risks

> Logs must identify events that could endanger health, safety, or fundamental rights. For example, they must identify anomalous predictions, data drift, and system malfunctions.

Detect substantial modifications

> Logs must record evidence of changes that alter the system's behavior or compliance status, such as retraining on new data, modifying features, and deploying a new model version without approved validation.

Support post-market monitoring (providers only)

> Providers must continuously monitor deployed systems to confirm ongoing compliance, detect emerging risks, and demonstrate in their logs that safeguards remain effective.

Monitor operational behavior and oversight (deployers only)

> Deployers must ensure and document in their logs that their systems operate safely day-to-day, that human-in-the-loop processes are working, and that every decision can be traced to its origin.

Rather than traditional application logging, this regulation focuses on logs that help explain *decision-making behavior*, not just technical errors. The logging must be "appropriate to the intended purpose," meaning logs should make it possible to trace a specific output back to its inputs, the model version used, and any relevant human intervention.

For high-risk systems used in *remote biometric identification or categorization*, where permitted by law, additional requirements apply. These systems must record the following:

- When the system was used (the start and end time of each session)
- Which reference database was used for comparison
- Which inputs resulted in a match
- The identity of the human reviewers who verified or approved results

To meet Article 12 obligations, AI systems should automatically log the categories of events listed in Table 10-2).

Table 10-2. Required logs and traceability per event category

Event categories	Examples of captured data
Model lifecycle	Model name, registered model version, training run ID, and when the model was retrained
Inference activity	Timestamp, request ID, input features, system output, (when applicable) confidence levels, latency, and error codes
Data sources	The exact version of the dataset used for training and validation, schema, and data drift metrics
Code/deployment source	Git commit hash and deployment pipeline run ID
System and infrastructure	Cluster/job ID, serving endpoint, environment variables, and package versions
Human oversight	Reviewer ID, review timestamp, action (take decision/override), and reason codes
Security and access	User ID, workspace access logs, API calls, and configuration changes

The following Databricks features provide a solid foundation for implementing Article 12 requirements:

Inference tables

> These automatically record every model invocation (inputs, outputs, model version, latency, and status) to create a structured audit trail for each prediction.

System tables

: These capture activities such as job runs, endpoint invocations, and data access for operational oversight.

Data Quality Monitoring

: This continuously computes data drift, bias, and performance metrics to detect new or evolving risks, and it supports alerts for post-market monitoring.

MLflow Tracking and Model Registry

: These store training metadata, datasets, parameters, metrics, and model versions to detect substantial modifications and maintain model lineage.

MLflow Tracing

: This records intermediate steps and reasoning chains in complex or multicomponent AI systems.

In Chapters 3 through 5, we established the foundations of traceability and reproducibility in ML projects. We demonstrated how to connect every component—code, data, infrastructure configurations, and model artifacts—to ensure that every deployment is fully auditable. Then, in Chapter 6, we focused on the monitoring of ML systems. Finally, in Chapters 7 through 9, we expanded these principles to the unique challenge of AI systems, focusing on how to maintain traceability, reproducibility, and observability within GenAI workflows.

Article 13: Transparency and Provision of Information to Deployers

Article 13 extends the principles of Article 11. While Article 11 focuses on *technical documentation* for regulators, Article 13 ensures *transparency for deployers*—the organizations that actually operate high-risk AI systems.

The goal is simple: deployers must be able to *understand, interpret, and appropriately use* the system's outputs.

Transparency must be built into a system from the start. This means more than exposing algorithms; it ensures that users can recognize the system's capabilities and limitations and have confidence in its results. Without this, human oversight (Article 14) and risk management (Article 9) cannot function effectively.

Every high-risk AI system must be accompanied by *clear, accessible instructions for use* in digital form. These must describe the following:

- The provider's identity and contact details
- The system's intended purpose, accuracy, robustness, and cybersecurity characteristics
- Known limitations and foreseeable risks
- How outputs should be interpreted and under what conditions they may be unreliable
- Data or input requirements
- Human oversight expectations
- Maintenance needs and software update procedures
- How to collect and interpret logs

The *provider* is responsible for supplying this information. The *deployer* must follow it, to ensure that the system is used as intended. If a *deployer* alters the system's design, retrains the model, or changes its purpose, it effectively becomes a *provider* and assumes full compliance obligations, including risk assessment and conformity evaluation.

Transparency thus links design and operation: providers make systems explainable, and deployers keep them understandable in practice. It ensures that even complex AI remains interpretable, accountable, and safely governed throughout its lifecycle.

Article 14: Human Oversight

Article 14 introduces one of the most practical and human-centered requirements of the EU AI Act: every high-risk AI system must be designed so that it can be effectively overseen by natural persons during its operation.

Human oversight is not an optional safeguard. It is a regulatory requirement that ensures AI systems remain controllable, accountable, and aligned with human judgment. The goal is to prevent or minimize risks to health, safety, and fundamental rights. It serves as the final line of defense when automated systems make decisions that could impact individuals or society.

Human oversight under the AI Act requires systems to be designed with *appropriate human-machine interfaces* that allow trained personnel to do the following:

- Understand the system's capabilities and limitations
- Detect anomalies or unexpected behavior
- Remain aware of *automation bias*
- Correctly interpret outputs in context
- Override or disregard AI-generated decisions when necessary
- Interrupt or safely stop a system's operation if it behaves unexpectedly

These oversight measures must be *proportionate* to the system's risk level, autonomy, and intended use. For example, in low-frequency, high-impact decisions (such as medical diagnoses or credit approvals), human review may be mandatory before each decision is finalized. In higher-volume operational systems (such as predictive maintenance), periodic audits or threshold-based alerts may be sufficient.

For biometric identification systems covered under Annex III(1)(a), Article 14 adds a further safeguard: no identification result may be acted upon unless verified by at least two qualified human reviewers (except where law enforcement exemptions apply).

Human oversight can be implemented by both *providers* and *deployers*, depending on the system design. Providers should enable technical controls and interfaces embedded into the system (for example, dashboards, alerts, or review workflows) to make human monitoring and intervention possible. Deployers should enable organizational or procedural controls (such as human review steps, approval protocols, or secondary validation processes) before executing actions.

When deploying AI applications on Databricks, organizations can embed several human-in-the-loop safeguards into the operational workflow. We discussed one of them in Chapter 5. At the code level, organizations can enforce branch protection rules in GitHub so that all changes to the main branch require approval by at least two reviewers before merging. Organizations can also enforce deployment protection rules so that deployment to production can only proceed after human approval.

MLflow provides an interface for manual review of model outputs. It enables reviewers to inspect predictions, approve or override them, and log decisions for audit and traceability.

Article 15: Accuracy, Robustness, and Cybersecurity

Article 15 brings together three essential pillars of trustworthy AI: *accuracy, robustness, and cybersecurity*. It also requires high-risk AI systems to maintain these qualities *consistently throughout their lifecycle*. It defines what the system itself must *technically achieve* to be considered safe and compliant.

A high-risk AI system must not only perform well under test conditions but also remain reliable, stable, and secure once deployed, even as its data, environment, and users evolve over time.

