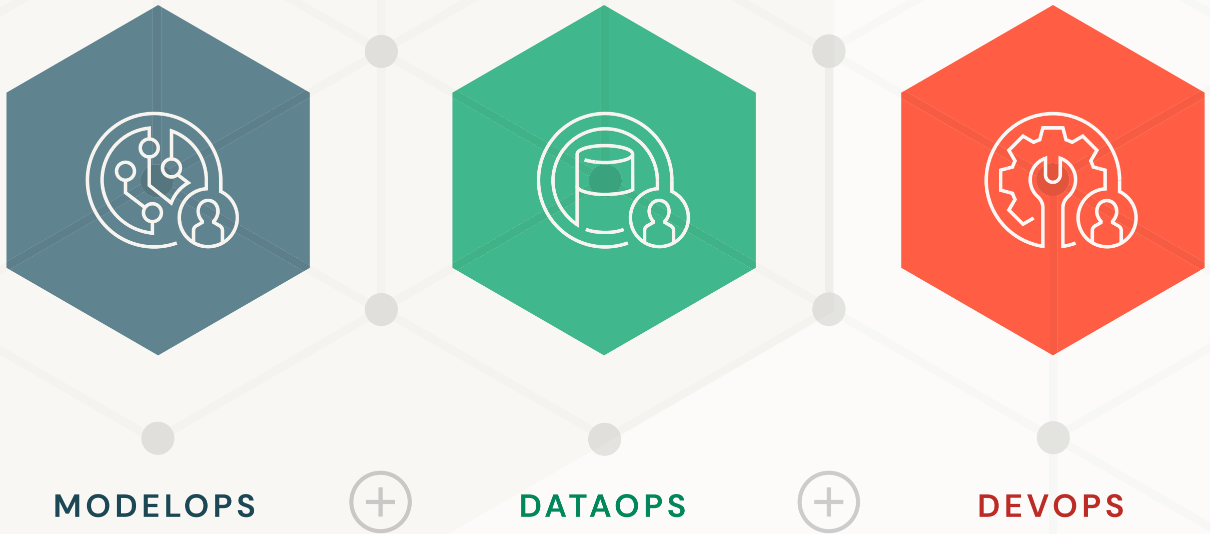
2ND EDITION

eBook

The Big Book of MLOps

NOW INCLUDING A SECTION ON LLMOPS



databricks

JOSEPH BRADLEY | RAFI KURLANSIK | MATT THOMSON | NIALL TURBITT

Contents

Contents

Contents

databricks

CHAPTER 1

Introduction

Machine learning operations (MLOps) is a rapidly evolving field where building and maintaining robust, flexible and efficient workflows is critical. At Databricks, we view MLOps as the set of processes and automation for managing data, code and models to improve performance stability and long-term efficiency in ML systems. Distilling this into a single equation:

> MLOps = DataOps + DevOps + ModelOps

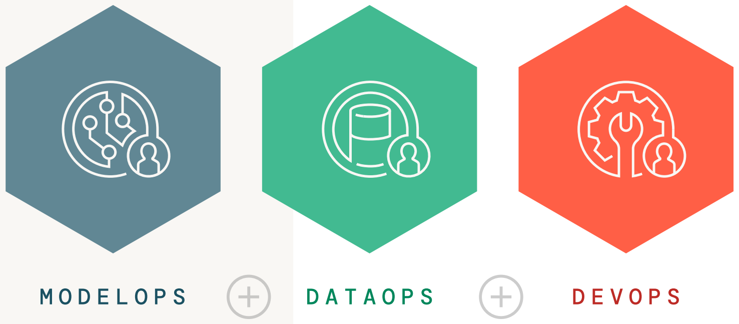
Through this lens, we strive to continuously innovate and advance our product offerings to simplify the ability to build AI-powered solutions on the Lakehouse. We believe there is no greater accelerant to delivering ML to production than building on a unified, data-centric AI platform. On Databricks, both data and models can be managed and governed in a single governance solution in the form of Unity Catalog. The previously complex infrastructure required to serve real-time models can now be replaced and easily scaled with Databricks Model Serving. Long-term efficiency and performance stability of ML in production can be achieved using Databricks Lakehouse Monitoring. These components collectively form the data pipelines of an ML solution, all of which can be orchestrated using Databricks Workflows.

Perhaps the most significant recent change in the machine learning landscape has been the rapid advancement of generative AI. Generative models such as large language models (LLMs) and image generation models have revolutionized the field, unlocking previously unattainable levels of natural language and image generation. However, their arrival also introduces a new set of challenges and decisions to be made in the context of MLOps.

With all these developments in mind, we're excited to present this updated version of the Big Book of MLOps. This guide incorporates new Databricks features such as Models in Unity Catalog, Model Serving, and Lakehouse Monitoring into our MLOps architecture recommendations. We start by outlining the themes that still remain relevant from the previous version of the Big Book of MLOps. Following this, we unpack the new features introduced in this version, their impact on the previous reference architecture, and best practices when incorporating these into your MLOps workflows. Next, we present our updated MLOps reference architecture, along with the details of its processes. Finally, we provide guidance for deploying generative AI applications to production on Databricks, focusing on productionizing LLMs.

CHAPTER 2

Big Book of MLOps V1 Recap



We begin with a brief recap of the core points discussed in the previous version of the Big Book of MLOps. While the recommended reference architecture has evolved due to new features and product updates, the core themes discussed, such as the importance of MLOps, guiding principles and the fundamentals of MLOps on Databricks, remain pertinent. In this section we focus on summarizing those elements that remain unchanged. For a more in-depth discussion of any of these points, we refer the reader to last year's Big Book of MLOps.

Why should I care about MLOps?

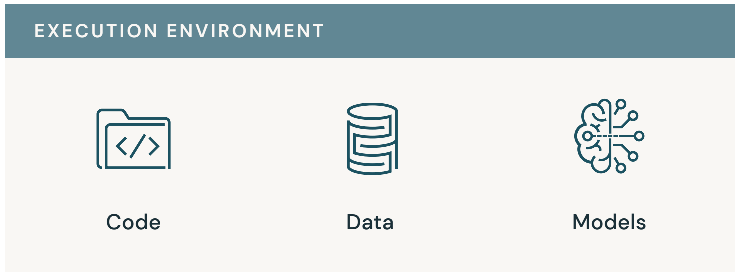
We continue to stress the importance of defining an effective MLOps strategy. Databricks customers like CareSource, which has since implemented our recommended MLOps architecture, have witnessed firsthand the value this can bring. Through streamlining the process of delivering models to production, time to business value is accelerated. This efficiency has the knock-on effect of giving data science teams the freedom and confidence to transition to subsequent projects without the need for continuous manual oversight of models in production.

Guiding principles

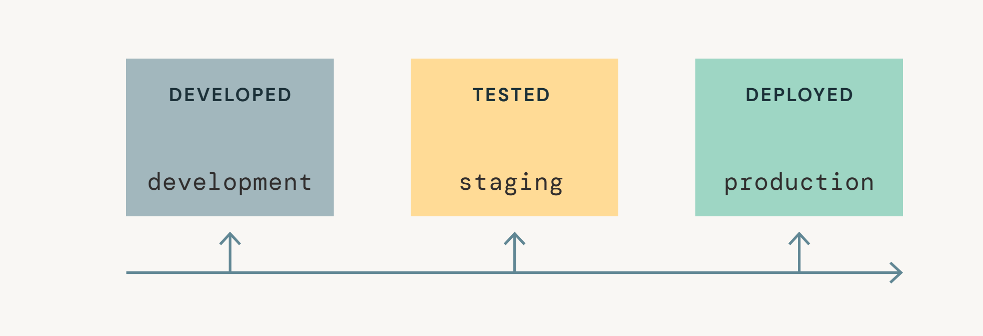
One guiding principle that continues to lie at the heart of the Lakehouse AI vision is taking a data-centric approach to machine learning. With the increasing prevalence of generative AI, this perspective remains just as important. The core constituents of any ML project can be viewed simply as data pipelines: feature engineering, training, model deployment, inference and monitoring pipelines are all data pipelines. As such, operationalizing an ML solution requires joining data from predictions, monitoring and feature tables with other relevant data. Fundamentally, the simplest way to achieve this is to develop AI-powered solutions on the same platform used to manage production data.

Note: Throughout this paper we operate under the assumption of three distinct execution environments — development, staging and production — in the form of three separate Databricks workspaces. There can be variations of these three stages, such as alternative naming conventions or splitting staging into separate "test" and "QA" substages. Although not recommended, it is also possible to create three distinct environments within a single Databricks workspace through the use of access controls and Git branches. Regardless of how environment separation is achieved, the core principles of the workflow and recommendations presented are generally applicable.

Semantics of development, staging and production



An ML solution comprises data, code and models. These assets need to be developed, tested (staging) and deployed (production). For each of these stages, we also need to operate within an execution environment. As such, each of data, code, models and execution environments are notionally divided into development, staging and production.

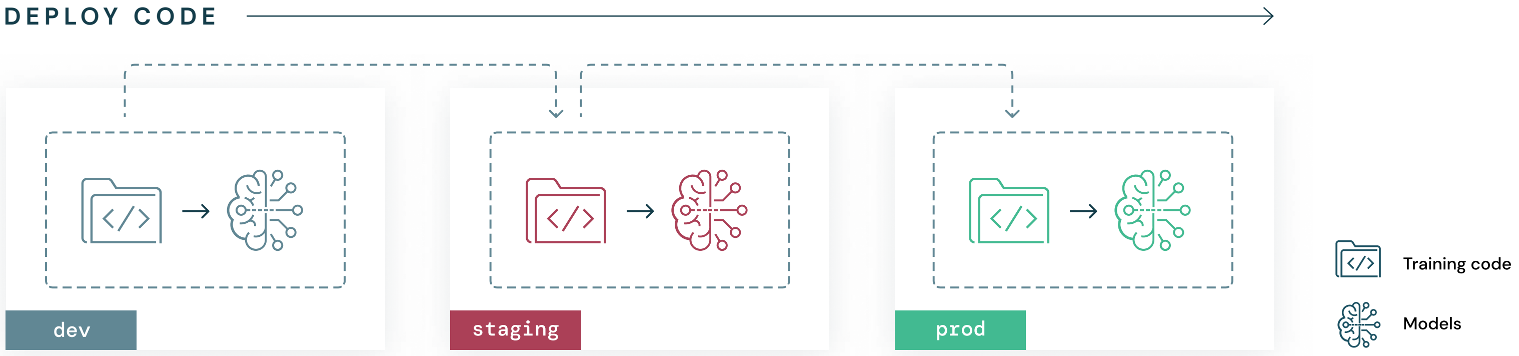


Each of these stages has distinct access controls and quality guarantees, ranging from the open and exploratory development stage through to the locked-down and quality-assured production stage.

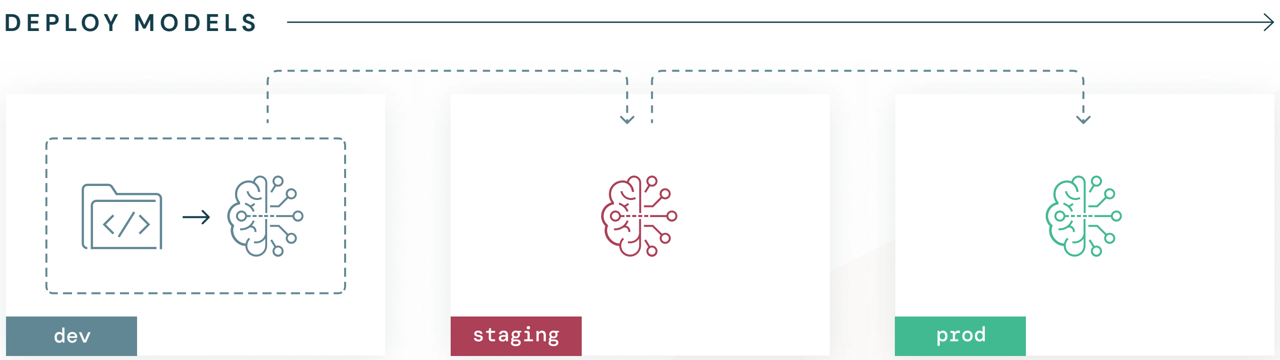
ML deployment patterns

Code and models often progress asynchronously through these stages. Thus, it becomes crucial to leverage a solution that allows for the management of model artifacts independently of code, making it possible to update a production model without necessarily making a code change. Data, much like code and models, can be labeled as development, staging or production, indicating not only its origin but also its quality and reliability.

Given the independent lifecycles of code and models, there are two opposing strategies to moving code and ML models from development, through staging and subsequently to production:



- Code for an ML project is developed in the development environment, and this code is then moved to the staging environment, where it is tested. Following successful testing, the project code is deployed to the production environment, where it is executed.
- Model training code is tested in the staging environment using a subset of data, and the model training pipeline is executed in the production environment
- The model deployment process of validating a model and additionally conducting comparisons versus any existing production model all run within the production environment



- Model training is executed in the development environment. The produced model artifact is then moved to the staging environment for model validation checks, prior to deployment of the model to the production environment.
- This approach requires a separate path for deploying ancillary code such as inference and monitoring code. Subsequently, any pipelines that need to run in the production environment to support the operationalization of the model will necessarily need to go through a separate "deploy code" lifecycle — the code for these components being tested in staging and then deployed to production.
- This pattern is typically used when deploying a one-off model, or when model training is expensive and read-access to production data from the development environment is possible

As in our prior paper, we **recommend a deploy code approach for the majority of use cases**, and the reference architecture presented in this update continues to follow this recommendation.

CHAPTER 3

What's New?

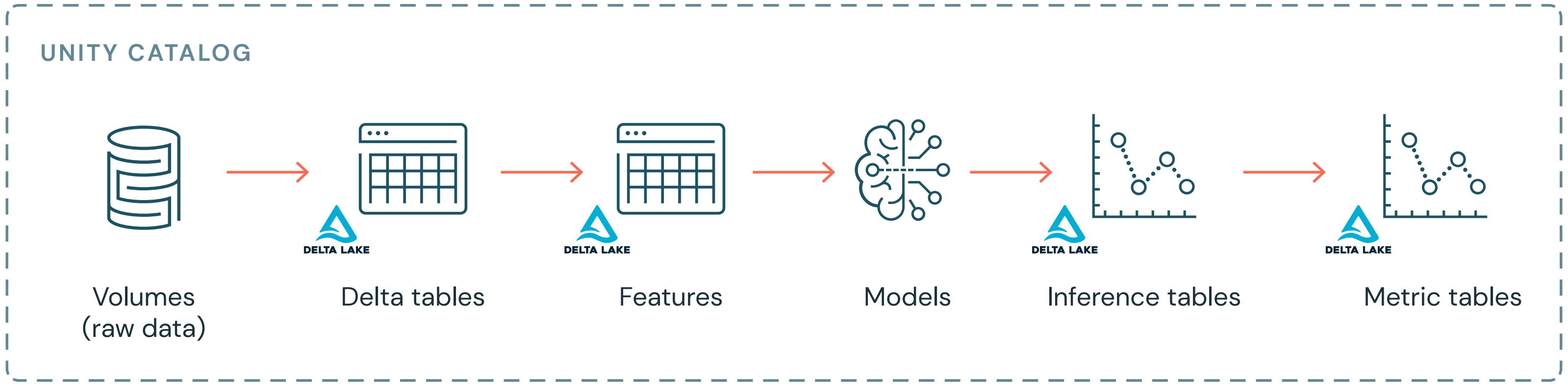
In this section we outline the key features and product updates introduced into our updated MLOps reference architecture. For each of these, we highlight the benefits they bring and how they impact our end-to-end MLOps workflow.

Unity Catalog

The Lakehouse forms the foundation of a data-centric AI platform. Key to this is the ability to manage both data and AI assets from a unified governance solution on the Lakehouse. Databricks Unity Catalog enables this by providing centralized access control, auditing, lineage, and data discovery capabilities across Databricks workspaces.

These benefits are now extended to MLflow models with the introduction of Models in Unity Catalog. By providing a hosted version of the MLflow Model Registry in Unity Catalog, the full lifecycle of an ML model can be managed while leveraging Unity Catalog's capability to share assets across Databricks workspaces and trace lineage across both data and models.

