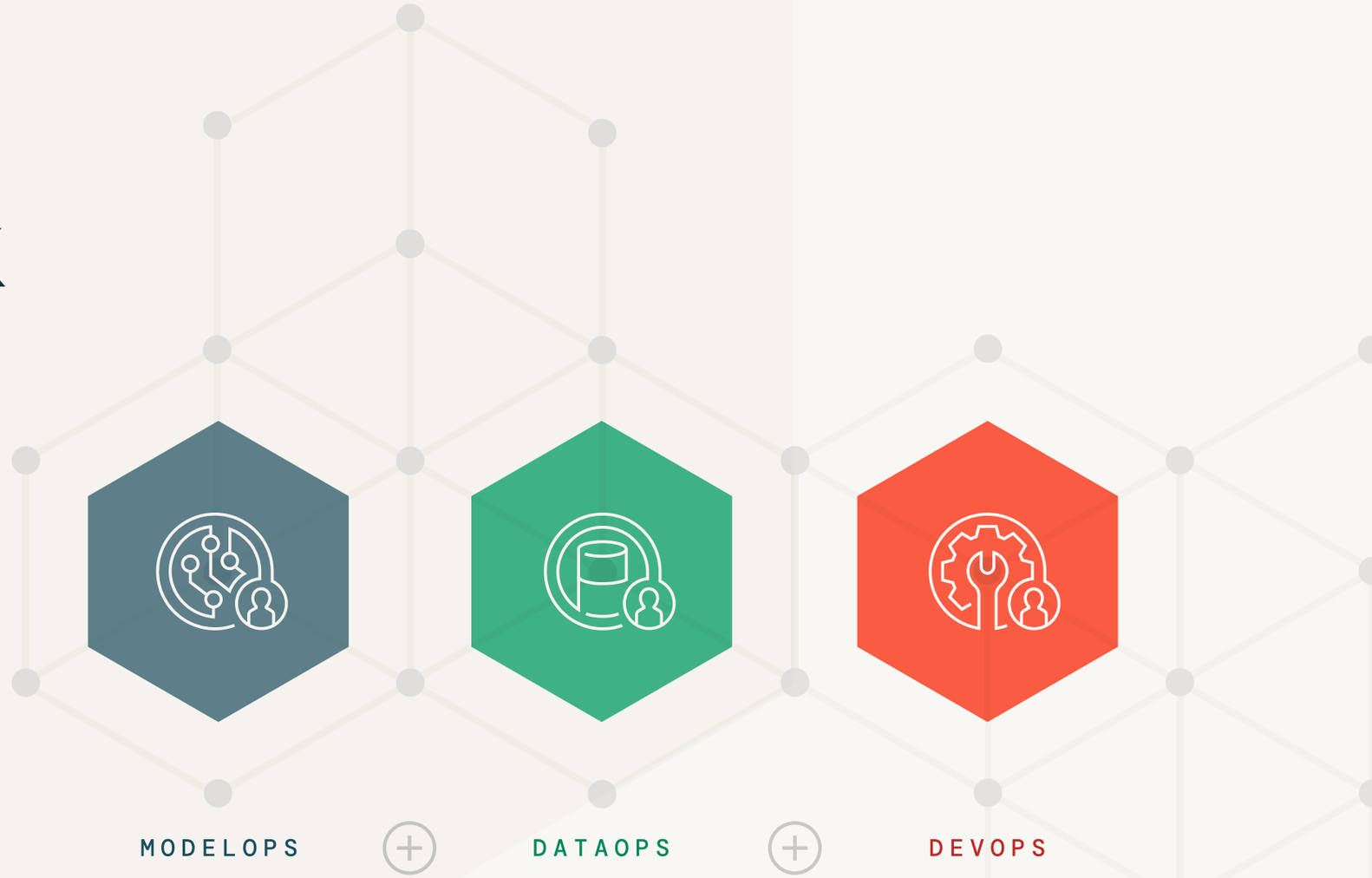
eBook

The Big Book of MLOps

A data-centric approach to build and scale AI, including LLMOps



databricks

JOSEPH BRADLEY | RAFI KURLANSIK | MATT THOMSON | NIALL TURBITT

Contents

AUTHORS:

Joseph Bradley
Lead Product Specialist

Rafi Kurlansik
Lead Product Specialist

Matt Thomson
Director, EMEA Product Specialists

Niall Turbitt
Lead Data Scientist

databricks

CHAPTER 1:

Introduction

Note: Our prescription for MLOps is general to any set of tools and applications, though we give concrete examples using Databricks features and functionality. We also note that no single architecture or prescription will work for all organizations or use cases. Therefore, while we provide guidelines for building MLOps, we call out important options and variations. This whitepaper is written primarily for ML engineers and data scientists wanting to learn more about MLOps, with high-level guidance and pointers to more resources.