Accuracy represents the degree to which the system's predictions or classifications correspond to reality or the intended output. Under Article 15, providers must do the following:

- Define and document the accuracy metrics that are appropriate to the system's purpose
- Validate performance against those metrics during development and testing
- Disclose those metrics in the *instructions for use*

The "appropriate" level of accuracy depends on the system's context and risk. For instance, an AI-based medical triage tool or credit-scoring model must achieve far higher validated accuracy and lower false negative rates than a document classifier or maintenance predictor.

Robustness is a system's ability to *operate reliably under varying conditions*, including imperfect data, software faults, environmental noise, and human interaction. A robust AI system must continue to function safely even in the presence of errors or inconsistencies, and it must recover gracefully if something goes wrong.

Key robustness measures include the following:

Error and anomaly handling

> This includes detecting and managing unexpected inputs or missing values without failure.

Fail-safe mechanisms

> These provide fallback modes or manual override when conditions deviate from expected norms.

Technical redundancy

> The system must include backup models, alternate logic paths, or rule-based guards that maintain safe operation.

Continual learning safeguards

> For high-risk AI systems that continue to learn after deployment, mechanisms must be in place to prevent

> situations where biased or incorrect outputs are reintroduced as new inputs and gradually distort the system's behavior over time.

Cybersecurity in high-risk AI systems means that such systems must be *resilient to manipulation and malicious interference*, including attacks targeting data, models, or infrastructure. This includes AI-specific vulnerabilities such as the following:

Data poisoning

> This involves injecting corrupt or biased samples into training datasets.

Model poisoning

> This involves compromising pre-trained components or shared embeddings.

Adversarial examples

> This involves intentionally crafting inputs that are designed to make the model misclassify data.

Model evasion

> This involves probing decision boundaries to reverse-engineer sensitive logic.

Confidentiality or extraction attacks

> These attempt to gain access to training data or proprietary parameters.

As we noted in the discussion of Article 9, providers must implement technical and organizational measures to prevent, detect, and mitigate these threats, and to ensure that any compromise does not lead to unsafe outputs or privacy violations.

MLOps Checklist

Before teams deploy an AI system to production, especially in regulated environments, they must assess whether both the system and the surrounding processes meet required standards for safety, governance, and compliance. However, they should not treat this type of checklist as a one-time, final gate.

Instead, an MLOps checklist is most effective when applied at *key lifecycle milestones*, such as before production deployment, during major releases, or as part of periodic compliance reviews. Running it earlier helps identify gaps proactively, while revisiting it over time ensures that the system remains compliant as it evolves.

Importantly, not all criteria need to be fully satisfied from day one. Teams should treat this checklist as a *maturity framework*, prioritizing critical risks first and progressively strengthening controls as the system scales and its impact increases.

With that in mind, we can synthesize these requirements into a single actionable view. The checklist in Table 10-3 illustrates how modern MLOps practices directly support compliance with the EU AI Act.

Table 10-3. MLOps checklist

Item #	Assessment criterium	Linked article(s)
	Documentation	
1	The business goals, intended purpose, and KPIs of the AI system are documented and kept up to date.	9, 11, and 13
2	The roles, responsibilities, and accountability of all team members are documented and regularly reviewed.	9 and 11
3	AI system risk assessment and mitigation plans are created, maintained, and updated throughout the lifecycle.	9
4	Documentation includes assumptions, design choices, and justification for dataset selection and model type.	10 and 11
5	Technical documentation demonstrates compliance (in architecture, datasets, risk management, and performance).	11
6	Documentation resides in version control and is updated with code changes for full traceability.	11 and 12
7	Instructions for deployers describe intended use, limitations, and	13

Item #	Assessment criterium	Linked article(s)
	maintenance procedures.	
8	Human oversight procedures and escalation paths are described in documentation.	14
9	Every release includes a versioned compliance summary or “model card.”	11 and 13
	Infrastructure	
10	Infrastructure is defined as code (IaC) to ensure traceability and reproducibility.	9 and 15
11	IaC is stored in version control and updated through pull requests.	9 and 12
12	Changes to infrastructure are automatically deployed via controlled CD pipelines.	9 and 15
13	Only automated deployment pipelines can apply changes; manual deployment is restricted.	9 and 15
14	At least two environments (pre-production and production) exist for controlled deployment and are synchronized in configuration.	9 and 15
15	Infrastructure health and cost are continuously monitored; alerts are	9 and 15

Item #	Assessment criterium	Linked article(s)
	configured for anomalies.	
	Data and models	
16	Code for data gathering, cleaning, and feature preparation is version-controlled.	10 and 12
17	Datasets used for training, validation, and testing are versioned and auditable.	10 and 12
18	Dataset lineage and transformations are traceable.	10 and 12
19	Data quality rules and expectations are defined and enforced automatically.	10 and 15
20	Dataset representativeness, completeness, and bias checks are documented and validated.	10
21	Processing of sensitive or special-category data is justified, access-controlled, and deleted after use.	10 and 15
22	Drift monitoring is implemented for both training and inference data.	10 and 12
23	Offline evaluation metrics (e.g., accuracy, F1) are logged and monitored over time.	9 and 15

Item #	Assessment criterium	Linked article(s)
24	Feedback loops capture real-world performance and update the risk management process.	9 and 10
	Code and deployment governance	
25	All code related to the AI system (data, model, and API) is version controlled (with a preference for a single version control repository).	9 and 12
26	Pull requests with peer review are required for all code and configuration changes.	9 and 14
27	Continuous integration (CI) runs automated tests and validation at PR creation.	9 and 15
28	Continuous delivery (CD) promotes artifacts only after passing quality and compliance gates.	9 and 15
29	Deployment pipelines facilitate recording run IDs, approvers, and linking deployed model versions to code.	12 and 14
30	Rollback procedures allow reversion to previous safe model and infrastructure states.	9 and 15

Item #	Assessment criterium	Linked article(s)
31	Developers do not have direct write access to production environments. Such access can be provided temporarily when production failures occur.	9 and 15
	Logging and auditability	
32	The system automatically logs all model lifecycle events (training, retraining, and deployment).	12
33	Each inference request logs the timestamp, model version, input reference, output, and latency.	12
34	Logs capture data versions, schema, drift metrics, and alert history.	12
35	All logs are retained securely, tamper resistant, and available for audit.	12
36	Human interventions (reviews, overrides, and approvals) are logged with user IDs and reasons.	12 and 14
37	Logging covers cybersecurity events (unauthorized access, failed authentications, and data exfiltration attempts).	12 and 15
	Transparency and explainability	

Item #	Assessment criterium	Linked article(s)
38	The system provides meaningful explanations for outputs (e.g., feature importance, and SHapley Additive exPlanations [SHAP]).	13 and 14
39	Model behavior and limitations are documented in language that’s understandable by deployers and users.	13
40	Uncertainty estimates or confidence levels are logged and exposed when relevant.	13 and 15
41	Providers supply digital instructions describing interpretation, accuracy, and robustness.	13
42	Explanations are validated with intended users to ensure understandability.	13 and 14
43	Human approval gates are embedded in model deployment workflows.	14
44	Dashboards or review interfaces enable oversight of predictions and system health.	14
45	Oversight personnel are trained to detect anomalies and automation bias.	14