In addition to managing ML models, feature tables are also a part of Unity Catalog. With Feature Engineering in Unity Catalog, any Delta table in Unity Catalog that has been assigned a primary key (and additionally a timestamp key) can be used as a source of features to train and serve models. Furthermore, feature tables can now also be shared across different workspaces, and lineage recorded between other assets in the Lakehouse.



Assets of an ML workflow, all managed via Unity Catalog

BENEFITS AND ARCHITECTURE IMPLICATIONS

Unified governance

- Unity Catalog unlocks the ability to apply the same governance and security policies to both data and models. Consolidating these assets into a single, unified solution facilitates efficient and secure management, and more generally simplifies the overall MLOps solution.
- From an architecture design standpoint, this means the flexibility to govern both data and AI assets under the same namespace, enabling management at the environment, project or team level.

Read access to production assets

- With Unity Catalog, data and models are governed at the account level, promoting easy sharing of assets across Databricks workspaces. Data scientists in the development environment can be granted read-only access to data and AI assets from the production environment. The specifics of cross-workspace permissions can be adjusted for each project or organization.
- The implications of this are multifold: Data scientists can train models using production data, detect and debug model quality degradation by examining production monitoring tables, deep dive on model predictions using production inference tables, and easily compare in-development models with live production models. This not only accelerates the model development process but also improves the robustness and quality of the developed models.

Models in Unity Catalog

- It is now recommended to register Models in Unity Catalog in place of the classic Workspace Model Registry. Models will be registered under a three-level name in the form `<catalog>.<schema>.<model>` (see Organizing data and AI assets for a more detailed walkthrough of this).
- This three-level name provides the ability to inherently express the environment a model was produced in and apply associated governance permissions at each level of the catalog/schema/registered model hierarchy. For example, permissions can be configured such that all users have read-only access to models in the prod catalog, only ML team service principals can create new models in the use case–specific prod.fraud_detection schema, and all members of the fraud detection team can execute model versions in the `prod.fraud_detection.fraud_clf` registered model.

- The model alias — a mutable, named reference to a version of a registered model — can be used to mark a model for deployment. Inference workloads can be defined to target a specific alias. For example, you could assign the "Champion" alias of the "fraud_clf" registered model to the model version that should serve production traffic. The inference workload of the fraud detection solution would then target that alias, using the "Champion" model to make predictions.

Lineage

- With Unity Catalog, a robust link between data and AI assets can natively be recorded. Lineage can be traced from a model version in Unity Catalog back to the data used for training. Additionally, downstream lineage records consumers of assets in Unity Catalog. For example, consumers of a registered model could include Model Serving endpoints or Workflows, facilitating impact analysis if a regression is detected in the model.
- This lineage also extends from a model version to the underlying MLflow run in MLflow Tracking. Parameters, metrics, artifacts and Git information can all be tracked to this MLflow run. Additionally, metadata detailing data sources, notebooks, jobs and endpoints associated with the assets will be recorded automatically for both tables and models stored in Unity Catalog. Designing an MLOps implementation to leverage MLflow during model training ensures that this full lineage can be captured from the beginning of a model's lifecycle.

Discoverability

- Through centralizing data and AI assets in a single solution, Unity Catalog enhances the discoverability of these assets, making it simpler and quicker to locate and utilize the appropriate resources for a particular component of the MLOps solution. For example, teammates with access to view each other's models can see which data sources are being used, and use that information to address their own use cases.

Model Serving

Databricks Model Serving provides a production-ready, serverless solution to simplify real-time ML model deployment. With Model Serving, it is possible to efficiently deploy models as an API so that you can integrate model predictions with applications or websites. Given the complexity that is often involved with deploying a real-time ML model, Model Serving reduces operational costs, streamlines the ML lifecycle, and makes it easier for data science teams to focus on the core task of integrating production-grade real-time ML into their solutions.

BENEFITS AND ARCHITECTURE IMPLICATIONS

Lakehouse native

- Model Serving natively integrates with other components of the Lakehouse platform, such as Unity Catalog and MLflow. When a registered MLflow model is served, request-response payloads from Model Serving endpoints can be automatically logged to inference tables in Unity Catalog. In addition to the lineage captured from an inference table back to a registered model, we also gain the ability to implement model monitoring (see Lakehouse Monitoring). From an operational perspective, these production inference tables can be made available to data scientists in the development environment.
- Given its close integration with MLflow, Model Serving can automatically build a container from a logged MLflow model and deploy the model as a REST endpoint. This abstracts away what would ordinarily be a notably more complex process for the user.
- If models are trained using ML features from Unity Catalog, Model Serving uses lineage in Unity Catalog to automatically serve features for batch and online serving

Simplified deployment

- Data scientists or ML engineers can easily create a Model Serving endpoint from a model version without requiring extensive infrastructure knowledge or experience. Thus, creating or updating a Model Serving endpoint becomes a trivial additional step in the model deployment pipeline.

High availability and scalability

- Built for production use, Model Serving supports very low latency (p50 overhead latency of less than 10ms) and high query volumes (QPS of greater than 25k), and can automatically scale up and down based on demand. This ensures optimal performance and cost-efficiency.
- Model deployment configurations should be robustly tested prior to exposing a real-time model endpoint to production traffic to meet predefined service level agreements (SLAs). As such, we must factor such tests into our MLOps reference architecture. We unpack the different types of testing that can be conducted below in Testing Model Serving. We also illustrate how this testing is incorporated into an MLOps workflow in the Reference Architecture section.

Online evaluation

- Serving real-time models often involves online evaluation, in addition to standard offline model evaluation approaches typical of batch or streaming inference pipelines. Your choice of online evaluation strategies such as A/B testing or canary deployments can affect the design and implementation of pipelines being deployed to production. Model Serving facilitates the implementation of such online evaluation approaches through the ability to serve multiple models to a Model Serving endpoint.
- The design decisions taken around online evaluation will depend on use case requirements. We explore these in more detail in the Model deployment section below.

Secure and cost effective

- Models are deployed in a secure network boundary. Model Serving leverages serverless compute that will terminate when the model is deleted, or scaled down to zero.

Lakehouse Monitoring

Databricks Lakehouse Monitoring is a data-centric monitoring solution to ensure that both data and AI assets are of high quality, accurate and reliable. Built on top of Unity Catalog, it provides the unique ability to implement both data and model monitoring while maintaining lineage between the data and AI assets of an MLOps solution. This unified and centralized approach to monitoring greatly simplifies the process of diagnosing errors, detecting quality drift and performing root cause analysis.

BENEFITS AND ARCHITECTURE IMPLICATIONS

Lakehouse native

- Lakehouse Monitoring integrates with Unity Catalog to automatically write monitoring tables to the Lakehouse — computed metrics are stored in Delta tables called metric tables. All the advantages of Unity Catalog as outlined previously thus apply. A key benefit is the streamlined process of joining monitoring tables with others in the Lakehouse, providing an efficient way to cross-analyze data. Users can perform these analyses using SQL, and present results in both dashboards and notebooks.
- The core aim of Lakehouse Monitoring is to expedite the ability to detect and diagnose deviations in data or model quality. Thus, configuring appropriate permissions on production monitoring tables and dashboards such that they are accessible to data scientists should be factored into the production deployment process.
- To keep monitoring tables up to date, Lakehouse Monitoring APIs allow you to schedule a refresh of the metric tables on a regular basis

Monitoring with Model Serving

- Lakehouse Monitoring has tight integration with Model Serving, in particular allowing the ability to monitor inference tables produced by a Model Serving endpoint. With this integration, a pipeline can be created to process logged requests and responses from a Model Serving inference table, join with other relevant tables in the Lakehouse (e.g., labels or business metrics) and run Lakehouse Monitoring on the resulting table to produce data and model quality metrics.

Simplification

- Lakehouse Monitoring provides a single monitoring tool and experience, regardless of whether monitoring data or AI. At present this is the only tool that supports a common API for monitoring both models and data.

Customization

- Users can introduce custom metrics based on specific business needs, further enhancing monitoring capabilities. Custom metrics can include aggregates or drift metrics that adhere to specific business logic.
- Additionally, it is possible to examine model performance on given data slices. This is often crucial to catch errors in subpopulations of interest, or when evaluating bias/fairness.

Dashboards and alerts

- Lakehouse Monitoring automatically generates a Databricks SQL dashboard for visualizing computed monitoring metrics. Additionally, this generated dashboard has user-editable parameters for both the whole dashboard and individual charts, allowing users to customize the date range, slicing logic, model versions, etc. Users can also add their own charts to the dashboard that join the monitoring metrics with external business data.
- Alerts can be defined against metrics in the generated metrics tables. A natural evolution to this is to set up alerts when quality or performance indicators deviate from expectations. This can be achieved using Databricks SQL alerts operating on a defined Databricks SQL query. The definition of this alerting criteria, and frequency of evaluation, will subsequently become part of the code deployed to the production environment.

CHAPTER 4

Design Decisions

Before presenting our updated reference architecture, let's highlight a number of design considerations you should take into account when first architecting an MLOps solution on Databricks.

Unity Catalog

ORGANIZING DATA AND AI ASSETS

One architectural decision that extends beyond a single MLOps solution is defining how both data and AI assets are organized within Unity Catalog. Below are a few factors motivating why thoughtful organization of AI assets within Unity Catalog is essential.

Efficiency and accessibility

- With a growing number of AI assets, organizations can face a situation where data and models are spread across different platforms and storage systems. This dispersion can make it difficult to find, manage and govern AI assets.
- A well-structured organization of data and AI assets within Unity Catalog ensures that data scientists, ML engineers and other stakeholders can easily locate and access the resources they need. This reduces the time spent searching for specific models or tables, and allows more time for actual use case development.

Scalability

- As an organization's data needs grow, so too does the number of AI assets. A well-architected Unity Catalog structure helps manage this growth, ensuring that the number of use cases can scale smoothly without becoming unmanageable.
- Having a well-defined structure to store AI assets further facilitates the addition of new data and ML models without disrupting existing use cases in production
- Further, a well-defined MLOps workflow enforces a standardized approach to deploying AI assets to production using Unity Catalog. This facilitates scalability to many use cases within an organization.

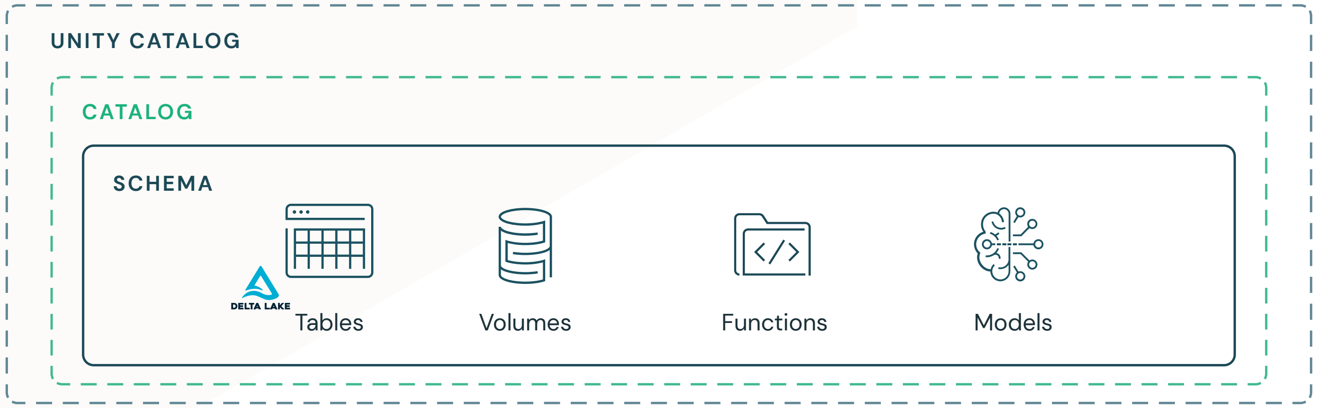
Governance

- By categorizing and structuring data and AI assets in a well-defined manner, organizations can implement effective access controls using Unity Catalog, ensuring that only authorized individuals can access certain data or models. In many cases this is required for compliance with data protection regulations, and for maintaining data privacy and security.
- For the purposes of auditability, Unity Catalog captures a log of actions performed against the metastore, and these logs are delivered as part of Databricks audit logs.

Collaboration

- In many organizations, multiple teams or individuals will be working with the same AI assets — sharing features and models across different use cases
- Consolidating both data and AI assets into a centralized location, and clearly delineating where specific assets are located and how they should be used, facilitates collaboration

CONCEPTS



Unity Catalog has a number of core concepts that we should define before unpacking the considerations when organizing AI assets along with data in Unity Catalog.

Catalog

Akin to a database. It serves as a container for all your data and AI assets, such as tables and models. A catalog can be thought of as a namespace for these data and AI assets. Later, in our proposed architecture, we will illustrate separate “dev,” “staging” and “prod” catalogs. These catalogs will contain data and AI models corresponding to the environment in which they were produced.

Schema

A logical construct within a catalog that groups related tables, views and models together. Put simply, it’s a way to organize related assets within a catalog.

Data tables

Data tables are the leaf level assets in Unity Catalog and can be referenced using a three-level name provided in the form `<catalog>.<schema>.<table>` . Any Delta table that has a primary key (and optionally a time series key) can be used as a source of ML features which can be joined with training samples to train an ML model.

Volume

Represents a logical volume of storage in a cloud object storage location. Volumes provide capabilities for accessing, storing, governing and organizing files. While tables provide governance over tabular data sets, volumes add governance over non-tabular data sets. You can use volumes to store and access files in any format, including structured, semi-structured and unstructured data. A volume will reside within a schema, under `<catalog>.<schema>.<volume-name>`.

Functions

SQL and Python UDFs are a part of Unity Catalog and can be accessed using a three-level name provided in the form `<catalog>.<schema>.<function>`. Functions enable the ability to use Python UDFs to compute on-demand features as inputs for machine learning, and can be used in model training, batch inference and real-time inference. Functions are appropriate in cases where fresh features (e.g., computing a user's distance to a restaurant) need to be computed at inference time, or where features from data sources cannot be pre-materialized to a data lake for regulatory or security reasons (e.g., a user's credit score).

Registered model

An MLflow model that has been registered to Unity Catalog. The registered model has a unique name, versions, model lineage and other metadata. When registering a model to Unity Catalog, a three-level name is provided in the form `<catalog>.<schema>.<model>`.

Model version

A version of a registered model. When a new model is added to the Model Registry, it is added as Version 1. Each model registered to the same model name increments the version number. A specific model version can be loaded using the model URI `models:/<catalog>.<schema>.<model>/<model_version>`.

Model alias

A mutable, named reference to a particular version of a registered model. Typical uses of aliases are to specify which model version should be deployed in a model deployment pipeline, or to write inference workloads that target a specific alias. Aliases are flexible, and as such can be tailored to suit your use case or organization requirements. For example, you could assign the model version that should serve the majority of production traffic the "Champion" alias. Inference workloads could then target that alias using the model URI `models:/<catalog>.<schema>.<model>@Champion`.

CONSIDERATIONS

When designing how to organize your data and AI assets in Unity Catalog, it's worth considering how to best leverage the three-level namespace of catalog, schema and entity name. For example:

1. Do you equate each schema to a different business unit or team, under which tables and models specific to that business unit or team reside?
2. Do you have schemas within a catalog mapped to a medallion architecture of Bronze, Silver and Gold tables? If so, where do feature tables and ML models then reside?

While there is no one-size-fits-all approach to managing AI assets with data in Unity Catalog, there are a number of guiding principles that can be applied when contending with these decisions:

- **Team size**
  The size of a team can influence the level of detail in the organizational structure of data and AI assets in Unity Catalog. Larger teams might require more granular categorizations (e.g., specifying a schema per business unit or team), while smaller teams might function well with a simpler structure (e.g., all teams operating across shared schemas — Bronze, Silver, Gold — within each catalog).

- **Complexity of projects**
  More complex projects, involving the registration of many different models, may necessitate dedicated schemas per project. It should be noted, however, that this requirement can be handled in some cases through the creation of a custom MLflow PyFunc model. This approach can be used to create a wrapper around many models using a single PyFunc model, which can then be registered in the same fashion as any other individual model.