The past decade has seen rapid growth in the adoption of machine learning (ML). While the early adopters were a small number of large technology companies that could afford the necessary resources, in recent times ML-driven business cases have become ubiquitous in all industries. Indeed, according to MIT Sloan Management Review, 83% of CEOs report that artificial intelligence (AI) is a strategic priority. This democratization of ML across industries has brought huge economic benefits, with Gartner estimating that $3.9T in business value will be created by AI in 2022.

However, building and deploying ML models is complex. There are many options available for achieving this but little in the way of well-defined and accessible standards. As a result, over the past few years we have seen the emergence of the machine learning operations (MLOps) field. **MLOps is a set of processes and automation for managing models, data and code to improve performance stability and long-term efficiency in ML systems.** Put simply, MLOps = ModelOps + DataOps + DevOps.

The concept of developer operations (DevOps) is nothing new. It has been used for decades to deploy software applications, and the deployment of ML applications has much to gain from it. However, strong DevOps practices and tooling alone are insufficient because ML applications rely on a constellation of artifacts (e.g., models, data, code) that require special treatment. Any MLOps solution must take into account the various people and processes that interact with these artifacts.

Here at Databricks we have seen firsthand how customers develop their MLOps approaches, some of which work better than others. We launched the open source MLflow project to help make our customers successful with MLOps, and with over 10 million downloads/month from PyPI as of May 2022, MLflow's adoption is a testament to the appetite for operationalizing ML models.

This whitepaper aims to explain how your organization can build robust MLOps practices incrementally. First, we describe the people and process involved in deploying ML applications and the need for operational rigor. We also provide general principles to help guide your planning and decision-making. Next, we go through the fundamentals of MLOps, defining terms and broad strategies for deployment. Finally, we introduce a general MLOps reference architecture, the details of its processes, and best practices.