Item #	Assessment criterium	Linked article(s)
46	Humans can override or stop the system safely if behavior is unexpected.	14 and 15
47	Records of oversight actions are stored for audit and continuous improvement.	12 and 14
	Accuracy, robustness, and cybersecurity	
48	Accuracy thresholds are defined and validated under representative conditions.	15
49	System performance is periodically re-evaluated with real-world data.	9 and 15
50	Fail-safe mechanisms and fallback modes are implemented for degraded states.	15
51	Regular stress, load, and penetration tests verify robustness and latency targets.	15
52	Least-privilege access and network isolation are enforced for models and data.	9 and 15
53	Secrets (API keys and tokens) are stored securely and never embedded in code.	15

Item #	Assessment criterium	Linked article(s)
54	Dependencies and supply chain components are verified and scanned for vulnerabilities.	15
55	Controls are in place to prevent adversarial, poisoning, or model-evasion attacks.	15
56	Cybersecurity controls are periodically reviewed and updated.	15
	Monitoring and continual compliance	
57	Post-market monitoring collects data on system behavior and emerging risks.	9 and 12
58	Alerts are generated when performance, drift, or bias metrics exceed thresholds.	9, 10, and 15
59	Results from monitoring feed back into risk management and retraining processes.	9 and 10
60	Compliance status is periodically reviewed and reported to governance stakeholders.	9, 11, and 12

Conclusion

One of the biggest challenges organizations face with AI governance is not a lack of frameworks, policies, or principles. It's making them *practical and actionable*. Governance often lives in slide decks and policy documents, while real decisions are made in code, pipelines, and product iterations.

This is exactly where the practices covered throughout this book come together.

Every chapter has not been just about building better AI systems. It has been about building *governable systems by design*:

- When you version code, data, and models, you enable *traceability.*
- When you evaluate outputs systematically, you enable *accountability.*
- When you monitor systems in production, you enable *risk detection.*
- When you design feedback loops, you enable *continuous oversight.*
- When you standardize pipelines, you enable *repeatability and auditability.*

As you move forward, consider governance as part of your system design—not an afterthought. Build processes that scale, document decisions that matter, and foster a culture where ethical considerations are treated as first-class engineering requirements.

In the end, AI governance is about trust. And trust, once earned through consistent and responsible practice, becomes a lasting competitive advantage.

---

[1] European Law Institute, "The Concept of 'AI System' under the New AI Act: Arguing for a Three-Factor Approach," Commission Guidelines on the Application of the Definition of an AI System and the Prohibited AI Practices Established in the AI Act, 2024, *https://oreil.ly/BGtNv*.

[2] Future of Life Institute, *High-Level Summary of the AI Act*, February 27, 2024, *https://oreil.ly/uTGlH*.

Epilogue

Congratulations on finishing this book.

Writing it was far more challenging than I ever anticipated, especially because it covers a space that evolves at an extraordinary pace. Throughout the process, I had to significantly update the code multiple times (particularly the LLMOps chapters), while Databricks continuously introduced new features, renamed components, and improved abstractions. At times, it felt like trying to document a moving target.

And yet, the core principles remain stable.

Versioning data and models, ensuring reproducibility, applying guardrails, monitoring systems, and maintaining governance are not tied to specific tools. They reflect the fundamentals of building reliable and scalable AI systems, and they are not going away.

What is changing is how we build. We no longer build solutions to last; we build solutions to adapt.

AI is now used to build AI. Engineers rely on agents to generate code, while the platforms they use evolve just as quickly. This creates a new reality: features change, better abstractions emerge, and what was considered best practice just months ago may already be outdated.

Rather than chasing every new trend, focus on strong engineering practices. They provide the stability needed to build robust, compliant, and sustainable systems.

Ultimately, success starts with understanding what “good” looks like. I hope this book has given you that foundation.

Feel free to reach out:

LinkedIn: *linkedin.com/in/maria-vechtomova*

Email: *vechtomova.maria@gmail.com*

For more resources about MLOps and LLMOps, check out the book's GitHub repository: *https://github.com/mvechtomova/oreilly-mlops-with-databricks*.

Index

A

A/B testing, for models, A/B Testing-Moving assignment into the model logic

– moving assignment into the model logic, Moving assignment into the model logic-Moving assignment into the model logic

– routing strategies, Choosing a routing strategy-Request-level assignment

access control, platform design with, Access Control Strategy-Access Control Strategy

access modes, Policies and access modes

accuracy, EU AI Act provisions for, Article 15: Accuracy, Robustness, and Cybersecurity

Agent-as-a-Judge, MLflow GenAI Evaluation Framework

agents

– core logic, Core Agent Logic-Core Agent Logic

– custom MLflow, Defining a Custom Agent-Testing the Agent

– initialization/async handling, Initialization and Async Handling

– logging, Agent Logging

– memory with Lakebase, Agent Memory with Lakebase-Utilizing a connection pool

– prediction/streaming, Prediction and Streaming

– registering/preparing for serving, Registering the Agent and Preparing for Serving

– testing, Testing the Agent

AI gateways, AI gateways, Unity AI Gateway

AI governance, AI Governance-Conclusion

– AI project lifecycle, AI Project Lifecycle

– EU AI Act

– Article 9: risk management system, Article 9: Risk Management System-Security and cybersecurity risks

– Article 10: data and data governance, Article 10: Data and Data Governance-Business analysis

– Article 11: technical documentation, Article 11: Technical Documentation

– Article 12: record-keeping, Article 12: Record-Keeping

– Article 13: transparency/provision of information to deployers, Article 13: Transparency and Provision of Information to Deployers

– Article 14: human oversight, Article 14: Human Oversight

– Article 15: accuracy/robustness/cybersecurity, Article 15: Accuracy, Robustness, and Cybersecurity

– high-risk AI requirements, Requirements for High-Risk AI-Article 15: Accuracy, Robustness, and Cybersecurity

– risk categories, Risk Categories and Whom They Apply To-Risk Categories and Whom They Apply To

– MLOps checklist, MLOps Checklist

– scope, The Scope of AI Governance-Risk Categories and Whom They Apply To

alerting, Alerting

all-purpose compute, All-purpose versus job compute, Using all-purpose compute

API limits, Resource and API Limits

APIs, MLflow tracing, MLflow Tracing APIs

approximate nearest neighbor (ANN) search, Databricks AI Search, Querying and retrieval quality

augmented LLM, What Is an Agent?

AWS (Amazon Web Services), Continuous Deployment Pipeline

B

batch inference, Workflow orchestration

bias monitoring, Fairness, Bias, and Regulatory Compliance

branch protection rules, Branch and deployment protection rules

business analysis

– EU AI Act provisions, Business analysis

– ML versus AI, AI Project Lifecycle

C

categorical feature drift, Using the Dashboard

CD (continuous deployment) pipeline, Continuous integration and continuous deployment (CI/CD), Continuous Deployment Pipeline-Branch and deployment protection rules

chi-squared test, Chi-squared Test

child model, Regression and Demand Forecasting

chunking, Chunking-Processing chunks

- embedding window considerations, Embedding window considerations
- processing chunks, Processing chunks
- strategies, Chunking strategies

CI (continuous integration) pipeline, Continuous integration and continuous deployment (CI/CD), Continuous Integration Pipeline, Practical considerations: Integration tests in CI

classic compute, Classic compute versus serverless compute, Serverless versus Classic Compute for Lakeflow Jobs

cluster configurations, Cluster Configurations on Databricks-Cost tracking

- all-purpose cluster/jobs compute cluster configuration, All-Purpose and Jobs Compute Configuration-Cost tracking
  - advanced settings, Advanced settings
  - cost tracking, Cost tracking
  - Photon acceleration, Photon acceleration
  - policies/access modes, Policies and access modes
  - runtime versions, Runtime versions
  - worker/driver nodes, Worker and driver nodes
- serverless compute, Serverless Compute
  - cost tracking, Cost tracking
  - serverless environments, Serverless environments

code development on Databricks, Developing on Databricks-Using all-purpose compute