- **Access levels and permissions**
  Consideration should be given to the different access levels and permissions that will be required. Some assets might need to be accessible to everyone on a given team, while others should be restricted to specific individuals or roles. With Unity Catalog, both data and AI assets can be grouped and governed at the environment, project or team level.

- **Models, functions and features in Unity Catalog**
  With Models in Unity Catalog, AI artifacts can be governed in the same way as data in the Lakehouse. This unified governance enables the sharing of models across Databricks workspaces, and teams within an organization. As such, where models, functions and features reside within a given catalog is often determined by organizational requirements around sharing of these assets across teams and use cases. From a catalog perspective, Models in Unity Catalog should be treated no differently than data in Unity Catalog — models are registered to a catalog indicating the environment in which they were produced. Aliases can then provide the ability to indicate which models are deployed in which contexts (e.g., a "Champion" model serves the majority of production traffic, while the "Challenger" model serves a small fraction for testing).

- **Discoverability**
  A standardized approach across teams in how assets are organized within Unity Catalog is important for setting up ACLs and discoverability. Notably, there are also methods like tags to aid discoverability.

RECOMMENDED ORGANIZATION

While there is no universal blueprint for organizing data and AI assets in Unity Catalog, the structure we propose below serves as a robust starting point. It's designed with flexibility in mind, providing a clear demarcation between assets created in different environments while also enabling simplified discoverability, sharing, and governance of data and AI assets across teams. Notably, we replicate the structure across dev, staging and prod catalogs. This enables users to develop and test against catalog structures — dev and staging, respectively — mirroring that of the production environment. For this example, we show all ML assets for a specific "fraud detection" use case organized under a single "fraud_detection" schema.

dev

```
bronze
     > Tables:
    silver
     > Tables:
    gold
     > Tables:
       user_features
       product_features
    fraud_detection
     > Tables:
       offline_location_features
       fraud_clf_inference
       fraud_clf_metrics
     > Volumes:
     > Models:
       fraud_clf
     > Functions:
       compute_distance
```

staging

```
bronze
     > Tables:
    silver
     > Tables:
    gold
     > Tables:
       user_features
       product_features
    fraud_detection
     > Tables:
       fraud_clf_inference
       fraud_clf_metrics
     > Volumes:
     > Models:
       fraud_clf
     > Functions:
       compute_distance
```

prod

```
bronze
     > Tables:
    silver
     > Tables:
    gold
     > Tables:
       user_features
       product_features
    fraud_detection
     > Tables:
       fraud_clf_inference
       fraud_clf_metrics
      > Volumes:
      > Models:
        fraud_clf
      > Functions:
        compute_distance
```

Catalog level

At the catalog level, assets are segregated based on the environment from which they were produced. As a result, teams have an inherent understanding of the maturity and readiness of each asset.

dev catalog

- Assets under active development and testing. We illustrate additional tables and models within this catalog, indicative of new assets in development.
- Relatively open access, enabling users to easily read and write assets during the development process

staging catalog

- Used as an area for writing assets produced during the testing of code in the staging environment prior to deployment to the production environment
- Additionally, this catalog could also be used to store more permanent preproduction assets for the purposes of mirroring the production environment during integration testing
- Data and models stored in the staging catalog may be temporary, or cleaned up on some periodic basis
- Limited write access aside from that of administrators and service principals
- Read access should be enabled for users who may need to debug integration tests

prod catalog

- Assets that have been produced using production-deployed code
- We can inherently assert that these assets have been produced by code that has been tested prior to deployment to the production environment
- Access to write to the prod catalog is typically limited to a small number of administrators or service principals
- Access to read from the prod catalog can be granted to users in non-production Databricks workspaces

Schema level

Within each catalog, assets are further categorized into schemas.

1. `bronze`: Typically raw data, which is unaltered and as is from the source
2. `silver`: Cleaned or processed data, often transformed from raw data in Bronze
3. `gold`: Further enriched or aggregated data, ready for analysis or model training

   TABLES

   Feature tables:

   - Tables with a primary key used to train models and serve features
   - Here we illustrate how generic feature tables such as user and product features might appear under the "Gold" schema, given that such features could be shared across multiple use cases

4. `fraud_detection`: Consolidated schema for all machine learning assets relevant to the fraud detection use case, including (but not limited to):

   MODELS

   - Each registered model typically addresses some aspect of the schema-specific use case or project (in some cases there may be multiple models per use case)
   - In our example we illustrate a single registered model for the fraud detection use case, called "fraud_clf" (fraud classifier)
   - Every time a new MLflow model is registered to a given model, it will produce a new incremental model version
   - Aliases are then assigned to specific model versions to mark them for deployment and to manage rollouts (e.g., "Champion," "Challenger")

   VOLUMES

   - Unstructured data (e.g., images, text) used to train models

TABLES

Feature tables:

- Feature tables created specifically for the fraud detection use case
- In the above example, we show a table called "offline_location_features" in development under the dev catalog. This table contains historic location-based features specific to the fraud detection use case.

Inference tables:

- Tables containing request-response data generated by Model Serving

Metric tables:

- Monitoring tables generated by Lakehouse Monitoring

FUNCTIONS

- Python UDFs used by models to compute features on demand at inference time
- In our example we illustrate a "compute_distance" function which computes real-time location-based features at inference time
- Note that functions can be used in many scenarios, and are not confined to just feature computation for models

For data engineering, we present a medallion architecture mapping to `bronze`, `silver` and `gold` schemas. This structure provides a step-by-step data transformation process, ensuring quality and consistency. While this naming convention is useful for example purposes, conceptual equivalents are often termed differently between organizations.

Model Serving

Incorporating real-time models into existing MLOps workflows necessitates a new set of considerations one must take into account. Collecting use case requirements such as service level agreements (SLAs) from your business stakeholders up front is essential in order to appropriately design a workflow that sufficiently tests a real-time model prior to deployment. There are two primary ways in which an MLOps workflow deploying a real-time model diverges from a MLOps workflow deploying batch inference pipelines:

1. **Pre-deployment testing**
   Ensuring system performance. A unique requirement for deploying real-time models is ensuring that model serving infrastructure is tested, in addition to regular pre-deployment testing (e.g., integration tests).

2. **Real-time model deployment**
   Ensuring model accuracy (or other ML performance metrics). Unlike batch inference pipelines, real-time models often require a paradigm of online model evaluation to ensure that the most accurate model is always serving predictions. This may involve updating a model on a more regular basis based on newly arriving data. As a result, this necessitates a different approach to deployment.

PRE-DEPLOYMENT TESTING

On top of the standard unit and integration tests that one would employ in a typical MLOps workflow, performance testing model serving infrastructure prior to deploying a real-time model is required in order to test how the model and serving infrastructure handles request traffic. Such testing could include:

- **Deployment readiness checks**
  These checks are conducted prior to creating or updating a Model Serving endpoint, to validate that configuration scripts are correctly specified, required dependencies are present, and the correct input data structure is defined, etc.

- **Load testing**
  Load testing involves conducting a comprehensive assessment of the real-time system's performance, stability and responsiveness under varying degrees of demand. The following are all components of load testing:

LATENCY

Ensure the model's inference and overall system response times meet predefined SLAs. Metrics like median latency and long tail (95th or 99th percentile) can capture both typical and worst-case response scenarios.

THROUGHPUT

Measure the application's capacity to handle queries over time, typically gauged in queries per second (QPS). Evaluations should consider varying load conditions to discern how performance fluctuates with demand shifts.

STANDARD LOAD EVALUATION

Analyze system behavior under expected request volumes, scaling from regular to anticipated peak levels. This not only gauges throughput but also examines metrics like response time and error rate.

STRESS ASSESSMENT

Deliberately overwhelm the system to observe its response to abnormally high demands, looking for graceful failure and effective recovery.

Note: The deployment patterns are not mutually exclusive. Organizations can, and often do, combine multiple patterns to suit requirements and risk profiles. For instance, one could start with a shadow deployment to evaluate a new model version without impacting the user experience, and based on findings, proceed with a gradual rollout.

REAL-TIME MODEL DEPLOYMENT

Given a newly trained ("Challenger") model and an existing ("Champion") model in production, there are a variety of approaches that can be taken for real-time deployments. Some common deployment patterns include:

- **A/B testing**
  Deploy multiple model versions concurrently and distribute the traffic among them to test and evaluate their performance. Based on predefined success criteria, such as accuracy or conversion rates, the best-performing model is then selected to handle all traffic.

  RELATION TO OTHER PATTERNS: A/B testing usually involves splitting traffic evenly or in some predefined ratio between model versions. Unlike gradual rollouts, where traffic is incrementally shifted based on performance metrics, A/B testing maintains its traffic splits until a decision is made.

- **Gradual rollout**
  This pattern begins by exposing a new model version to a small selected segment of request traffic (sometimes referred to as a canary deployment). Performance is closely monitored. If the new version meets defined success criteria, traffic to this model version is gradually increased. This allows for the benefits of continuous exposure to real traffic, while still having the safety net of scaling back if anomalies arise.

  RELATION TO OTHER PATTERNS: Unlike A/B testing, where traffic is set to predefined ratios, gradual rollout adjusts traffic based on model performance metrics. This is a more adaptive and responsive approach compared to other patterns.

- **Shadow deployment**
  In this pattern, a new model version runs alongside the existing version, but it does not actively serve traffic. Instead, the new version receives a copy of the incoming traffic and generates predictions, which can be compared to the current version for evaluation purposes without affecting the user experience.

  RELATION TO OTHER PATTERNS: Akin to a silent observer, where you can evaluate a new model version without affecting end users. This offers a risk-free comparison unlike a gradual rollout. Note that this involves performing additional work as multiple predictions are generated for the same input.

Implementing in Databricks

- **MODEL ALIASES**
  When implementing strategies like A/B testing or canary deployments, use model aliases to identify and manage different model versions. For example, using aliases you can easily switch traffic between a “Champion” model version and “Challenger” model version without needing extensive reconfigurations.

- **CONTROL ENDPOINT TRAFFIC**
  Databricks Model Serving provides the functionality to create a single Model Serving endpoint with two models and split endpoint traffic between those models. For instance, during A/B testing, you can set specific traffic percentages for each model version. Similarly, in rolling deployments, adjust these percentages as you phase in the new model.

- **LAKEHOUSE MONITORING WITH MODEL SERVING**
  With Lakehouse Monitoring you can set up monitoring on automatically captured inference tables from Model Serving endpoints. As such we can capture performance metrics of different model versions exposed to traffic. This can be particularly useful for any of the deployment patterns outlined above, where you need to closely monitor and compare predictions between different model versions.

While each deployment pattern outlined above has its strengths, the pattern you choose is ultimately dependent on the requirements of the specific use case and organization. With Databricks Model Serving, such patterns can be implemented easily, streamlining the deployment process for real-time ML models.

CHAPTER 5

Reference Architecture

Note: While the model promotion logic presented here is reliant on use of "Champion" and "Challenger" aliases, this naming convention is entirely adaptable. The flexibility of model aliases in Unity Catalog enables you to attribute aliases specific to your team's needs and terminologies.

In the reference architecture presented, we follow a deploy code workflow and assume a 1:1 mapping between environments and Unity Catalog. Hence, there are dev, staging and prod catalogs corresponding to assets produced in the development, staging and production environments, respectively. In this context an environment is the equivalent of a Databricks workspace.

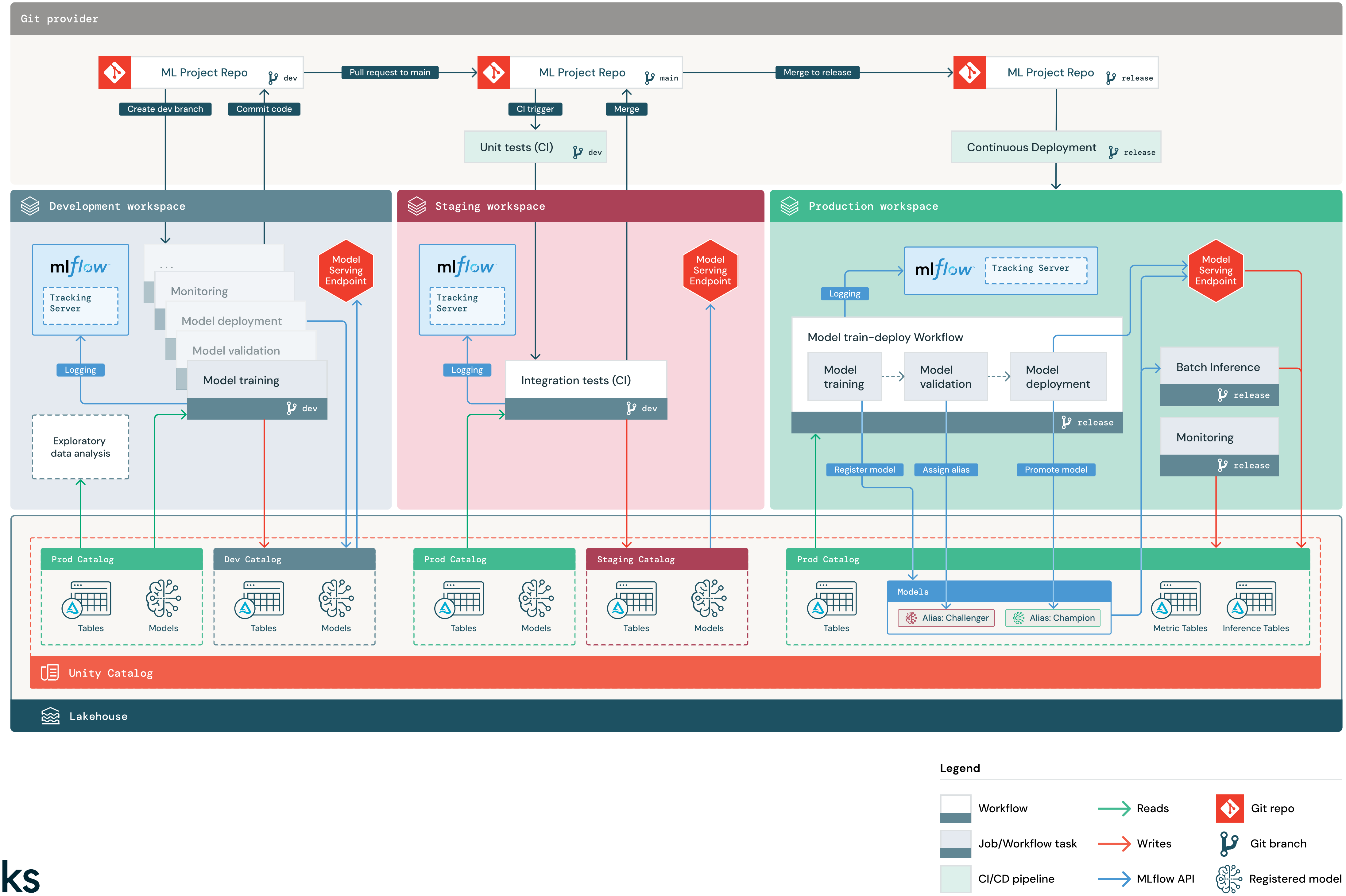
Within each catalog we have the ability to manage both data and models. This architecture allows assets in the prod catalog to be accessed from the development environment, provided appropriate permissions are granted. Typically, this would involve granting read-only access to the prod catalog from the development environment, although it's important to note that not all organizations may allow this level of accessibility. Given this access, data scientists can develop ML code using production data in the development environment, where write access is restricted to only the dev catalog. Additionally, data scientists working within the development environment can load current production models residing in the prod catalog to compare against newly developed models. While our workflow involves registering and managing models within catalogs in Unity Catalog, each Databricks workspace has its own dedicated MLflow Tracking server to which metrics, parameters and model artifacts are logged.

Note that environment terminology may vary across organizations and even teams. We present an architecture with development, staging and production environments; however, conceptually similar environments may have different naming conventions within your organization (e.g., QA/testing/pre-prod environment may be equivalent to staging).