People and process

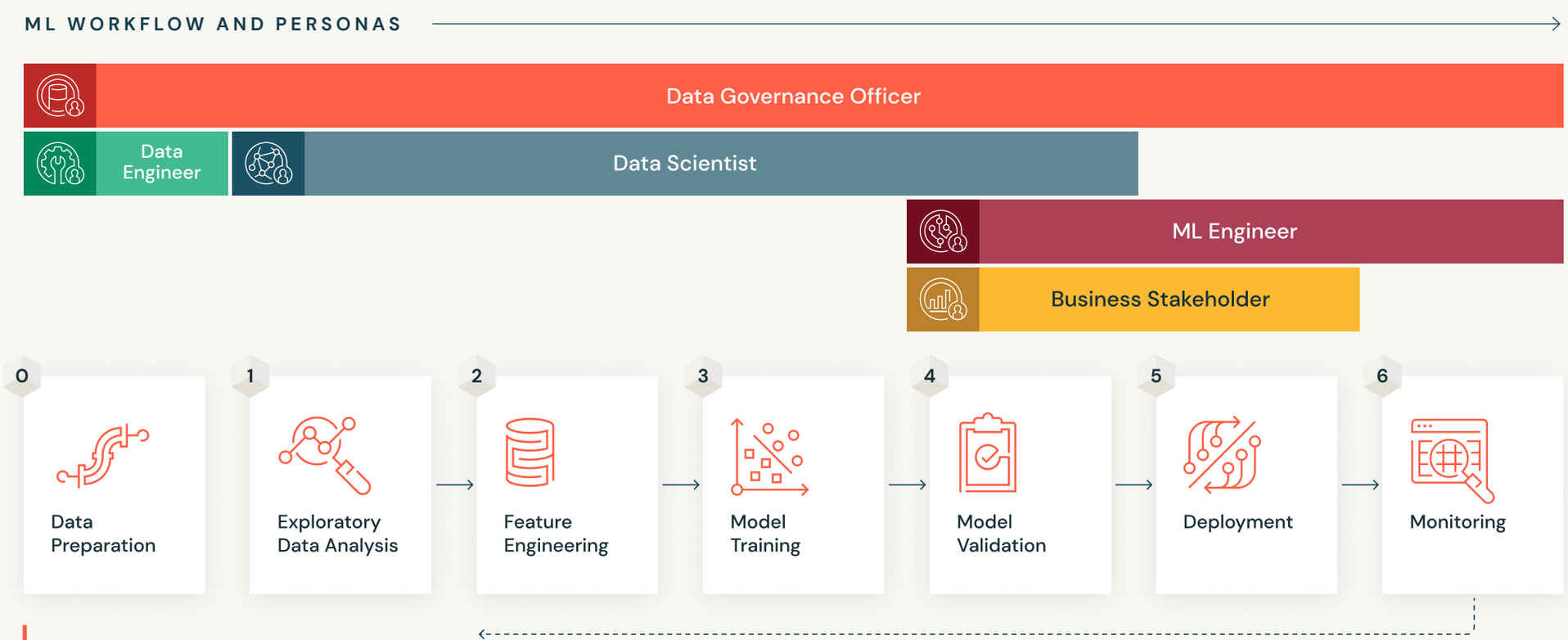


Figure 1

People

Building ML applications is a team sport, and while in the real world people "wear many hats," it is still useful to think in terms of archetypes. They help us understand roles and responsibilities and where handoffs are required, and they highlight areas of complexity within the system. We distinguish between the following personas:

ML PERSONAS

Data Engineer

Responsible for building data pipelines to process, organize and persist data sets for machine learning and other downstream applications.

Data Scientist

Responsible for understanding the business problem, exploring available data to understand if machine learning is applicable, and then training, tuning and evaluating a model to be deployed.

ML Engineer

Responsible for deploying machine learning models to production with appropriate governance, monitoring and software development best practices such as continuous integration and continuous deployment (CI/CD).

Business Stakeholder

Responsible for using the model to make decisions for the business or product, and responsible for the business value that the model is expected to generate.

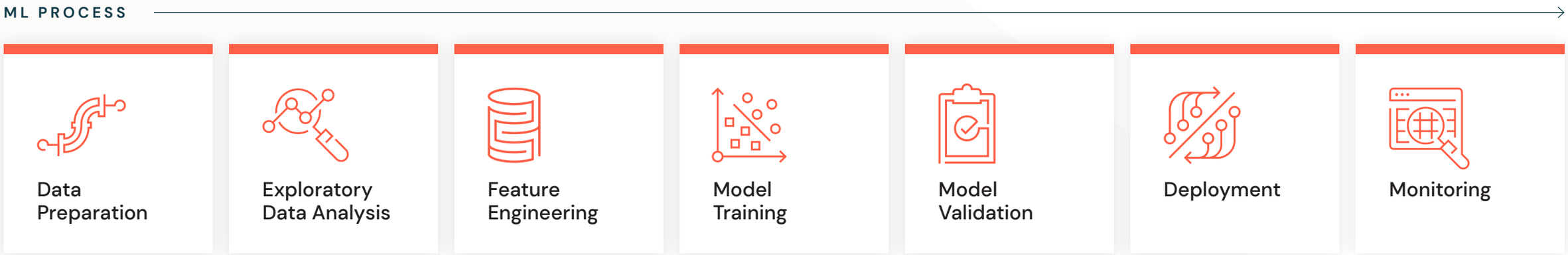
Data Governance Officer

Responsible for ensuring that data governance, data privacy and other compliance measures are adhered to across the model development and deployment process. Not typically involved in day-to-day operations.

Process

Together, these people develop and maintain ML applications. While the development process follows a distinct pattern, it is not entirely monolithic. The way you deploy a model has an impact on the steps you take, and using techniques like reinforcement learning or online learning will change some details. Nevertheless, these steps and personas involved are variations on a core theme, as illustrated in Figure 1 above.

Let's walk through the process step by step. Keep in mind that this is an iterative process, the frequency of which will be determined by the particular business case and data.



Data preparation

Prior to any data science or ML work lies the data engineering needed to prepare production data and make it available for consumption. This data may be referred to as "raw data," and in later steps, data scientists will extract features and labels from the raw data.

Exploratory data analysis (EDA)

Analysis is conducted by data scientists to assess statistical properties of the data available, and determine if they address the business question. This requires frequent communication and iteration with business stakeholders.