- cluster configurations, Cluster Configurations on Databricks-Cost tracking
  - all-purpose/jobs compute configuration, All-Purpose and Jobs Compute Configuration-Cost tracking
  - serverless compute, Serverless Compute
- developer tools, Databricks Developer Tools-Using DABs
  - DABs, Declarative Automation Bundles-Using DABs
  - Databricks CLI, Databricks CLI
  - Databricks Connect, Databricks Connect-Databricks Connect
  - Databricks extension for Visual Studio Code, Databricks Extension for Visual Studio Code-Databricks Extension for Visual Studio Code
  - Databricks notebook, Running Code in a Databricks Notebook-Using all-purpose compute
- project setup, Project Setup-Project Code: Data Preprocessing
  - dataset, The Dataset
  - project code/data preprocessing, Project Code: Data Preprocessing
  - pyproject.toml file, The pyproject.toml file-Pytest configurations
  - Python tooling, Python Tooling

code hygiene, Code Hygiene

code reviews, Code Hygiene

code version control, Code version control

code-based graders, How to Evaluate LLM Systems

compute

– in Databricks, Databricks Compute-All-purpose versus job compute

– serverless versus classic, Serverless versus Classic Compute for Lakeflow Jobs

concept drift, Large Language Models

conda, Python Tooling

connection pools, Utilizing a connection pool-Utilizing a connection pool

consecutive drift, Introduction to Databricks Data Quality Monitoring

context engineering, Context Engineering-Utilizing a connection pool

– agent memory with Lakebase, Agent Memory with Lakebase-Utilizing a connection pool

– conversion of raw data to chunks, Getting from Raw Data to Chunks-Processing chunks

– chunking, Chunking-Processing chunks

– data ingestion, Data ingestion-Data ingestion

– PDF processing, PDF processing-PDF processing

– project configuration, Project configuration-Project configuration

– Databricks AI Search index, Databricks AI Search Index-Optimizing search quality

– Genie agent, Genie Agent (Formerly, Genie Space)-Interacting with Genie

– MCP servers/tool calling, MCP Servers on Databricks and Tool Calling-Implementing tool calling

continuous deployment (CD) pipeline, Continuous integration and continuous deployment (CI/CD), Continuous Deployment Pipeline-Branch and deployment protection rules

continuous integration (CI) pipeline, Continuous integration and continuous deployment (CI/CD), Continuous Integration Pipeline, Practical considerations: Integration tests in CI

cost monitoring, Cost Monitoring-Cost-monitoring dashboard

– cost-monitoring dashboard, Cost-monitoring dashboard-Cost-monitoring dashboard

– Databricks billing, How Databricks billing works

– Databricks services/associated costs, Databricks services and associated costs

– tag-based cost attribution, Tag-based cost attribution

cost tracking, Cost tracking, Cost tracking

custom agents, MLflow, Defining a Custom Agent-Testing the Agent

– core logic, Core Agent Logic-Core Agent Logic

– initialization/async handling, Initialization and Async Handling

– prediction/streaming, Prediction and Streaming

– testing, Testing the Agent

custom scorers, in MLflow GenAI evaluation framework, MLflow GenAI Evaluation Framework

cybersecurity, Article 15: Accuracy, Robustness, and Cybersecurity

– EU AI Act provisions, Article 15: Accuracy, Robustness, and Cybersecurity

– risks, Security and cybersecurity risks

D

DABs (see Declarative Automation Bundles)

data drift

– concept drift versus, Concept Drift versus Data Drift

– metrics for monitoring, Metrics for Monitoring Data Drift-Population Stability Index

data governance, Article 10: Data and Data Governance-Business analysis

data leakage, Training Data

Data Quality Monitoring, Introduction to Databricks Data Quality Monitoring-Alerting

– alerting, Alerting

– inference tables, From Inference Tables to Data Quality Monitoring-Refreshing the Monitor

– creating a data quality monitor, Creating a Data Quality Monitor-Creating a Data Quality Monitor

– inference table structure, Inference Table Structure

– processing an inference table, Processing an Inference Table

– refreshing the monitor, Refreshing the Monitor

– using the dashboard, Using the Dashboard

– metrics for monitoring data drift, Metrics for Monitoring Data Drift-Population Stability Index

– chi-squared test, Chi-squared Test
– Jensen-Shannon distance, Jensen-Shannon Distance
– Kolmogorov-Smirnov test, Kolmogorov-Smirnov Test
– L-infinity distance, L-Infinity Distance
– Population Stability Index, Population Stability Index
– total variation distance, Total Variation Distance
– Wasserstein distance, Wasserstein Distance
– ML failures in production, Observability
data version control, Data version control
data-related risks, Data-related and model behavior risks
Databricks (generally), Databricks MLOps Components-Databricks Apps
– code development on (see code development on Databricks)
– compute, Databricks Compute
– cross-cutting components, Cross-Cutting Components-Databricks Apps
– DABs, Declarative Automation Bundles
– Databricks Apps, Databricks Apps
– Lakebase, Lakebase
– Unity Catalog, Unity Catalog
– Lakeflow Jobs, Lakeflow Jobs
– MLOps components, Databricks MLOps Components-Databricks Apps

– cross-cutting components, Cross-Cutting Components-Databricks Apps
– Data Quality Monitoring, Data Quality Monitoring
– Databricks AI Guardrails, Databricks AI Guardrails
– Databricks AI Search, Databricks AI Search
– Databricks compute, Databricks Compute
– Databricks Model Serving, Databricks Model Serving
– feature engineering, Feature Engineering
– Lakeflow Jobs, Lakeflow Jobs
– MLflow, MLflow
– Unity AI Gateway, Unity AI Gateway

Databricks AI Guardrails, Databricks AI Guardrails

Databricks AI Search, Databricks AI Search
– creating the endpoint and index, Creating the endpoint and index-Creating the endpoint and index
– Genie space versus, Databricks AI Search versus Genie
– index creation, Databricks AI Search Index-Optimizing search quality
– optimizing search quality, Optimizing search quality-Optimizing search quality
– querying/retrieval quality, Querying and retrieval quality

Databricks Apps, Databricks Apps

Databricks CLI, Databricks CLI

Databricks Connect, Databricks Connect-Databricks Connect

Databricks dashboard, Monitoring dashboard

Databricks Data Quality Monitoring, Introduction to Databricks Data Quality Monitoring-Alerting

– alerting, Alerting

– drift metrics table, Drift Metrics Table-Drift Metrics Table

– inference tables and, From Inference Tables to Data Quality Monitoring-Refreshing the Monitor

– metrics for monitoring data drift, Metrics for Monitoring Data Drift-Population Stability Index

– profile metrics table, Profile Metrics Table-Profile Metrics Table

Databricks extension for Visual Studio Code, Databricks Extension for Visual Studio Code-Databricks Extension for Visual Studio Code