Furthermore, just as environments can have different names, the branch naming and management strategies in Git can also vary. The Git workflow we present involves a dev branch merged into "main," and subsequently the main branch is merged into a "release" branch. However, in some cases teams may prefer working on "feature" branches corresponding to specific features or tasks in places of the dev branch. Similarly, the creation of a release branch might not always be necessary, with teams using tags to mark a particular commit as a release instead.

Thus, both environment names and CI/CD workflow depicted are intended to be used as guidance rather than to be prescriptive. The architecture and workflow outlined should be adapted to fit the unique needs and circumstances of your team and project.

Multi-environment view

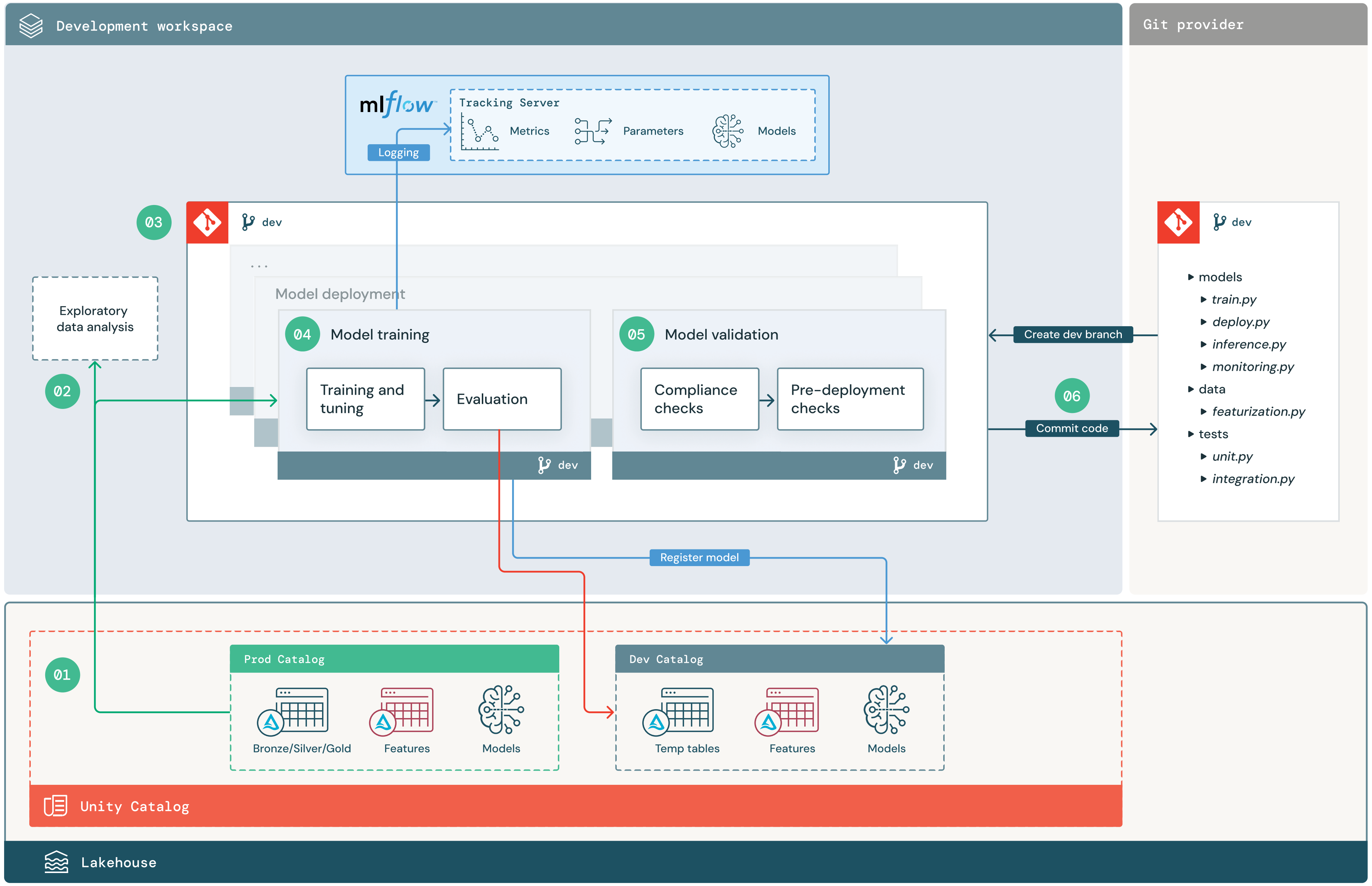


In the following sections we provide a detailed explanation of the precise steps in which code is moved across the three environments illustrated above. At a high level, we have the following steps:

1. **Development:** ML code is developed in the development environment, with code pushed to a dev (or feature) branch.
2. **Testing:** Upon making a pull request from the dev branch to the main branch, a CI trigger runs unit tests on the CI runner and integration tests in the staging environment.
3. **Merge code:** After successfully passing these tests, changes are merged from the dev branch to the main branch.
4. **Release code:** The release branch is cut from the main branch, and doing so deploys the project ML pipelines to the production environment.
5. **Model training and validation:** The model training pipeline ingests data from the prod catalog. Upon validating, the resulting model artifact is registered to the prod catalog. A "Challenger" alias is attached to the newly registered model version.
6. **Model deployment:** A model deployment pipeline evaluates the current "Champion" model versus "Challenger" model, with the best-performing model version taking the "Champion" alias after this evaluation.
7. **Model inference:** Model Serving or other inference pipelines load the "Champion" model to compute predictions. Predictions are logged to inference tables, which can be used to monitor the "Champion" model's performance.
8. **Monitoring:** Scheduled or continuous pipeline to refresh Lakehouse Monitoring metric tables. Inference tables are monitored to detect data or model drift. Databricks SQL dashboards are automatically created to display monitor metrics.

Development

Expanding on the above, we start by looking in detail at the development environment.



Legend

▭	Workflow	→	Reads	◆	Git repo
▨	Job/Workflow task	→	Writes	⑂	Git branch
□	Pipeline setup	→	MLflow API	❂	Registered model

Data

To support their development activities, data scientists will have read-write access to the dev catalog. This catalog is where temporary data and feature tables are written to from the development workspace. Additionally, this dev catalog is used to register any models created during code development to Unity Catalog.

Ideally, data scientists working within the development workspace will possess read-only access to production data in the prod catalog. This facilitates their ability to read production tables, as well as load models that have been registered to the prod catalog. In cases where it is not possible to grant read-only access to the prod catalog, a snapshot of production data may be written to the dev catalog to enable data scientists to develop and evaluate their project code.

Note that read access to inference and metric tables in the prod catalog will enable data scientists to examine current production model predictions and Lakehouse monitoring metrics.

Exploratory data analysis (EDA)

The data scientist's work typically begins with Exploratory Data Analysis (EDA) in the development environment. This is an interactive, iterative process — typically conducted in notebooks — to assess whether the available data has the potential to solve the business problem at hand. EDA is also where the data scientist will begin identifying data preparation and featurization steps for later model training. This ad hoc process is generally not part of a pipeline that will be deployed in other execution environments.

Within Databricks this EDA process can be accelerated with Databricks AutoML. AutoML not only generates baseline models given a data set, but also provides the underlying model training code in the form of a Python notebook. Notably for EDA, AutoML calculates summary statistics on the provided data set, creating a notebook for the data scientist to review and adapt.

Project code

This code repository contains all the pipelines, modules and ancillary project files involved in the ML solution. Development ("dev") branches are used to develop changes to existing pipelines or create new ones. Even during EDA and initial phases of a project, data scientists should develop within a repository to help with tracking changes and sharing code.

Model training development

Data scientists develop the model training pipeline in the development environment using Lakehouse tables from the dev or prod catalogs.

- **TRAINING AND TUNING**

  The training process logs model parameters, metrics and artifacts to the MLflow Tracking server. After training and tuning hyperparameters, the final model artifact is logged to the tracking server to record a robust link between the model, its input data and the code used to generate it.

- **EVALUATION**

  Model quality is evaluated by testing on held-out data. The results of these tests are logged to the MLflow Tracking server. At this point it can be determined if a newly developed model outperforms that of the current model in production. Given sufficient permissions, any production model registered to the prod catalog can be loaded into the development workspace and compared against a newly trained model.

  If governance requires additional metrics or supplemental documentation about the model, this is the time to add that functionality to the code using MLflow Tracking. Model interpretations (e.g., plots produced by SHAP) and plain text descriptions are common, but defining the specifics for such governance requires input from business stakeholders or a data governance officer.

MODEL TRAINING OUTPUT

The model training pipeline produces an ML model artifact stored in the MLflow Tracking server. At this point the model artifact is tracked to the development environment MLflow Tracking server. However, when this pipeline is executed in either staging or production workspaces, the model is tracked to the respective MLflow Tracking servers of these workspaces.

Upon model training completion, the model is registered to Unity Catalog. When executed in the development environment, this model is registered to the dev catalog. Pipeline code is typically parameterized in such a manner that the model will be registered to the catalog corresponding to the environment the model training pipeline is executed in.

In the proposed architecture, we deploy a multitask Databricks Workflow in which the first task is the model training pipeline, followed by subsequent model validation, then model deployment tasks. As such, the model training task will yield a model URI (or path) which can be used by the subsequent model validation task. This model URI value can be passed into subsequent tasks using the taskValues subutility in Databricks Utilities.

Model validation and deployment development

In addition to the model training pipeline, other ancillary pipelines such as model validation and model deployment pipelines are developed in the development environment.

MODEL VALIDATION

This pipeline will take the model URI from the previous model training pipeline, load the model from Unity Catalog and apply validation checks.

The scope of validation checks on a newly trained model are typically context dependent. These checks can range from fundamental checks like asserting the model artifact's format and verifying the presence of required metadata for subsequent deployment and inference pipelines, to more complex checks, especially in highly regulated industries. The latter may involve predefined compliance checks and asserting that the model performance meets a certain threshold on selected data slices.

The primary function of this model validation pipeline is to determine whether a model should proceed to the deployment step of our workflow. If the model passes pre-deployment checks, we attach a “Challenger” alias to the registered model in Unity Catalog. Conversely, if these checks fail, we exit the process, and using Workflows can alert users about the task failure.

- **MODEL DEPLOYMENT**

  The model deployment pipeline typically either directly promotes the newly trained “Challenger” model to “Champion” status using an alias update, or facilitates a comparison between the existing “Champion” model and the new “Challenger” model. In addition to this, this pipeline might also be responsible for setting up any required inference infrastructure, such as Model Serving endpoints. We save a detailed discussion of the steps involved in the model deployment pipeline for the “Production” section below.

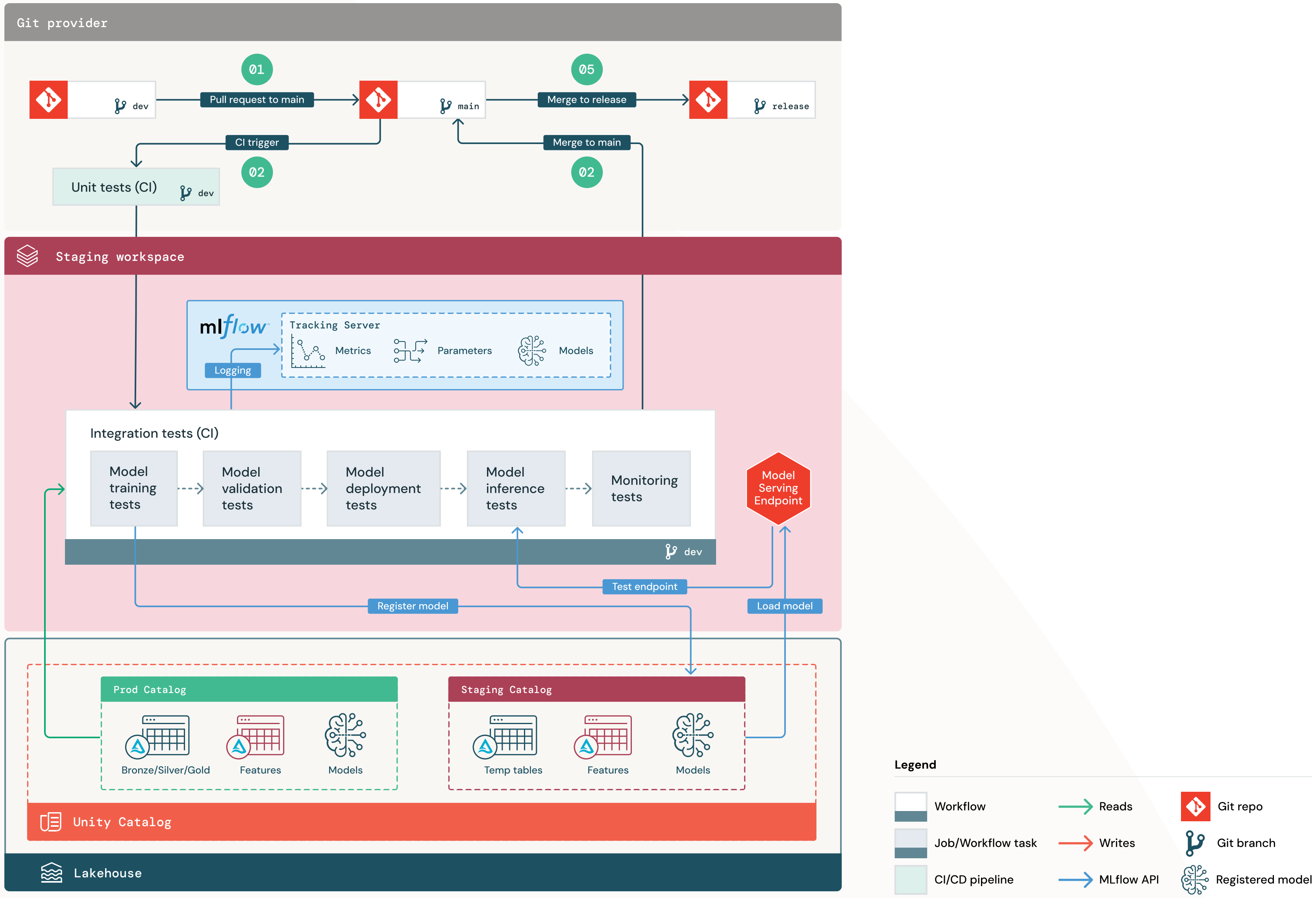
Commit code

After developing code for training, validation, deployment and other ancillary pipelines, the data scientist or ML engineer commits the dev branch changes into source control.

This development section does not discuss Model Serving, inference or monitoring pipelines in detail; see the “Production” section below for more information.

Staging

One of the core motivations for the deploy code approach presented in this architecture is that all code can be robustly tested prior to deployment in the production environment. The transition of code from development to production occurs via the staging environment. In the staging environment, all pipelines intended for production deployment are rigorously tested. While both data scientists and ML engineers share responsibility for writing tests for code and models, ML engineers typically manage the continuous integration pipelines and orchestration within a project.



Data

The staging environment should have its own catalog in Unity Catalog (the "staging" catalog shown in the figure above) for testing ML pipelines and registering models to Unity Catalog. Assets written to this catalog are generally temporary and only retained long enough to run tests and to investigate test failures. The staging catalog can be made readable from the development environment for debugging.

Merge code

Data scientists develop the model training pipeline in the development environment using Lakehouse tables from the dev or prod catalogs.

- **PULL REQUEST**

  The deployment process begins when a pull request is created against the main branch of the project in source control.

- **UNIT TESTS (CI)**

  The pull request automatically builds source code and triggers unit tests. These unit tests will run on the runner of the continuous integration platform being used. If unit tests fail, the pull request is rejected.

  Note that unit tests are part of the software development process and are continuously executed and added to the codebase during the development of any code. Running unit tests as part of a continuous integration pipeline ensures that any changes or additions from development branches do not inadvertently break existing functionality.

Integration tests (CI)

Upon passing the unit tests, the pull request undergoes integration tests. These tests run all pipelines (in a limited capacity) to confirm that they function correctly together. Pipelines often share common code functionality, which is why it can be important to test pipelines collectively. Integration tests are executed in the staging environment, which should mimic the production environment as much as is reasonable.

Additionally, if deploying an ML application with real-time inference, Model Serving infrastructure should be created and tested in the staging environment. This involves triggering the model deployment pipeline, which creates a Model Serving endpoint in the staging environment, loads a test model (e.g., a model trained on a limited subset of data). To test the endpoint, some of the testing approaches mentioned in the Pre-deployment testing section could be employed.

Integration tests can trade off fidelity of testing for speed and cost. For example, when models are expensive to train, it is common to test the model training pipeline on small data sets or for fewer iterations to reduce cost. When models are deployed behind REST APIs, some high-SLA models may warrant full-scale load testing within these integration tests, whereas other models may be tested with small batch jobs or a few requests to a temporary Model Serving endpoint.

Merge

If all tests pass, the new code is merged into the main branch of the project. If tests fail, the CI/CD system should notify users and post results on the pull request.

Note: It can be useful to schedule periodic integration tests on the main branch, especially if the branch is updated frequently with concurrent pull requests.

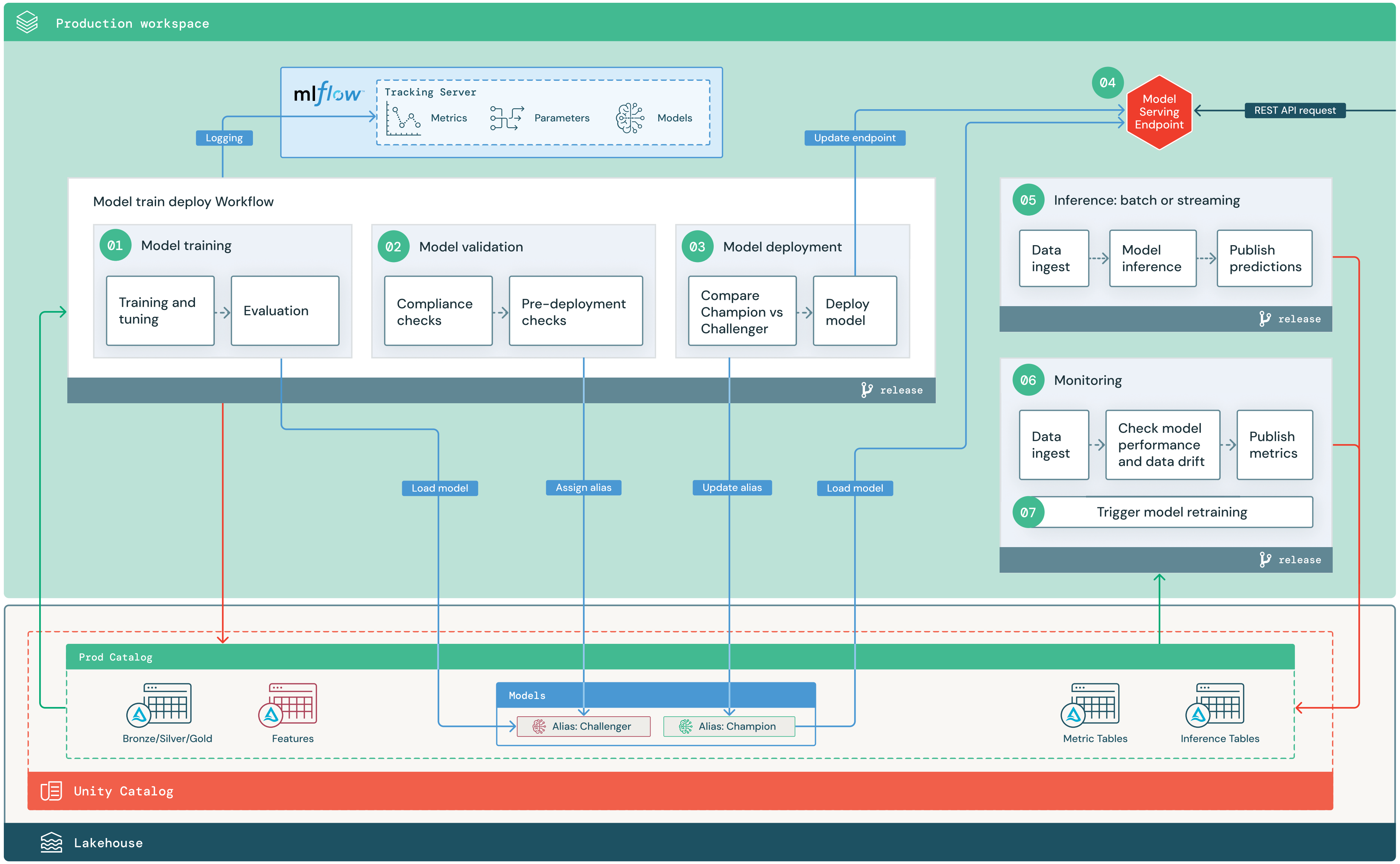
Cut release branch

Once CI tests have passed and the dev branch merged into the main branch, ML engineers can cut a release branch from that commit.

Production

The production environment is where ML pipelines are operationalized and start directly serving the business or application. Managed by select ML engineers and admins, this environment is where the pipelines of the ML project are deployed and executed. These pipelines trigger model training, validate and deploy new model versions, publish predictions to downstream tables or applications, and monitor the entire process to avoid performance degradation and instability. While we illustrate batch and streaming inference alongside real-time model serving below, it's worth noting that most ML applications typically only use one of these methods, based on business needs.

Data scientists usually do not have write or compute access in the production environment. However, it is important to provide them with visibility to test results, logs, model artifacts and the status of ML pipelines in production. This visibility allows them to identify and diagnose problems in production. Data scientists working in the development environment can be granted read access to model artifacts and monitoring tables in the prod catalog. Furthermore, this allows data scientists to load registered models from the prod catalog to compare against models in development.



Legend

⬜	Workflow	→	Reads	◆	Git repo
▤	Job/Workflow task	→	Writes	⑂	Git branch
🟩	CI/CD pipeline	→	MLflow API	⚙	Registered model

Model training

In the architecture illustrated above we deploy a Databricks Workflow consisting of three tasks: model training, model validation and model deployment. Once deployed, this workflow runs either when code changes affect upstream featurization or training logic, or when automated retraining is scheduled or triggered. The first task in this workflow, the model training task, loads tables and/or feature tables from the prod catalog and performs the following steps:

- **TRAINING AND TUNING**

  During the training process, logs are recorded to the production environment MLflow Tracking server. These include model metrics, parameters, tags and the model itself. If using feature tables, the model will be logged to MLflow using the Databricks Feature Store client. This will result in the logged model being packaged with feature lookup information, which can be used at inference time.

  In the development environment, data scientists may test many algorithms and hyperparameters, but it is common to restrict those choices to the top-performing options in the production training code. Restricting tuning can reduce the variance from tuning during automated retraining, and expedite the training and tuning process.

  Alternatively, the optimal set of hyperparameters for a model may be determined in the development environment if read-only access to the prod catalog is available. The model training pipeline deployed in production can then be executed using this selected set of hyperparameters using a configuration file passed into the pipeline.

Note: We describe a fully automated approach here, consolidating the model validation task and subsequent model deployment task into a single Databricks Workflow. In some scenarios compliance checks require human expertise, where human reviewers evaluate computed statistics or visualizations from the model validation pipeline.

In such cases the model validation pipeline and model deployment pipeline can be separated into different Databricks Workflows. Upon successful completion of the model validation task, users are notified and can then review the pipeline's output. Once a team member has approved the model, a separate model deployment workflow can be manually triggered to deploy the approved model.

Note that model aliases can be useful in these scenarios to denote which models are currently deployed, offering further flexibility to the model validation and deployment process.

- **EVALUATION**

  Model quality is evaluated by testing on held-out production data. The results of these tests are logged to the MLflow Tracking server. During development, data scientists select meaningful evaluation metrics for the use case, and those metrics or their custom logic are used in this step.

- **REGISTER MODEL**

  Upon completion of model training, the model artifact is registered to the prod catalog. The model appears as a newly registered model version under the model path in Unity Catalog. Once the model training pipeline successfully runs, the model URI of the newly registered model in Unity Catalog is yielded as a task value, enabling its use by subsequent tasks in the workflow.

Model validation

As outlined in the "Development" section above, the model validation pipeline uses the model URI from the preceding model training pipeline, and loads the model from Unity Catalog. The model artifact then undergoes a series of validation checks, adjusted to fit the specific context of the use case. These checks can encompass everything from basic format and metadata validations through to performance evaluations (e.g., performance on selected data slices) and compliance checks (for tags, documentation, etc.).

If the model successfully passes all validation checks, the "Challenger" alias is assigned to the model version in Unity Catalog. In the event that the newly trained model does not pass all validation checks, the process will exit and users can be notified on failure of the task to investigate further. Tags can be used to add key-value attributes to the model version depending on the outcome of these validation checks. For example, adding a tag like "model_validation_status": "PENDING", and updating the value to "PASSED/FAILED" following execution of the model validation pipeline.

Note that since the model is registered to Unity Catalog, data scientists working in the development environment can load this model version from the prod catalog to investigate further in the event of model validation failure. Regardless of the outcome, results are recorded to the registered model in the prod catalog through annotations to the model version.

Model deployment

The model deployment pipeline is executed upon completion of the model validation pipeline. At this point, the newly registered model having passed all validation checks in the preceding step will have been assigned the "Challenger" alias.

Like the model validation pipeline, the functionality of the model deployment pipeline can greatly vary depending on the context of the use case. We outline an approach where a "Champion" model is already in use in production. To prevent performance degradation, the newly trained "Challenger" model must be compared against the "Champion" model it aims to replace.

- COMPARE "CHALLENGER" VS. "CHAMPION"

  Comparing a new "Challenger" model versus an existing "Champion" model can be done in either an offline or online manner. An offline comparison would evaluate both models against a held-out data set, with results tracked to the MLflow Tracking server.

  In cases involving real-time models, it is often necessary to perform longer running online comparisons, such as A/B tests, or gradual rollouts. In the case of a gradual rollout, for example, the deployment process is inherently iterative. There will be multiple evaluations and traffic adjustments as the model version is gradually rolled out to all the traffic. Once the model version is exposed to full production traffic, the "Champion" alias will be assigned to the model version.

  As alluded to in the real-time model deployment design decision section, Model Serving enables you to automatically collect inference tables containing endpoint request-response data, and monitor these tables with Lakehouse Monitoring. This combined functionality enables data scientists to actively monitor performance of a new model version before exposing it to all live traffic. Alerts can be additionally configured to notify users when certain performance thresholds are achieved.

Depending on the outcome of the model comparison, either the "Challenger" model version will have its alias updated to "Champion," or the existing "Champion" model will retain its alias.

In the event of the first deployment, where there is no existing "Champion" model, the "Challenger" model should be compared to a business heuristic or other threshold as a baseline.

Although we describe a fully automated approach here, additional manual approval steps can be incorporated if required, facilitated through workflow notifications or CI/CD callbacks from the model deployment pipeline.

DEPLOY MODEL

For use cases involving batch or streaming inference, simply promoting a model version after validation and subsequent comparison checks to the "Champion" alias is sufficient at this point. The downstream batch or streaming inference pipeline will then pick up the model according to the "Champion" alias, and use this model to compute predictions.

Real-time use cases necessitate an additional step to set up the infrastructure needed to expose the model as a REST API endpoint. Databricks greatly simplifies this step through the ability to create and manage a Model Serving endpoint.

In our proposed architecture, we perform the comparison between "Challenger" and "Champion" models, and following this, update an existing Model Serving endpoint to use the model version of the "Champion" model. Thus, the "deploy model" step in our model deployment pipeline would involve determining the model version of the "Champion" model and updating the Model Serving endpoint if the "Challenger" model has replaced the "Champion" model. When updating an endpoint, Databricks performs a zero-downtime update by keeping the existing endpoint configuration up until the new one becomes ready. Doing so reduces risk of interruption for endpoints that are in use.

In the event that there is not an existing endpoint, this step will involve the creation of a new Model Serving endpoint, configured to use the model version of the "Champion" model.

Model Serving

When configuring a Model Serving endpoint, the name of the model in Unity Catalog will be specified, along with the model version to serve — in this case the model version of the "Champion" model. If the model version was trained using features from Unity Catalog, the model stores the dependencies to features and functions used. Model Serving will automatically use this dependency graph to look up features from appropriate online stores at inference time. Additionally, this approach can be used to apply functions to perform preprocessing on data, or compute on-demand features before scoring the model.

Notably, it is possible to create a single endpoint with multiple models and specify the endpoint traffic split between those models. This functionality can be used to conduct the longer running online "Champion" versus "Challenger" comparison outlined above.

For monitoring endpoint health, it is also possible to combine Model Serving with external monitoring tools such as Prometheus or Datadog.

Inference: batch or streaming

The inference pipeline is responsible for reading the latest data in the prod catalog, executing functions to compute on-demand features, loading the "Champion" model, performing inference, and publishing predictions. For higher throughput, higher-latency use cases, batch or streaming inference is generally the most cost-effective option. Additionally, in scenarios where low-latency predictions are required, but predictions can be computed in an offline manner, these batch predictions can be published to an online key-value store such as DynamoDB or Cosmos DB.

For this pipeline we reference a registered model in Unity Catalog by its alias. As such, we specify the inference pipeline to load and apply the "Champion" model version for batch or streaming inference. If the "Champion" version is updated to reference a new model version, the inference workload automatically picks it up on its next execution. Thus, the model deployment step is decoupled from inference pipelines.

A batch job would likely publish predictions to tables in the prod catalog, over a JDBC connection, or to flat files. A streaming job would likely publish predictions either to Unity Catalog tables or to message queues like Apache Kafka®.

Lakehouse Monitoring

Lakehouse Monitoring monitors statistical properties (data drift, model performance, etc.) of input data and model predictions. These metrics are published for dashboards and alerts.

- **DATA INGESTION**
  This pipeline reads in logs from batch, streaming or online inference.
- **CHECK ACCURACY AND DATA DRIFT**
  The pipeline then computes metrics about the input data, the model's predictions and the infrastructure performance. Metrics that measure statistical properties are generally chosen by data scientists during development, whereas metrics for infrastructure are generally chosen by ML engineers. Note that custom metrics can be defined and monitored with Lakehouse Monitoring.
- **PUBLISH METRICS AND SET UP ALERTS**
  The pipeline writes to Lakehouse tables in the prod catalog for analysis and reporting. These tables should be readable into the development environment to allow data scientists to perform granular analysis if required. Tools such as Databricks SQL are used to produce monitoring dashboards, allowing for health checks and diagnostics. The monitoring job or the dashboarding tool issues notifications when health metrics surpass defined thresholds.
- **TRIGGER MODEL TRAINING**
  When the model monitoring metrics indicate performance issues, or when a model inevitably becomes out of date, the data scientist may need to return to the development environment and develop a new model version. SQL alerts can be used to notify data scientists when this happens.

Retraining

This architecture supports automatic retraining using the same model training pipeline above. While we recommend beginning with a simple schedule for periodic retraining, organizations can add triggered retraining when needed.

Note: While automated retraining is supported in this architecture, it isn't required, and caution must be taken in cases where it is implemented. It is inherently difficult to automate selecting the correct action to take from model monitoring alerts. For example, if data drift is observed, does it indicate that we should automatically retrain, or does it indicate that we should engineer additional features to encode some new signal in the data?

- **SCHEDULED**
  If fresh data are regularly made available, rerunning model training on a defined schedule can help models to keep up with changing trends and behavior.
- **TRIGGERED**
  If the monitoring pipeline can identify model performance issues and send alerts, it can additionally trigger retraining. For example, if the distribution of incoming data changes significantly or if the model performance degrades, automatic retraining and redeployment can boost model performance with minimal human intervention. This can be achieved through a SQL alert to check whether a metric is anomalous (e.g., check drift or model quality against a threshold). The alert can be configured to use a webhook destination, which can subsequently trigger the training workflow.

When the retraining pipeline or other ancillary pipelines themselves begin to exhibit performance issues, the data scientist may need to return to the development environment and resume experimentation to address such issues.

Implementing MLOps on Databricks

MLOps Stack provides a means of accelerating the creation of an MLOps workflow similar to the one outlined in the reference architecture section above. This repository provides a customizable stack for starting new ML projects on Databricks. After instantiating, a new project will have CI/CD pipelines and a number of example ML pipelines such as a model training pipeline, model deployment pipeline and batch inference pipeline, among others.