Databricks Feature Serving, Databricks Model Serving, Feature Serving

Databricks Feature Store, Feature Engineering

Databricks Lakebase, Memory stores

Databricks Model Serving, Databricks Model Serving

– endpoint deployment, Deploying a Databricks Model Serving Endpoint-Deployment Considerations

– deployment considerations, Deployment Considerations

– deployment jobs, Deployment Jobs-Agent deployment job

– Unity AI gateway/guardrails, Unity AI Gateway and Guardrails-Calling the Model Serving endpoint

– endpoint types, Hosting Options

Databricks notebook, running code in, Running Code in a Databricks Notebook-Using all-purpose compute

– using all-purpose compute, Using all-purpose compute

– using serverless compute, Using serverless compute

Databricks Online Feature Store, Publishing a Table to a Databricks Online Feature Store

Databricks Runtime, Runtime versions

Databricks secret scopes, Security and cybersecurity risks

Databricks Unity AI Gateway, Hosting Options

Databricks-hosted models (pay-per-token), Databricks-hosted models (pay-per-token)

dataset engineering, ML versus AI, AI Project Lifecycle

dataset/context engineering, Dataset/context engineering

debugging, Debugging

Declarative Automation Bundles (DABs), Declarative Automation Bundles, Declarative Automation Bundles-Using DABs

– alerting with, Alerting

– databricks.yml file, The databricks.yml file

– databricks.yml file syntax, Syntax of the databricks.yml file-Syntax of the databricks.yml file

Delta Lake, Dataset/context engineering

demand forecasting, Regression and Demand Forecasting

dependency hijacking, Security and cybersecurity risks

deployment (see ML model deployment)

developer tools (Databricks), Databricks Developer Tools-Using DABs

– Declarative Automation Bundles, Declarative Automation Bundles-Using DABs

– databricks.yml file syntax, Syntax of the databricks.yml file-Syntax of the databricks.yml file

– using DABs, Using DABs

– running code in a Databricks notebook, Running Code in a Databricks Notebook-Using all-purpose compute

– using all-purpose compute, Using all-purpose compute

– using serverless compute, Using serverless compute

documentation

– as team practice, Documentation and Knowledge Preservation

– EU AI Act provisions, Article 11: Technical Documentation

drift metrics table, Introduction to Databricks Data Quality Monitoring, Drift Metrics Table-Drift Metrics Table

driver nodes, Worker and driver nodes

DTAP (development, testing, acceptance, and production) software development sequence, How Many Environments Do You Need?

E

embedding foundation models, Types of Foundation Models

embedding window, Embedding window considerations

end-to-end testing, End-to-end testing

endpoint deployment, Deploying a Databricks Model Serving Endpoint-Deployment Considerations

– deployment considerations, Deployment Considerations

– deployment jobs, Deployment Jobs-Agent deployment job

– Unity AI gateway/guardrails, Unity AI Gateway and Guardrails-Calling the Model Serving endpoint

EU AI Act

– Article 9: risk management system, Article 9: Risk Management System-Security and cybersecurity risks

– Article 10: data and data governance, Article 10: Data and Data Governance-Business analysis

– Article 11: technical documentation, Article 11: Technical Documentation

– Article 12: record-keeping, Article 12: Record-Keeping

– Article 13: transparency and provision of information to deployers, Article 13: Transparency and Provision of Information to Deployers

– Article 14: human oversight, Article 14: Human Oversight

– Article 15: accuracy, robustness, and cybersecurity, Article 15: Accuracy, Robustness, and Cybersecurity

– definition of AI under, Definition of AI Under the EU AI Act-The challenge of definition: A three-factor approach

– European Law Institute challenge of AI system definition in EU AI Act, The challenge of definition: A three-factor approach-The challenge of definition: A three-factor approach

– fairness/bias/regulatory compliance, Fairness, Bias, and Regulatory Compliance

– high-risk AI requirements, Requirements for High-Risk AI-Article 15: Accuracy, Robustness, and Cybersecurity

evals, LLM System Evaluation

evaluation tools, Evaluation tools

evaluation, defined, LLM System Evaluation

experiment tracking, Experiment Tracking-MLflow Run

experiments, defined, Experiment Tracking

external models, External models

F

fairness monitoring, Fairness, Bias, and Regulatory Compliance

feature drift, Using the Dashboard

feature engineering, Feature Engineering

Feature Engineering in Unity Catalog, Feature Engineering

feature function, Creating Feature Functions and a Feature Table

feature lookup, Model Serving with, Model Serving with Feature Lookup-Payload Structure

- creating feature functions and a feature table, Creating Feature Functions and a Feature Table
- payload structure, Payload Structure
- registering the model and publishing an online table, Registering the Model and Publishing an Online Table
- training set, Defining the Training Set-Defining the Training Set

Feature Serving, Feature Serving-Endpoint Deployment

feature store, Feature store

feature tables, Creating a Feature Table

FeatureSpec, Creating a FeatureSpec

foundation models, Foundation Models on Databricks-Types of Foundation Models

– hosting options, Hosting Options-External models

– types, Types of Foundation Models

functional reproducibility, Reproducibility

G

general purpose foundation models, Types of Foundation Models

general-purpose AI (GPAI) models, EU AI Act definition, Definition of AI Under the EU AI Act

generic monitoring, System and Infrastructure Health

Genie agent, Genie Agent (Formerly, Genie Space)-Interacting with Genie

Genie space

– creating, Creating the Genie agent

– Databricks AI Search versus, Databricks AI Search versus Genie

– interacting with, Interacting with Genie

– provisioning the SQL warehouse, Provisioning the SQL warehouse

GitHub flow, Git Branching Model-Git Branching Model

governance (see AI governance)

GPAI (general-purpose AI) models, EU AI Act definition, Definition of AI Under the EU AI Act

graders, for evaluating LLM systems, How to Evaluate LLM Systems

guardrails, Guardrails

– Databricks AI Guardrails, Databricks AI Guardrails

– Unity AI Gateway, Unity AI Gateway and Guardrails-Calling the Model Serving endpoint

– authentication considerations, Authentication considerations

– guardrails as part of the agent logic, Guardrails as part of the agent logic

– guardrails on the LLM level, Guardrails on the LLM level

– implementing deployment, Implementing deployment

– Model Serving endpoint, Calling the Model Serving endpoint

– service principal authentication, Setting up service principal authentication

Guardrails AI scorers, Guardrails as part of the agent logic

guidelines scorers, in MLflow GenAI evaluation framework, MLflow GenAI Evaluation Framework

H

hierarchical navigable small world (HNSW) algorithm, Databricks AI Search, Querying and retrieval quality

high-risk AI, Requirements for High-Risk AI-Article 15: Accuracy, Robustness, and Cybersecurity

– (see also risk management system, for high-risk AI)

– governance, Article 10: Data and Data Governance-Business analysis

– (see also EU AI Act)

– business analysis, Business analysis

– dataset/context engineering, Dataset/context engineering

– deployment, Deployment

– model development/adaptation, Model development/adaptation

– monitoring/continual learning, Monitoring and continual learning

– project scoping, Project scoping

human graders, How to Evaluate LLM Systems

human oversight, EU AI Act provisions, Article 14: Human Oversight

human review, Human review and feedback

hyperparameter tuning, Hyperparameter Tuning and Nested Runs-Hyperparameter Tuning and Nested Runs

I

inference tables, From Inference Tables to Data Quality Monitoring-Refreshing the Monitor, Security and cybersecurity risks