These pipelines are deployed to specified Databricks workspaces as Databricks Workflows using the Databricks CLI with Databricks asset bundles. Databricks asset bundles in particular enable the ability to programmatically validate, deploy and run Databricks Workflows such as Databricks jobs, and Delta Live Tables. Additionally it provides the ability to manage MLflow resources within Databricks such as MLflow experiments and MLflow models.

CHAPTER 6

LLMOps

With the recent rise of Generative AI we are prompted to consider how MLOps processes should be adapted to this new class of AI-powered applications. In this section we focus on the productionization of large language models (LLMs), the most prevalent form of Generative AI.

LLMs have splashed into the mainstream of business and news, and there is no doubt that they will continue to disrupt countless industries. In addition to bringing great potential, they present a new set of questions for MLOps:

- Is prompt engineering part of operations, and if so, what is needed?
- Since the "large" in "LLM" is an understatement, how do cost/performance trade-offs change?
- Is it better to use paid APIs or to fine-tune one's own model?
- …and many more!

The good news is that "LLMOps" (MLOps for LLMs) is not that different from traditional MLOps. However, some parts of your MLOps platform and process may require changes, and your team will need to learn a mental model of how LLMs coexist alongside traditional ML in your operations.

In this section, we will explore how MLOps changes with the introduction of LLMs. We will discuss several key topics in detail, from prompt engineering and fine-tuning, to packaging and cost/performance trade-offs. We also provide a reference architecture diagram to illustrate what may change in your production environment.

What changes with LLMs?

For those not familiar with large language models (LLMs), see this summary for a quick introduction. In short, LLMs are a new class of natural language processing (NLP) models that have significantly surpassed their predecessors in size and performance across a variety of tasks, such as open-ended question answering, summarization and execution of near-arbitrary instructions.

From the perspective of MLOps, LLMs bring new requirements, with implications for MLOps practices and platforms. We briefly summarize key properties of LLMs and the implications for MLOps here, and we will delve into more detail in the next section.

KEY PROPERTIES OF LLMS	IMPLICATIONS FOR MLOPS
LLMs are available in many forms: - Very general proprietary models behind paid APIs - Open source models that vary from general to specific applications - Custom models fine-tuned for specific applications	**Development process:** Projects often develop incrementally, starting from existing, third-party or open source models and ending with custom fine-tuned models.
Many LLMs take general natural language queries and instructions as input. Those queries can contain carefully engineered "prompts" to elicit the desired responses.	**Development process:** Designing text templates for querying LLMs is often an important part of developing new LLM pipelines. **Packaging ML artifacts:** Many LLM pipelines will use existing LLMs or LLM serving endpoints; the ML logic developed for those pipelines may focus on prompt templates, agents or "chains" instead of the model itself. The ML artifacts packaged and promoted to production may frequently be these pipelines, rather than models.
Many LLMs can be given prompts with examples and context, or additional information to help answer the query.	**Serving infrastructure:** When augmenting LLM queries with context, it is valuable to use previously uncommon tooling such as vector databases to search for relevant context.
LLMs are very large deep learning models, often ranging from gigabytes to hundreds of gigabytes.	**Serving infrastructure:** Many LLMs may require GPUs for real-time model serving. **Cost/performance trade-offs:** Since larger models require more computation and are thus more expensive to serve, techniques for reducing model size and computation may be required.
LLMs are hard to evaluate via traditional ML metrics since there is often no single "right" answer.	**Human feedback:** Since human feedback is essential for evaluating and testing LLMs, it must be incorporated more directly into the MLOps process, both for testing and monitoring and for future fine-tuning.

The list above may look long, but as we will see in the next section, many existing tools and processes only require small adjustments in order to adapt to these new requirements. Moreover, many aspects do not change:

- The separation of development, staging and production remains the same.
- Git version control and the MLflow Model Registry in Unity Catalog remain the primary conduits for promoting pipelines and models toward production.
- The Lakehouse architecture for managing data remains valid and essential for efficiency.
- Existing CI/CD infrastructure should not require changes.
- The modular structure of MLOps remains the same, with pipelines for model training, model inference, etc.

Key components of LLM-powered applications

In the following section we will delve into the more detailed aspects of building AI-powered applications using LLMs. The field of LLMOps is quickly evolving, however the following have emerged as key components and considerations to bear in mind. Some, but not necessarily all of the following components make up a single LLM-based application.

Prompt engineering

Prompt engineering is the practice of adjusting the text prompts given to an LLM to elicit more accurate or relevant responses. While it's a nascent field, several best practices are emerging. We will discuss tips and best practices and link to useful resources.

1. Prompts and prompt engineering are model-specific. A prompt given to two different models will generally not produce the same results. Similarly, prompt engineering tips do not apply to all models. In extreme cases, many LLMs have been fine-tuned for specific NLP tasks and do not require prompts. On the other hand, very general LLMs benefit greatly from carefully crafted prompts.

2. When approaching prompt engineering, go from simple to complex: track, templatize and automate.

   - Start by tracking queries and responses so that you can compare them and iterate to improve prompts. Existing tools such as MLflow provide tracking capabilities; see MLflow LLM Tracking for more details. Checking structured LLM pipeline code into version control also helps with prompt development, for git diffs allow you to review changes to prompts over time. Also see the section below on packaging model and pipelines for more information about tracking prompt versions.
   - Then, consider using tools for building prompt templates, especially if your prompts become complex. Newer LLM-specific tools such as LangChain and LlamaIndex provide such templates and more.
   - Finally, consider automating prompt engineering by replacing manual engineering with automated tuning. Prompt tuning turns prompt development into a data-driven process akin to hyperparameter tuning for traditional ML. The DSPy framework is a good example of a tool for both defining and automatically optimizing LLM pipelines.

Resources

There are lots of good resources about prompt engineering, especially for popular models and services:

→ DeepLearning.AI course on ChatGPT Prompt Engineering

→ DAIR.AI Prompt Engineering Guide

→ Best practices for prompt engineering with the OpenAI API

→ Replicate blog post – A guide to prompting Llama 2

3 Most prompt engineering tips currently published online are for ChatGPT, due to its immense popularity. Some of these generalize to other models as well. We will provide a few tips here:

- Use clear, concise prompts, which may include an instruction, context (if needed), a user query or input, and a description of the desired output type or format.
- Provide examples in your prompt ("few-shot learning") to help the LLM to understand what you want.
- Tell the model how to behave, such as telling it to admit if it cannot answer a question.
- Tell the model to think step-by-step or explain its reasoning.
- f your prompt includes user input, use techniques to prevent prompt hacking, such as making it very clear which parts of the prompt correspond to your instruction vs. user input.

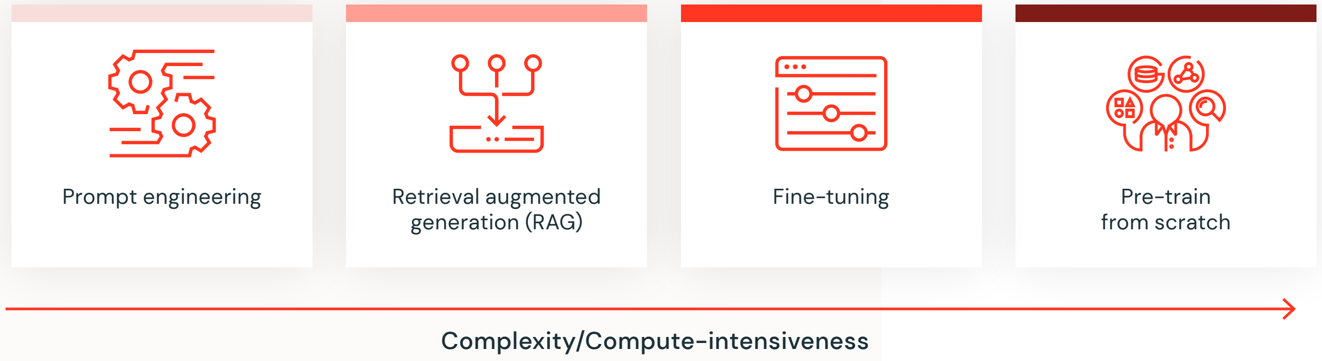
Leveraging your own data

While prompt engineering can yield remarkable results, it might not always suffice, especially in circumstances where domain-specific knowledge is required, or where contextual data is frequently updated. Incorporating your own data in LLM-powered applications can not only enhance a model's performance, but may also provide a strategic edge through customizing a model's output to your specific domain or use case requirements. Leveraging proprietary data can be the key differentiator in achieving superior results and gaining a competitive advantage.

We provide a high level overview of the various approaches that can be taken to leverage your own data with LLMs, and explore broadly when they should be used. In subsequent sections we will unpack these approaches in more detail.

Method	Definition	Primary use case	Data requirements	Training time	Advantage	Considerations
Prompt engineering	Crafting specialized prompts to guide LLM behavior	Quick, on-the-fly model guidance	None	None	Fast, cost-effective, no training required	Less control than fine-tuning
Retrieval augmented generation (RAG)	Combining an LLM with external knowledge retrieval	Dynamic datasets & external knowledge	External knowledge base or database (e.g. vector database)	Moderate (e.g. computing embeddings)	Dynamically updated context, enhanced accuracy	Significantly increases prompt length and inference computation
Fine-tuning	Adapting a pre-trained LLM to specific datasets or domains	Domain or task specialization	Thousands of domain-specific or instruction examples	Moderate — long (depending on data size)	Granular control, high specialization	Requires labeled data, computational cost
Pre-training	Training an LLM from scratch	Unique tasks or domain-specific corpora	Large datasets (billions to trillions of tokens	Long (days to many weeks)	Maximum control, tailored for specific needs	Extremely resource-intensive

Note that the techniques outlined above are not mutually exclusive of one another. Rather, they can (and should) be combined to take advantage of the strengths of each. For instance, while you might fine-tune a model for a specific task, you can also employ prompt engineering to guide its response at inference time. Similarly, a pre-trained LLM can be further enhanced with RAG to dynamically fetch and incorporate external knowledge.


Prompt engineering
Retrieval augmented generation (RAG)
Fine-tuning
Pre-train from scratch
Complexity/Compute-intensiveness

When considering which approach to take, like any ML application, it's crucial to assess your specific needs, business objectives, and constraints. A good rule of thumb is to start with the simplest approach possible, such as prompt engineering with a third-party LLM API, to establish a baseline. Once this baseline is in place, you can incrementally integrate more sophisticated strategies like RAG or fine-tuning to refine and optimize performance. Using standard MLOps tools such as MLflow is equally crucial in LLM applications to track performance over different approach iterations.

- The choice of technique will be informed by a range of factors, including (but not limited to):
- The volume and quality of your data
- Required application response time
- Computational resources available
- Budgetary constraints
- The specific domain or application at hand

Regardless of the technique selected, building a solution in a well-structured, modularized manner will ensure that you will be prepared to iterate and adapt as you uncover new insights and challenges. In the following sections we will look at each of these techniques to leverage your data, outlining the considerations and best practices associated with each.

Retrieval augmented generation (RAG)

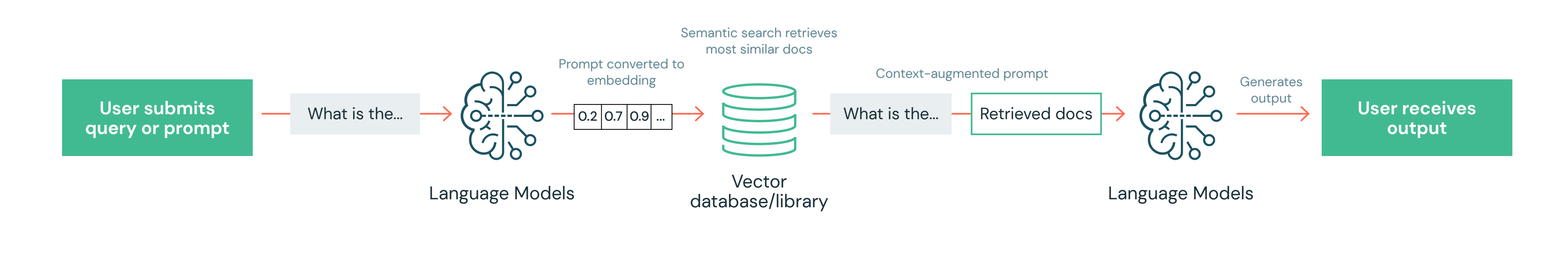
Retrieval augmented generation (RAG) offers a dynamic solution to a fundamental limitation of LLMs: their inability to access information beyond their training data cut-off point. RAG connects static LLMs with real-time data retrieval. Instead of relying solely on pre-trained knowledge, a RAG workflow pulls relevant information from external sources and injects it into the context provided to a model. RAG workflows are often used in document question-answering use cases where the answer might evolve over time, or come from up-to-date, domain-specific documents that were not part of the mode's original training data.

RAG provides a number of key benefits:

- **LLMs as reasoning engines:** RAG ensures that model responses are not just based on static training data. Rather, the model is used as a reasoning engine augmented with external data sources to provide responses that are up-to-date, accurate and relevant.
- **Reduce hallucinations:** By grounding the model's input on external knowledge, RAG attempts to mitigate the risk of producing "hallucinations" — instances where the model might generate inaccurate or fabricated information.
- **Domain-specific contextualization:** A RAG workflow can be tailored to interface with proprietary or domain-specific data. This ensures that the LLM's outputs are not only accurate but also contextually relevant, catering to specialized queries or domain-specific needs.
- **Efficiency and cost-effectiveness:** In use cases where the aim is to create a solution adapted to domain-specific knowledge, RAG offers an alternative to fine-tuning LLMs (discussed below) by enabling in-context learning without the overhead associated with traditional fine-tuning. This can be particularly beneficial in scenarios where models need to be frequently updated with new data. Note that where RAG reduces development time and cost, fine-tuning by contrast reduces inference time and cost.

TYPICAL RAG WORKFLOW

There are many ways to implement a RAG system, depending on specific needs and data nuances. Below we outline one commonly adopted workflow to provide a foundational understanding of the process.



Source: Databricks – Large Language Models: Application through Production

- **User prompt**
  The process begins with a user query or prompt. This input serves as the foundation for what the RAG system aims to retrieve.
- **Embedding conversion**
  The user's prompt is transformed into a high-dimensional vector (or embedding). This representation captures the semantic essence of the prompt and is used to search for relevant information in the database.
- **Information retrieval**
  Using the prompt's vector representation, RAG queries the external data sources or databases. A vector database (see below) can be particularly effective here, allowing for efficient and accurate data fetching based on similarity measures.
- **Context augmentation**
  The most relevant pieces of information are retrieved and concatenated to the initial prompt. This enriched context provides the LLM with supplemental information which can be used to produce a more informed response.

- **Response Generation**
  With the augmented context, the language model processes the combined data (original prompt + retrieved information) and generates a contextually relevant response.

- **Feedback Loop**
  Some RAG implementations might encompass a multi-hop feedback mechanism (see the following paper for an example). In cases where the response is deemed unsatisfactory, the system can revisit its search criteria, tweak the context, or even refine its retrieval strategy, subsequently generating a new answer.

Resources

→ Databricks — Using MLflow AI Gateway and Llama 2 to Build Generative AI Apps

→ LangChain question-answer example

→ Eugene Yan — Patterns for Building LLM-based Systems & Products

Vector Database

Retrieval augmented generation (RAG) hinges on the efficient retrieval of relevant data. At the heart of this retrieval process are embeddings — numerical representations of text data. To understand how RAG utilizes these vectors, let's first distinguish between a number of terms.

Vector index
A specialized data structure optimized to facilitate similarity search within a collection of vector embeddings.

Vector library
A tool to manage vector embeddings and conduct similarity searches. They predominantly:

- Operate on in-memory indexes.
- Focus solely on vector embeddings, often requiring a secondary storage mechanism for the actual data objects.
- Are typically immutable; post-index creation changes necessitate a complete rebuild of the index.

Examples of vector libraries include FAISS, Annoy, and ScaNN.

Vector database

Distinguished from vector libraries, vector databases:

- Store both the vector embeddings and the actual data objects, permitting combined vector searches with advanced filtering.
- Offer full CRUD (create, read, update, delete) operations, allowing dynamic adjustments without rebuilding the entire index.
- Are generally better suited for production-grade deployments due to their robustness and flexibility.

Examples of vector databases include Chroma, and Milvus.

In the RAG workflow described in the previous section we assume that the retrieved external data has been converted into embeddings. These embeddings are stored in a vector index, managed either through a vector library, or more holistically with a vector database. The choice between the two often hinges on specific requirements of the application, the volume of data, and the need for dynamic updates.

BENEFITS OF VECTOR DATABASES IN A RAG WORKFLOW

The following are a number of reasons why a vector database may be preferable over a vector library when implementing a RAG workflow:

- **Holistic data management:** Storing both vector embeddings and original data objects allows a RAG system to retrieve relevant context without needing to integrate with multiple systems.
- **Advanced filtering:** Beyond just similarity search, vector databases allow for application of filters on the stored data objects. This ensures more precise retrieval, enabling RAG to fetch contextually relevant and specific information based on both semantic similarity and metadata criteria.
- **Dynamic updates:** In fast-evolving domains, the ability to update the database without a complete rebuild of the vector index ensures that the language model accesses the up-to-date information.
- **Scalability:** Vector databases are designed to handle vast amounts of data, ensuring that as data grows, the RAG system remains efficient and responsive.

Fine-tuning LLMs

While prompt engineering and retrieval augmented generation (RAG) offer robust methods to guide a model's behavior, there are instances where they might not be adequate, especially for entirely novel or domain-specific tasks. In such cases, fine-tuning a LLM can be advantageous.

Fine-tuning is the process of adapting a pre-trained LLM on a comparatively smaller dataset that is specific to an individual domain or task. During the fine-tuning process, only a small number of weights are updated, allowing it to learn new behaviors and specialize in certain tasks.

The term "fine-tuning" can refer to several concepts, with the two most common forms being:

- **Supervised instruction fine-tuning:** This approach involves continuing training of a pre-trained LLM on a dataset of input-output training examples – typically conducted with thousands of training examples. Instruction fine-tuning is effective for question-answering applications, enabling the model to learn new specialized tasks such as information retrieval or text generation. The same approach is often used to tune a model for a single specific task (e.g. summarizing medical research articles), where the desired task is represented as an instruction in the training examples.

- **Continued pre-training:** This fine-tuning method does not rely on input and output examples but instead uses domain-specific unstructured text to continue the same pre-training process (e.g. next token prediction, masked language modeling). This approach is effective when the model needs to learn new vocabulary or a language it has not encountered before.

WHEN TO USE FINE-TUNING?

Choosing to fine-tune an open source LLM offers several advantages that tailor the model's behavior to better fit specific organizational needs. The following are a number of motivations behind choosing to fine-tune:

- **Customization and specialization:** LLMs trained on large, generic datasets – such as those provided by third-party APIs – often have broad knowledge but lack depth in niche areas. Fine-tuning allows organizations to specialize models for their specific domains or applications.
- **Full control over model behavior:** Fine-tuning provides granular control over the model's outputs. It allows organizations to address specific biases, enforce correctness and refine a model's behavior based on feedback.

FINE-TUNING IN PRACTICE

Fine-tuning, while advantageous, comes with practical considerations. Often, the optimal approach is not solely fine-tuning, but a blend of fine-tuning and retrieval methods like RAG. For example, you might fine-tune a model to generate specific outputs but also use RAG to inject data relevant to user queries.

Notably, fine-tuning large models with billions of parameters comes with its own set of challenges, particularly in terms of computational resources. To accommodate the training of such models, modern deep learning libraries like PyTorch FSDP and Deepspeed employ techniques like Zero Redundancy Optimizer (ZeRO), Tensor Parallelism, and Pipeline Parallelism. These methods optimize the distribution of the model training process across multiple GPUs, ensuring efficient use of resources.

A resource-efficient alternative to fine-tuning all parameters of a LLM is a class of methods referred to as parameter-efficient fine-tuning (PEFT). PEFT methods, such as LoRA and IA3 fine-tune LLMs by adjusting a limited subset of model parameters or a small number of extra-model parameters. These approaches not only conserve GPU memory, but often match the performance of full model fine-tuning. Open source libraries such as Hugging Face's PEFT library have been developed to easily implement this family of fine-tuning techniques for a subset of models.

Lastly, it's worth noting the significance of human feedback in a fine-tuning context, especially for applications like question-answering systems or chat interfaces. Such feedback helps refine the model, ensuring its outputs align more closely with user needs and expectations. To facilitate this iterative process, libraries like Hugging Face's trl or CarperAI's trlX can be used to apply methods like Proximal Policy Optimization (PPO) which incorporate human feedback into the fine-tuning process.

Pre-training

Pre-training a model from scratch refers to the process of training a language model on a large corpus of data (e.g. text, code) without using any prior knowledge or weights from an existing model. This is in contrast to fine-tuning, where an already pre-trained model is further adapted to a specific task or dataset. The output of full pre-training is a base model that can be directly used or further fine-tuned for downstream tasks.

WHEN TO USE PRE-TRAINING?

Choosing to pre-train an LLM from scratch is a significant commitment, both in terms of data and computational resources. Here are some scenarios where it makes sense:

1. Unique data sources: If you possess a unique and extensive corpus of data that is distinct from what available pre-trained LLMs have seen, it might be worth pre-training a model to capture this uniqueness.
2. Domain specificity: Organizations might want a base model tailored to their specific domain (e.g., medical, legal, code) to ensure even the foundational knowledge of the model is domain-specific.
3. Full control over training data: Pre-training from scratch offers transparency and control over the data the model is trained on. This may be essential for ensuring data security, privacy, and custom tailoring of the model's foundational knowledge.
4. Avoiding third-party biases: Pre-training ensures that your LLM application does not inherit biases or limitations from third-party pre-trained models.

PRE-TRAINING IN PRACTICE

Given the resource-intensive nature of pre-training, careful planning and sophisticated tooling are required. Libraries like PyTorch FSDP and Deepspeed, mentioned previously in the fine-tuning section, are similarly required for their distributed training capabilities when pre-training an LLM from scratch. The following only scratches the surface on some of the considerations one must take into account when pre-training an LLM:

- **Large scale data preprocessing:** A pre-trained model is only as good as the data it is trained on. Thus, it becomes vitally important to ensure robust data preprocessing is conducted prior to model training. Given the scale of the training data involved, this preprocessing typically requires distributed frameworks like Apache Spark. Consideration must be given to factors such as dataset mix and deduplication techniques to ensure the model is exposed to a wide variety of unique data points.
- **Hyperparameter selection and tuning:** Before executing full-scale training of an LLM, determining the set of optimal hyperparameters is crucial. Given the high computational cost associated with LLM training, extensive hyperparameter sweeps are not always feasible. Instead, informed decisions based on smaller-scale searches or prior research are employed. Once a promising set is identified, these hyperparameters are used for the full training run. Tooling like MLflow is essential to manage and track these experiments.
- **Maximizing resource utilization:** Given the high costs associated with long-running distributed GPU training jobs it is hugely important to maximize resource utilization. MosaicML's composer is an example of a library that uses PyTorch FSDP with additional optimizations to maximize Model FLOPs Utilization (MFU) and Hardware FLOPs Utilization (HFU) during training.
- **Handling GPU failures:** Training large models can run for days or even weeks. During such large scale training for this length of time, hardware failures, especially GPU failures, can (and typically do) occur. It is essential to have mechanisms in place to handle such failures gracefully.
- **Monitoring and evaluation:** Close monitoring of the training process is essential. Saving model checkpoints regularly and evaluating on validation sets not only act as safeguards but also provide insights into model performance and convergence trends.

In cases where pre-training an LLM from scratch is required, MosaicML Training provides a platform to conduct training of multi-billion parameter models in a highly optimized and automated manner. Automatically handling GPU failures and resuming training without human intervention, and MosaicML Streaming for efficient streaming of data into the training process are just some of the capabilities provided out-of-the-box.

Third-party APIs vs. self-hosted models

Choosing between third-party LLM APIs and self-hosting your own models has implications for cost, control, and data security. Here are a number of concerns one should consider when making that decision:

- **Data security and privacy:** Using third-party APIs often involves sending data to external servers. This can pose risks, especially when dealing with sensitive or proprietary information. In some cases regulations may simply prohibit data leaving data regions, or being sent to non-compliant environments. Hosting your own model ensures that data does not leave your secure environment, while retaining full access to the trained model.
- **Predictable and stable behavior:** Proprietary SaaS models can undergo updates or changes without prior notice. Such changes can lead to unpredictable model behavior. When hosting your own LLM, you have full control over its versions and updates.
- **Vendor lock-in:** Relying on third-party APIs means being dependent on the vendor's terms, pricing, and availability. With this, there is the risk of the service being deprecated or price changes. By self-hosting, you maintain autonomy and guard against potential vendor lock-in.

Model Evaluation

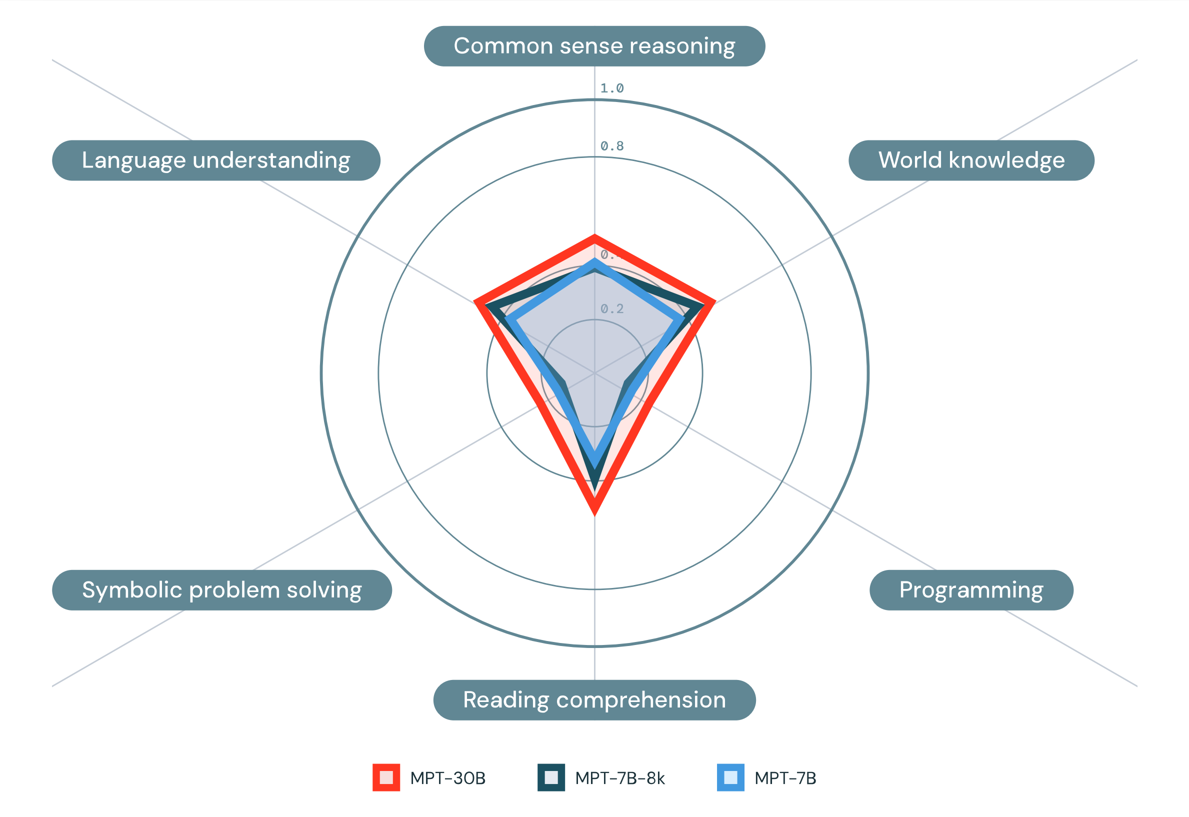
Evaluating LLMs is a challenging and evolving domain, primarily because LLMs often demonstrate uneven capabilities across different tasks. An LLM might excel in one benchmark, but slight variations in the prompt or problem can drastically affect its performance. The dynamic nature of LLMs and their vast potential applications only amplify the challenge of establishing comprehensive evaluation standards.

The following are a number of present challenges involved with evaluating LLM-powered applications:

- Variable performance: LLMs can be sensitive to prompt variations, demonstrating high proficiency in one task but faltering with slight deviations in prompts.
- Lack of ground truth: Since most LLMs output natural language, it is very difficult to evaluate the outputs via traditional NLP metrics (BLEU, ROUGE, etc.). For example, suppose an LLM were used to summarize a news article. Two equally good summaries might have almost completely different words and word orders, so even defining a "ground-truth" label becomes difficult or impossible.
- Domain-specific evaluation: For domain-specific fine-tuned LLMs, popular generic benchmarks may not capture their nuanced capabilities. Such models are tailored for specialized tasks, making traditional metrics less relevant. This divergence often necessitates the development of domain-specific benchmarks and evaluation criteria. See the example of Replit's code generation LLM.
- Reliance on human judgment: It is often the case that LLM performance is being evaluated in domains where text is scarce or there is a reliance on subject matter expert knowledge. In such scenarios, evaluating LLM output can be costly and time consuming.

Some prominent benchmarks used to evaluate LLM performance include:

- BIG-bench (Beyond the Imitation Game benchmark)
  A dynamic benchmarking framework, currently hosting over 200 tasks, with a focus on adapting to future LLM capabilities.
- EluetherAI LM Evaluation Harness
  A holistic framework that assesses models on over 200 tasks, merging evaluations like BIG-bench and MMLU, promoting reproducibility and comparability.
- Mosaic Model Gauntlet
  An aggregated evaluation approach, categorizing model competency into six broad domains (shown below) rather than distilling to a single monolithic metric.



Source: Mosaic Model Gauntlet

LLMS AS EVALUATORS

A number of alternative approaches have been proposed to use LLMs themselves to assist with model evaluation. These range from using LLMs to generate evaluations, through to entrusting LLMs as judges to evaluate other models capabilities or outputs. Ideally, when evaluating an LLM, a larger or more capable LLM should be employed as the evaluator. This premise is based on the understanding that a "smarter" model can plausibly produce a more accurate evaluation. The benefits of using LLMs as evaluators include:

- Speed, as they are faster than human evaluators
- Cost-effectiveness
- I n certain cases, they offer comparable accuracy to human evaluators

See the following Databricks blog post for a more detailed exploration on using LLMs as evaluators.

HUMAN FEEDBACK IN EVALUATION

While human feedback is important in many traditional ML applications, it becomes much more important for LLMs. Humans – ideally your end users – become essential for validating LLM output. While you can pay human labelers to compare or rate model outputs (or have an LLM evaluate your application as mentioned above), the best practice for user-facing applications is to build human feedback into the applications from the outset. For example, a tech support chatbot may have a "click here to chat with a human" option, which provides implicit feedback indicating whether the chatbot's responses were helpful. Explicit feedback can also be captured in this case by presenting users with the ability to click thumbs up/down buttons.

Packaging models or pipelines for deployment

In traditional ML, there are generally two types of ML logic to package for deployment: models and pipelines. These artifacts are generally managed toward production via a model registry and Git version control, respectively.

With LLMs, it is common to package ML logic in new forms. These may include:

- A lightweight call to an LLM API service (third-party or internal)
- A "chain" from LangChain or an analogous pipeline from another tool. The chain may call an LLM API or a local LLM model.
- An LLM or an LLM+tokenizer pipeline, such as a Hugging Face pipeline. This pipeline may use a pretrained model or a custom fine-tuned model.
- An engineered prompt, possibly stored as a template in a tool such as LangChain.

Though LLMs add new terminology and tools for composing ML logic, all of the above still constitute models and pipelines. Thus, the same tooling such as MLflow can be used to package LLMs and LLM pipelines for deployment. Built-in model flavors include:

- PyTorch and TensorFlow
- Hugging Face Transformers (relatedly, see Hugging Face Transformers' MLflowCallback)
- LangChain
- OpenAI API
- (See the documentation for a complete list)

For other LLM pipelines, MLflow can package the pipelines via the MLflow pyfunc flavor, which can store arbitrary Python code.