– creating a data quality monitor, Creating a Data Quality Monitor-Creating a Data Quality Monitor

– inference table structure, Inference Table Structure

– processing, Processing an Inference Table

– processing an inference table, Processing an Inference Table

– refreshing the monitor, Refreshing the Monitor

– structure, Inference Table Structure

– using the dashboard, Using the Dashboard

inference, workflow for, Workflow orchestration

integration testing, Integration testing

J

Jensen-Shannon distance (JSD), Jensen-Shannon Distance

job compute, all-purpose compute versus, All-purpose versus job compute

K

Kolmogorov-Smirnov test, Kolmogorov-Smirnov Test

L

L-infinity distance, L-Infinity Distance

Lakebase, Lakebase

– agent memory with, Agent Memory with Lakebase-Utilizing a connection pool

– interacting with, Interacting with Lakebase

– utilizing a connection pool, Utilizing a connection pool-Utilizing a connection pool

Lakebase Autoscaling, Lakebase Autoscaling for agent workloads-Creating a Lakebase Autoscaling project

Lakeflow Jobs, Lakeflow Jobs

– agent deployment job, Agent deployment job

– data preprocessing job, Data preprocessing job

– deployment jobs, Deployment Jobs-Agent deployment job

– LLM system monitoring, Lakeflow job

– serverless versus classic compute, Serverless versus Classic Compute for Lakeflow Jobs

large language models (see LLM entries)

linting, Linting and formatting

LLM application components, LLM Application Components-Human review and feedback

– AI gateways, AI gateways

– evaluation tools, Evaluation tools

– guardrails, Guardrails

– human review/feedback, Human review and feedback

– memory stores, Memory stores

– observability tools, Observability tools

– vector databases, Vector databases

LLM judges, How to Evaluate LLM Systems

LLM-as-a-Judge, MLflow GenAI Evaluation Framework

LLMOps, Foundation Models and Context Engineering-Conclusion

– agents, What Is an Agent?

– basics, Introduction to LLMOps-Structure of the LLMOps Chapters

– context engineering, Context Engineering-Utilizing a connection pool

– agent memory with Lakebase, Agent Memory with Lakebase-Utilizing a connection pool

– conversion of raw data to chunks, Getting from Raw Data to Chunks-Processing chunks

– Databricks AI Search index, Databricks AI Search Index-Optimizing search quality

    - Genie agent, Genie Agent (Formerly, Genie Space)-Interacting with Genie
    - MCP servers/tool calling, MCP Servers on Databricks and Tool Calling-Implementing tool calling
  - foundation models on Databricks, Foundation Models on Databricks-Types of Foundation Models
    - hosting options, Hosting Options-External models
    - types, Types of Foundation Models

LLMs (large language models), Large Language Models, LLM System Monitoring-Cost-monitoring dashboard

  - cost monitoring, Cost Monitoring-Cost-monitoring dashboard
    - dashboard, Cost-monitoring dashboard-Cost-monitoring dashboard
    - Databricks billing, How Databricks billing works
    - Databricks services/associated costs, Databricks services and associated costs
    - tag-based cost attribution, Tag-based cost attribution
  - monitoring, Large Language Models
  - system evaluation, LLM System Evaluation-MLflow GenAI Evaluation Framework
  - system monitoring, Monitoring Implementation-Monitoring considerations
    - considerations, Monitoring considerations
    - dashboard, Monitoring dashboard
    - deployment notebook, Deployment notebook

– Lakeflow job, Lakeflow job

– what to monitor, What to Monitor in LLM Systems

M

machine learning (see ML entries)

machine learning engineer, The Rise of MLOps

Machine Learning, Analytics, and Data (MAD) landscape, MLOps Components-MLOps Components

managed MCP ecosystems, Managed MCP ecosystems on Databricks-Managed MCP ecosystems on Databricks

maximum absolute difference, L-Infinity Distance

MCP (Model Context Protocol), Databricks Apps, MCP Servers on Databricks and Tool Calling-Implementing tool calling

MCP servers, MCP Servers on Databricks and Tool Calling-Implementing tool calling

Mem0, Memory stores

memory stores, Memory stores

metrics, for evaluating LLM systems, How to Evaluate LLM Systems

ML model deployment, Machine Learning Model Deployment-Conclusion

– Databricks platform design, Databricks Platform Design-Access Control Strategy

– access control strategy, Access Control Strategy-Access Control Strategy

– number of environments required, How Many Environments Do You Need?-How Many Environments Do You Need?

    – platform best practices, Platform Best Practices

– machine learning pipeline, Machine Learning Pipeline-Serverless versus Classic Compute for Lakeflow Jobs

    – data preprocessing, Data Preprocessing-Data Preprocessing

    – model-serving endpoint deployment, Deploying a Model Serving Endpoint

    – serverless versus classic compute for Lakeflow jobs, Serverless versus Classic Compute for Lakeflow Jobs

    – training/evaluating/registering models, Training, Evaluating, and Registering a Model

– promoting code to higher environments, Promoting Code and Models to Higher Environments-Branch and deployment protection rules

    – CD pipeline, Continuous Deployment Pipeline-Branch and deployment protection rules

    – CI pipeline, Continuous Integration Pipeline

    – git branching model, Git Branching Model-Git Branching Model

– promoting models to higher environments, Promoting Models to Higher Environments

– testing, Testing-End-to-end testing

ML model testing, ML Model Testing

MLflow, MLflow

– logging/registering models

    – model logging with the sklearn flavor, Model Logging with the sklearn Flavor-Model Logging with the sklearn Flavor

    – registering the model, Registering the Model

– viewing logged model, Viewing the Logged Model-Viewing the Logged Model

– wrapping the model using pyfunc, Wrapping the Model Using pyfunc-Wrapping the Model Using pyfunc

– span types, Anatomy of a Trace

– traditional ML, MLflow for Traditional ML-Conclusion

– experiment tracking, Experiment Tracking-MLflow Run

– hyperparameter tuning/nested runs, Hyperparameter Tuning and Nested Runs-Hyperparameter Tuning and Nested Runs

– logging/registering models, Logging and Registering Models with MLflow-Wrapping the Model Using pyfunc

– MLflow evaluation, MLflow Evaluation-MLflow Evaluation

– resource/API limits, Resource and API Limits

– training data, Training Data-Training Data

MLflow evaluation tools, MLflow for Traditional ML

MLflow experiment tracking, Experiment Tracking-MLflow Run

– MLflow experiment, MLflow Experiment

– MLflow run, MLflow Run-MLflow Run

– tracking URI, Tracking URI

MLflow GenAI, MLflow for GenAI-Conclusion

– agent logging, Agent Logging

– defining custom agents, Defining a Custom Agent-Testing the Agent

– core logic, Core Agent Logic-Core Agent Logic

– initialization/async handling, Initialization and Async Handling

– prediction/streaming, Prediction and Streaming

– testing, Testing the Agent

– evaluation in practice, Evaluating, Logging, and Registering the Agent

– LLM system evaluation, LLM System Evaluation-MLflow GenAI Evaluation Framework

– MLflow flavors, MLflow Flavors

– MLflow tracing, MLflow Tracing-MLflow Tracing APIs

– anatomy of a trace, Anatomy of a Trace

– MLflow tracing APIs, MLflow Tracing APIs

– registering the agent/preparing for serving, Registering the Agent and Preparing for Serving