Note about prompt versioning
Just as it is helpful to track model versions, it is helpful to track prompt versions (and LLM pipeline versions, more generally). Packaging prompts and pipelines as MLflow models simplifies versioning. Just as a newly retrained model can be tracked as a new model version in the MLflow model Registry, a newly updated prompt can be tracked as a new model version.

Note about deploying models vs. code
Your decisions around packaging ML logic as version-controlled code vs. registered models will help to inform your decision about choosing between the deploy models, deploy code, and hybrid architectures. Review the subsection below about human feedback, and make sure that you have a well-defined testing process for whatever artifacts you choose to deploy.

LLM Inference

Inference for LLM-powered solutions can come in a number of forms. In this section we focus on two primary methods for LLM inference: real-time and batch.

REAL-TIME INFERENCE

Real-time inference is ideal for applications requiring immediate responses, such as chat interfaces or question-answering systems. Real-time LLM inference can be further bucketed into the following:

- **Third-party LLM API with pre- and post- processing logic:**
  - Some inference pipelines, like RAG workflows, include pre-processing and post-processing logic, and use an external LLM API to generate output. Such pipelines can be deployed as REST API endpoints using services like Databricks Model Serving. For deployed LLM pipelines making requests to external APIs, CPU instance types are sufficient.
  - To standardize interactions with SaaS and OSS LLMs for such LLM pipelines, the MLflow AI Gateway can be used to manage requests to different LLM API providers. It offers a high-level interface that simplifies the interaction with these services by providing a unified endpoint to handle specific LLM related requests.
- **Pre-trained or fine-tuned OSS LLMs:**
  - When hosting your own LLMs, such as those downloaded from repositories like Hugging Face (and additionally fine-tuned), GPU instance types are typically required for optimized inference.

BATCH INFERENCE

Batch inference is well-suited for scenarios where immediate feedback is not critical (e.g. offline text summarization). For batch use cases, Spark can be leveraged to distribute inference across multiple machines (GPUs included).

INFERENCE WITH LARGE MODELS

Applicable to both real-time and batch inference scenarios is handling cases where large models exceed the memory of a single GPU. In such cases:

- Distribute serving across multiple GPUs, and/or;
- Consider loading the model with reduced precision, such as 8-bit or 4-bit quantization, to fit within memory constraints. Be aware that this option may affect the quality of the model's outputs.

Managing cost/performance trade-offs

One of the big Ops topics for LLMs is managing cost/performance trade-offs, especially for inference and serving. With "small" LLMs having hundreds of millions of parameters and large LLMs having hundreds of billions of parameters, computation can become a major expense. Thankfully, there are many ways to manage and reduce costs when needed. We will review some key tips for balancing productivity and costs.

1. **Start simple, but plan for scaling:** When developing a new LLM-powered application, speed of development is key, so it is acceptable to use more expensive options, such as paid APIs for existing models. As you go, make sure to collect data such as queries and responses. If the API provider terms of service permits it, you may be able to use this data to fine-tune a smaller, cheaper model which you can own.

2. **Scope out your costs:** How many queries per second do you expect? Will requests come in bursts? How much does each query cost? These estimates will inform you about project feasibility and will help you to decide when to consider bringing the model in-house with open source models and fine-tuning.

3. **Reduce costs by tweaking LLMs and queries:** There are many LLM-specific techniques for reducing computation and costs. These include shortening queries, tweaking inference configurations, and using smaller versions of models.

4. **Get human feedback:** It is easy to reduce costs but hard to say how changes impact your results unless you get human feedback from end users.

METHODS FOR REDUCING COSTS OF INFERENCE

USE A SMALLER MODEL

- Pick a different existing model. Try smaller versions of models (such as "llama-2-13b" instead of "llama-2-70b") or alternate architectures.
- Fine-tune a custom model. With the right training data, a fine-tuned small model can often perform as well or better than a large generic model.
- Use model distillation (or knowledge distillation). This technique "distills" the knowledge of the original model into a smaller model.
- Reduce floating point precision (quantization). Models can sometimes use lower precision arithmetic without losing much in quality.

REDUCE COMPUTATION FOR A GIVEN MODEL.

- Shorten queries and responses. Computation scales with input and output sizes, reducing costs by using more concise queries and responses.
- Tweak inference configurations. Some types of inference, such as beam search, require more computation.

OTHER

- Split traffic. If your return on investment (ROI) for an LLM query is low, then consider splitting traffic so that simpler, faster models or methods handle low ROI queries. Save LLM queries for high ROI traffic.
- Use pruning techniques. If you are training your own LLMs, there are pruning techniques that allow models to use sparse computation during inference. This reduces computation for most or all queries.

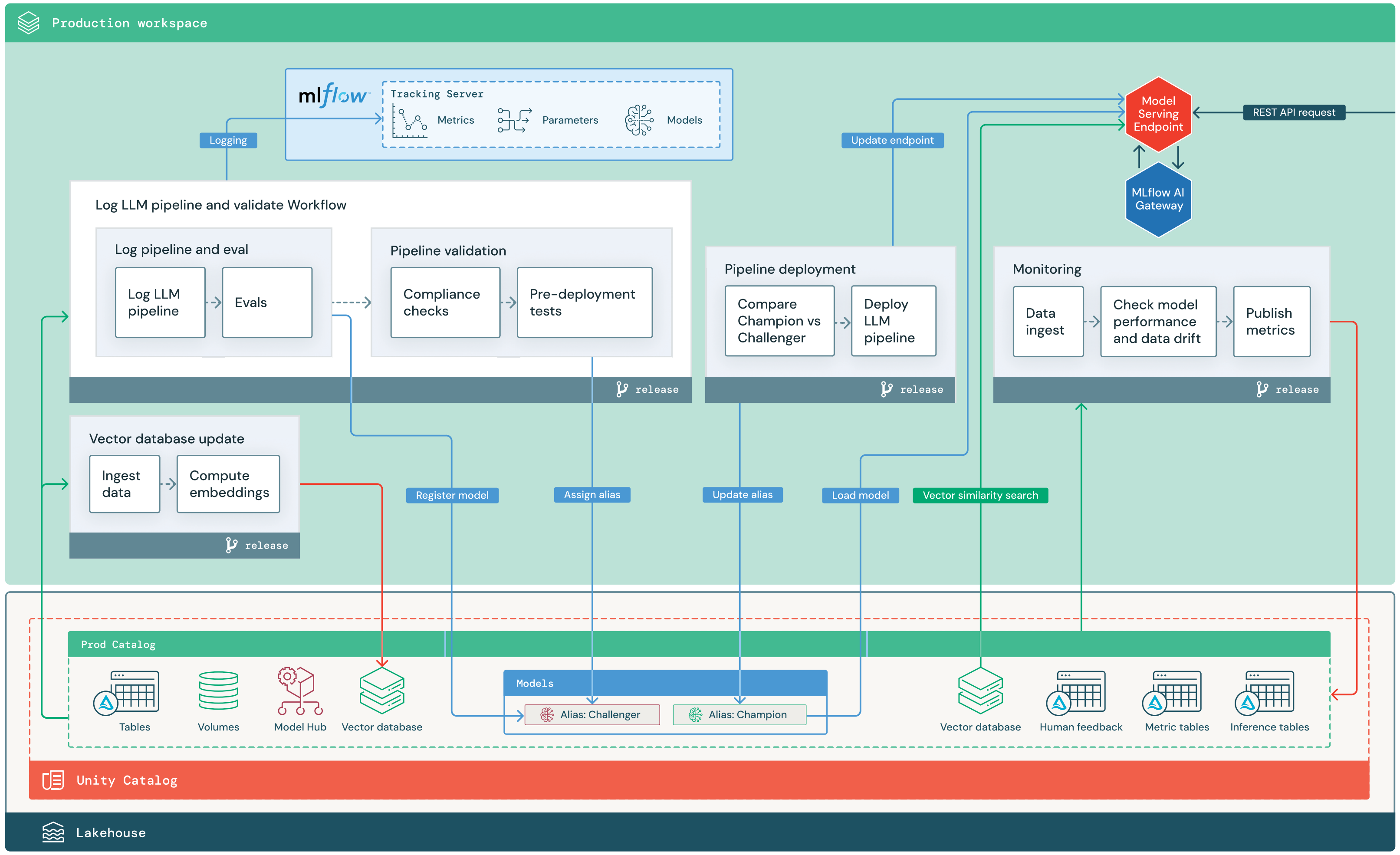
Reference architecture

To illustrate potential adjustments to your reference architecture from traditional MLOps, we provide a modified version of the previous production architecture for two separate LLM-based applications:

1. RAG workflow using a third-party API
2. RAG workflow using a self-hosted fine-tuned model.

Note that in either of these examples, the retrieval element using the vector database could be removed, and the LLM queried directly through the Model Serving endpoint.

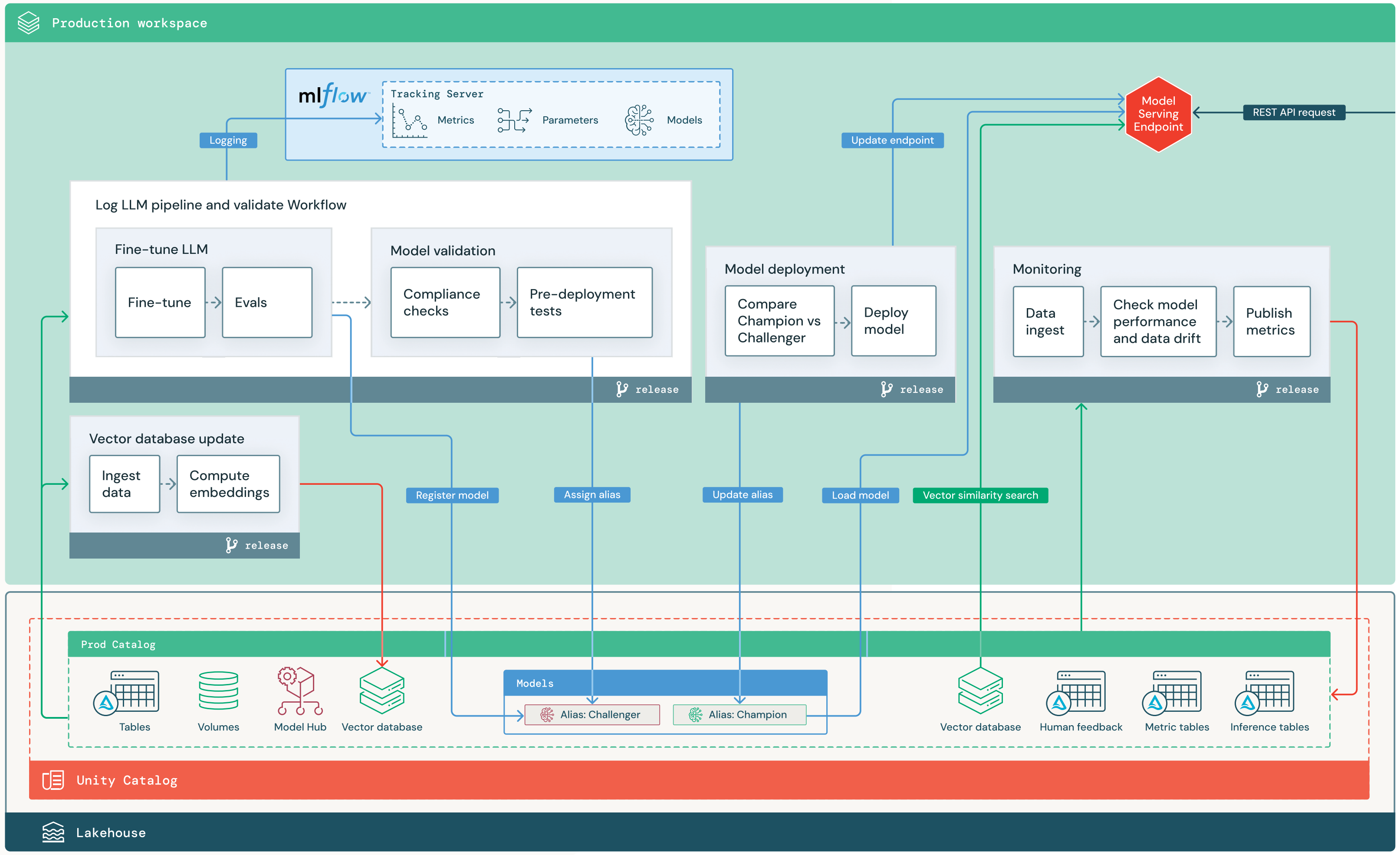
RAG with a third-party LLM API



Legend

▭ Workflow	→ Reads	◆ Git repo
▭ Job/Workflow task	→ Writes	⑂ Git branch
▭ CI/CD pipeline	→ MLflow API	⚙ Registered model

RAG with a fine-tuned OSS model



Legend

▭	Workflow	→	Reads	◆	Git repo
▭	Job/Workflow task	→	Writes	⑂	Git branch
▭	CI/CD pipeline	→	MLflow API	⚙	Registered model

The primary changes to the above production architectures are:

Model Hub

Since LLM applications often make use of existing, pretrained models, an internal or external model hub becomes a valuable part of the infrastructure. In the RAG with fine-tuned model example we illustrate using an existing base model from the model hub that is then fine-tuned in production.

Vector Database

Some (but not all) LLM applications use vector databases for fast similarity searches, most often to provide context or domain knowledge in LLM queries. To ensure that the deployed language model has access to up-to-date information, regular vector database updates can be scheduled as a Databricks job. Note that the logic to retrieve from the vector database and inject information into the LLM context can be packaged in the model artifact logged to MLflow using MLflow LangChain or PyFunc model flavors.

MLflow AI Gateway

In LLM-based applications where a third-party LLM API is used, the MLflow AI Gateway can be used as a standardized interface to route requests from vendors such as OpenAI and Anthropic. In addition to providing an enterprise-grade API gateway, the AI Gateway centralizes API key management and provides the ability to enforce cost controls.

Fine-tune LLM

Instead of a de novo model training pipeline, LLM applications will generally fine-tune an existing model (or use an existing model without any tuning). Fine-tuning is a lighter-weight process than training, but it is similar operationally. We represent model fine-tuning and model deployment as separate Databricks Workflows given that validating a fine-tuned model prior to deployment may be a manual process involving a human-in-the-loop.

Model Serving

In the case of RAG using a third-party API, one key architectural change is that the LLM pipeline will make external API calls, from the Model Serving endpoint to internal or third-party LLM APIs. It should be noted that this adds complexity, potential latency, and another layer of credential management. By contrast, in the fine-tuned model example, the model and its model environment will be deployed.

Human feedback in monitoring and evaluation

Human feedback loops may be used in traditional ML but become essential in most LLM applications. Human feedback should be managed like other data, ideally incorporated into monitoring based on near real-time streaming.

CHAPTER 7

Conclusion

In an era defined by data-driven decision making and intelligent automation, the importance of MLOps cannot be overstated. MLOps provides the essential scaffolding for developing, deploying, and maintaining AI models at scale, ensuring they remain accurate and continue to deliver business value. The emergence of LLMOps highlights the rapid advancement and specialized needs of the field of Generative AI. However, at its heart, LLMOps is still rooted in the foundational principles of MLOps.

Whether you are implementing traditional machine learning solutions or LLM-driven applications, the four core tenets remain constant:

- **Business goal:** Always keep your business goals in mind
- **Data-centric:** Prioritize a data-centric approach
- **Modular:** Implement solutions in a modular manner
- **Automated:** Aim for processes to guide automation

Databricks stands uniquely positioned as a unified, data-centric platform for both MLOps and LLMOps. Serving as the foundation, Unity Catalog provides a single governance solution for all data and AI assets. This is complemented by MLflow for experiment tracking, Model Serving for real-time deployment, Lakehouse Monitoring to ensure long term efficiency and performance stability, and Databricks Workflows to seamlessly orchestrate data pipelines.

As we look forward to the oncoming wave of AI advancements, it's clear that employing a robust MLOps strategy will remain central to unlocking AI's full potential. With firm MLOps foundations in place, organizations will be able to maximize their AI investments to drive innovation and deliver business value.