MLflow Model, Logging and Registering Models with MLflow

MLflow Model Registry, MLflow for Traditional ML

MLflow run, MLflow Run-MLflow Run

MLflow tracing, MLflow for Traditional ML, MLflow Tracing-MLflow Tracing APIs, Security and cybersecurity risks

MLflow Tracking, MLflow for Traditional ML

MLOps (generally), MLOps Principles and Components-Conclusion

– checklist, MLOps Checklist

– components, MLOps Components-Choosing Your Tools, Core Components-Feature store

– CI/CD, Continuous integration and continuous deployment (CI/CD)

– code version control, Code version control

    - core components, Core Components-Feature store
    - data version control, Data version control
    - feature store, Feature store
    - LLM application components, LLM Application Components-Human review and feedback
    - Model Serving, Model Serving
    - model training, Model training
    - model/container registry, Model and container registry
    - monitoring, Monitoring
    - tools, Choosing Your Tools
    - workflow orchestration, Workflow orchestration
- Databricks MLOps components, Databricks MLOps Components-Databricks Apps
    - cross-cutting components, Cross-Cutting Components-Databricks Apps
    - Data Quality Monitoring, Data Quality Monitoring
    - Databricks AI Guardrails, Databricks AI Guardrails
    - Databricks AI Search, Databricks AI Search
    - Databricks compute, Databricks Compute
    - Databricks Model Serving, Databricks Model Serving
    - feature engineering, Feature Engineering
    - Lakeflow Jobs, Lakeflow Jobs
    - MLflow, MLflow

– Unity AI Gateway, Unity AI Gateway

– definition, MLOps Principles and Components

– principles, MLOps Principles-Observability

– observability, Observability

– reliability, Reliability

– reproducibility, Reproducibility

– traceability, Traceability

– rise of MLOps, The Rise of MLOps

– team practices, Team Practices-Code Hygiene

– code hygiene, Code Hygiene

– documentation/knowledge preservation, Documentation and Knowledge Preservation

MLOps checklist, MLOps Checklist

model behavior risks, Data-related and model behavior risks

Model Context Protocol (MCP), Databricks Apps, MCP Servers on Databricks and Tool Calling-Implementing tool calling

model deployment

– ML versus AI, AI Project Lifecycle

– responsibility for, Model Serving: Architectures and Implementation

model development, ML versus AI, AI Project Lifecycle

model logging, Model Logging with the sklearn Flavor-Model Logging with the sklearn Flavor

model registering, Registering the Model

model registry, Model and container registry

Model Serving, Model Serving, Model Serving: Architectures and Implementation-Conclusion

– A/B testing, A/B Testing-Moving assignment into the model logic

– debugging, Debugging

– endpoint deployment, Endpoint Deployment-Endpoint Deployment

– Feature Serving, Feature Serving-Endpoint Deployment

– creating a feature table, Creating a Feature Table

– creating a FeatureSpec, Creating a FeatureSpec

– endpoint deployment, Endpoint Deployment

– publishing a table to a Databricks Online Feature Store, Publishing a Table to a Databricks Online Feature Store

– Model Serving with feature lookup, Model Serving with Feature Lookup-Payload Structure

– creating feature functions and a feature table, Creating Feature Functions and a Feature Table

– payload structure, Payload Structure

– registering the model and publishing an online table, Registering the Model and Publishing an Online Table

– training set, Defining the Training Set-Defining the Training Set

model training, Model training

model wrapping, Wrapping the Model Using pyfunc-Wrapping the Model Using pyfunc

model-based graders, How to Evaluate LLM Systems

monitoring

– defined, Observability

– EU AI Act provisions, Monitoring and continual learning

– ML versus AI, AI Project Lifecycle

monitoring LLM-based systems, Deploying in Monitoring LLM-Based Systems-Conclusion

– Databricks Model Serving endpoint, Deploying a Databricks Model Serving Endpoint-Deployment Considerations

– deployment considerations, Deployment Considerations

– deployment jobs, Deployment Jobs-Agent deployment job

– Unity AI gateway/guardrails, Unity AI Gateway and Guardrails-Calling the Model Serving endpoint

– LLM system monitoring, LLM System Monitoring-Cost-monitoring dashboard

– cost monitoring, Cost Monitoring-Cost-monitoring dashboard

– monitoring implementation, Monitoring Implementation-Monitoring considerations

– what to monitor, What to Monitor in LLM Systems

monitoring ML applications, Monitoring ML Applications-Conclusion

– Databricks Data Quality Monitoring, Introduction to Databricks Data Quality Monitoring-Alerting

– alerting, Alerting

– drift metrics table, Drift Metrics Table-Drift Metrics Table

– inference tables and Data Quality Monitoring, From Inference Tables to Data Quality Monitoring-Refreshing the Monitor

– metrics for monitoring data drift, Metrics for Monitoring Data Drift-Population Stability Index

– profile metrics table, Profile Metrics Table-Profile Metrics Table

– what to monitor, What to Monitor-System and Infrastructure Health

– concept drift versus data drift, Concept Drift versus Data Drift

– fairness/bias/regulatory compliance, Fairness, Bias, and Regulatory Compliance

– LLMs, Large Language Models

– recommender systems, Recommender Systems

– regression/demand forecasting, Regression and Demand Forecasting

– system/infrastructure health, System and Infrastructure Health

N

nanny model, Regression and Demand Forecasting

natural language queries, Querying and retrieval quality

nested runs, Hyperparameter Tuning and Nested Runs-Hyperparameter Tuning and Nested Runs

O

OAuth authentication, OAuth Authentication, Security and cybersecurity risks

observability

– as MLOps principle, Observability

– tools, Observability tools

observable system, defined, Observability

OpenID connect, OpenID Connect

operational risks, Operational risks

Optuna, Hyperparameter Tuning and Nested Runs

P

pay-per-token models, Databricks-hosted models (pay-per-token)

PDF processing, PDF processing-PDF processing

Photon, Photon acceleration

platform design, Databricks Platform Design-Access Control Strategy

- access control strategy, Access Control Strategy-Access Control Strategy
- best practices, Platform Best Practices

Population Stability Index (PSI), Population Stability Index

post-market monitoring, Post-market monitoring (via a feedback loop)

post-train tests, ML Model Testing

pre-train tests, ML Model Testing

principles, MLOps, MLOps Principles-Observability

- observability, Observability
- reliability, Reliability
- reproducibility, Reproducibility
- traceability, Traceability

profile metrics, Profile Metrics Table-Profile Metrics Table

profile metrics table, Introduction to Databricks Data Quality Monitoring

project scoping, Project scoping

project setup (Databricks), Project Setup-Project Code: Data Preprocessing

protected group, Profile Metrics Table

provisioned throughput endpoints, Provisioned throughput endpoints

PSI (Population Stability Index), Population Stability Index

pyfunc, model wrapping with, Wrapping the Model Using pyfunc-Wrapping the Model Using pyfunc

pyproject.toml file, The pyproject.toml file-Pytest configurations

– build-system section, Build-system section

– linting/formatting, Linting and formatting

– project section, Project section

– pytest configurations, Pytest configurations

PySpark, Databricks Connect-Databricks Connect

Python

– pyproject.toml file, The pyproject.toml file-Pytest configurations

– tooling, Python Tooling

Q

quality management systems, mandated under EU AI Act, Requirements for High-Risk AI

R

Ray Tune, Hyperparameter Tuning and Nested Runs, Hyperparameter Tuning and Nested Runs

real-time inference, Workflow orchestration

reasoning foundation models, Types of Foundation Models

recommender systems, Recommender Systems

record-keeping, EU AI Act provisions, Article 12: Record-Keeping

recursive split, Chunking strategies

regression, Regression and Demand Forecasting

regulatory compliance monitoring, Fairness, Bias, and Regulatory Compliance

reliability (MLOps principle), Reliability

(R)REPEATS (Reproducibility, Robustness, Equitability, Privacy, Explainability, Accountability, Transparency, and Security) principles framework, MLOps Principles

repojacking, Security and cybersecurity risks

reproducibility (MLOps principle), Reproducibility

reranking, Querying and retrieval quality

retrieval-related scorers, MLflow GenAI Evaluation Framework

risk management system, for high-risk AI, Article 9: Risk Management System-Security and cybersecurity risks

- – (see also EU AI Act)
- – development/testing as mitigation, Development and testing (mitigation)

– post-market monitoring via feedback loop, Post-market monitoring (via a feedback loop)

– project scoping/risk identification, Project scoping (risk identification)

– types of risks and mitigation strategies, Types of risks and mitigation strategies-Security and cybersecurity risks

– data-related/model behavior risks, Data-related and model behavior risks

– operational risks, Operational risks

– security/cybersecurity risks, Security and cybersecurity risks

risk-based approach to regulation, Risk Categories and Whom They Apply To

risks, of AI, Types of risks and mitigation strategies-Security and cybersecurity risks

robustness, EU AI Act provisions, Article 15: Accuracy, Robustness, and Cybersecurity

Ruff, linting/formatting with, Linting and formatting

runtime versions, Runtime versions

S

scorers, in MLflow GenAI evaluation framework, MLflow GenAI Evaluation Framework-MLflow GenAI Evaluation Framework

securables, Access Control Strategy

security risks, Security and cybersecurity risks

serverless compute, Serverless Compute, Serverless versus Classic Compute for Lakeflow Jobs

– classic compute versus, Classic compute versus serverless compute

– cost tracking, Cost tracking

– Databricks notebook, Using serverless compute

– serverless environments, Serverless environments

service principal authentication, Setting up service principal authentication

serving limitations, Serving Limitations

session persistence, Agent Memory with Lakebase

SIMBA (Stochastic Introspective Mini-Batch Ascent) optimizer, MLflow GenAI Evaluation Framework

sklearn, Model Logging with the sklearn Flavor-Model Logging with the sklearn Flavor

software testing, Software Testing-End-to-end testing

– Databricks considerations, Databricks considerations

– end-to-end testing, End-to-end testing

– integration testing, Integration testing

– integration tests in CI, Practical considerations: Integration tests in CI

– unit testing, Unit testing

span objects, in MLflow, Anatomy of a Trace

SPNs (service principals), Access Control Strategy

SQL warehouse, Provisioning the SQL warehouse

Stochastic Introspective Mini-Batch Ascent (SIMBA) optimizer, MLflow GenAI Evaluation Framework

T

tags, in MLflow, Anatomy of a Trace

team practices, Team Practices-Code Hygiene

– code hygiene, Code Hygiene

– documentation/knowledge preservation, Documentation and Knowledge Preservation

technical documentation, EU AI Act provisions, Article 11: Technical Documentation

template-based LLM scorers, MLflow GenAI Evaluation Framework

test pyramid, Software Testing-Software Testing

testing (see specific forms of testing)

tiered evaluation strategy, LLM System Monitoring

time-based splits, Training Data

tool calling, Implementing tool calling

tool-calling scorers, MLflow GenAI Evaluation Framework

total variation distance (TVD), Total Variation Distance

trace, in MLflow, Anatomy of a Trace

traceability (MLOps principle), Traceability

tracking URI, Tracking URI

traffic split, A/B Testing

training data, for traditional ML, Training Data-Training Data

training set, for Model Serving with feature lookup, Model Serving with Feature Lookup-Payload Structure

transparency, EU AI Act provisions, Article 13: Transparency and Provision of Information to Deployers

TVD (total variation distance), Total Variation Distance

typosquatting, Security and cybersecurity risks

U

UDFs (user-defined functions), Creating Feature Functions and a Feature Table

unit testing, Unit testing

Unity AI Gateway, Unity AI Gateway, Unity AI Gateway and Guardrails-Calling the Model Serving endpoint

- authentication considerations, Authentication considerations
- guardrails as part of the agent logic, Guardrails as part of the agent logic
- guardrails on the LLM level, Guardrails on the LLM level
- implementing deployment, Implementing deployment
- Model Serving endpoint, Calling the Model Serving endpoint
- service principal authentication, Setting up service principal authentication

Unity Catalog, Unity Catalog, Databricks Platform Design-Databricks Platform Design, Dataset/context engineering

Unity Catalog securables, Access Control Strategy

unprotected group, Profile Metrics Table

usage policy, Tag-based cost attribution

V

vector databases, Vector databases

version control, Code version control, Data version control

vision foundation models, Types of Foundation Models

Visual Studio Code, Databricks extension for, Databricks Extension for Visual Studio Code-Databricks Extension for Visual Studio Code

W

Wasserstein distance, Wasserstein Distance

worker nodes, Worker and driver nodes

workflow orchestration, Workflow orchestration

workplace securables, Access Control Strategy

About the Author

Maria Vechtomova is a co-founder of Cauchy, where she specializes in AI strategy, AI engineering, and AI training. She has worked in data and AI since 2014, across large corporate organizations and in roles ranging from data analyst and data scientist to MLOps tech lead. She has focused on MLOps for most of her career, since long before the term was widely used.

Maria believes that AI delivers value only when it is in production, integrated into business processes, and supported by reliable systems. She is passionate about teaching best practices and has taught MLOps and LLMOps to more than three hundred students through online cohort-based courses. She has extensive experience with Databricks and has been recognized as a Databricks MVP and MLflow Ambassador.

Colophon

The animal on the cover of *MLOps with Databricks* is a Eurasian coot (*Fulica atra*), a waterbird found in the freshwater lakes, ponds, marshes, and slow-moving rivers of Europe, Asia, Australia, and parts of North Africa.

The Eurasian coot has sooty black plumage, a bright white bill, and a distinctive white frontal shield on its forehead. It has red eyes and large, lobed toes that help it swim and walk across soft, marshy ground.

Feeding primarily on aquatic plants and algae (but also insects, mollusks, small fish, crustaceans, and amphibians), they forage by grazing on vegetation or diving underwater to reach their food.

Although often seen calmly swimming on open water, Eurasian coots can be highly territorial, especially during the breeding season. They frequently chase rivals across the water while flapping their wings and using their feet to splash aggressively.

While the Eurasian coot is classified as Least Concern by the International Union for Conservation of Nature (IUCN), many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Monica Kamsvaag, based on an antique line engraving from *Meyers Kleines Lexicon*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.

This file was downloaded from Z-Library project



Your gateway to knowledge and culture. Accessible for everyone.

z-library.sk z-lib.gs z-lib.fm go-to-library.sk

Official Telegram channel

Z-Access

https://wikipedia.org/wiki/Z-Library