Feature engineering

Data scientists clean data and apply business logic and specialized transformations to engineer features for model training. These data, or features, are split into training, testing and validation sets.

Model training

Data scientists explore multiple algorithms and hyperparameter configurations using the prepared data, and a best-performing model is determined according to predefined evaluation metric(s).

Model validation

Prior to deployment a selected model is subjected to a validation step to ensure that it exceeds some baseline level of performance, in addition to meeting any other technical, business or regulatory requirements. This necessitates collaboration between data scientists, business stakeholders and ML engineers.

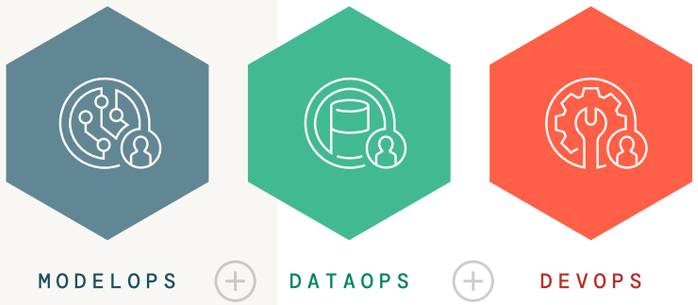
Deployment

ML engineers will deploy a validated model via batch, streaming or online serving, depending on the requirements of the use case.

Monitoring

ML engineers will monitor deployed models for signs of performance degradation or errors. Data scientists will often be involved in early monitoring phases to ensure that new models perform as expected after deployment. This will inform if and when the deployed model should be updated by returning to earlier stages in the workflow.

The data governance officer is ultimately responsible for making sure this entire process is compliant with company and regulatory policies.



Why should I care about MLOps?

Consider that the typical ML application depends on the aforementioned people and process, as well as regulatory and ethical requirements. These dependencies change over time — and your models, data and code must change as well. The data that were a reliable signal yesterday become noise; open source libraries become outdated; regulatory environments evolve; and teams change. ML systems must be resilient to these changes. Yet this broad scope can be a lot for organizations to manage — there are many moving parts! Addressing these challenges with a defined MLOps strategy can dramatically reduce the iteration cycle of delivering models to production, thereby accelerating time to business value.

There are two main types of risk in ML systems: **technical risk** inherent to the system itself and **risk of noncompliance** with external systems. Both of these risks derive from the dependencies described above. For example, if data pipeline infrastructure, KPIs, model monitoring and documentation are lacking, then you risk your system becoming destabilized or ineffective. On the other hand, even a well-designed system that fails to comply with corporate, regulatory and ethical requirements runs the risk of losing funding, receiving fines or incurring reputational damage. Recently, one private company's data collection practices were found to have violated the Children's Online Privacy Protection Rule (COPPA). The FTC fined the company $1.5 million and ordered it to destroy or delete the illegally harvested data, and all models or algorithms developed with that data.

With respect to efficiency, the absence of MLOps is typically marked by an overabundance of manual processes. These steps are slower and more prone to error, affecting the quality of models, data and code. Eventually they form a bottleneck, capping the ability for a data team to take on new projects.

Seen through these lenses, the aim of MLOps becomes clear: improve the long-term performance stability and success rate of ML systems while maximizing the efficiency of teams who build them. In the introduction, we defined MLOps to address this aim: MLOps is a **set of processes and automation** to manage **models, data and code** to meet the two goals of **stable performance and long-term efficiency in ML systems**. *MLOps = ModelOps + DataOps + DevOps*.

With clear goals, we are ready to discuss principles that guide design decisions and planning for MLOps.

Guiding principles

Given the complexity of ML processes and the different personas involved, it is helpful to start from simpler, high-level guidance. We propose several broadly applicable principles to guide MLOps decisions. They inform our design choices in later sections, and we hope they can be adapted to support whatever your business use case may be.

Always keep your business goals in mind

Just as the core purpose of ML in a business is to enable data-driven decisions and products, the core purpose of MLOps is to ensure that those data-driven applications remain stable, are kept up to date and continue to have positive impacts on the business. When prioritizing technical work on MLOps, consider the business impact: Does it enable new business use cases? Does it improve data teams' productivity? Does it reduce operational costs or risks?

Take a data-centric approach to machine learning

Feature engineering, training, inference and monitoring pipelines are data pipelines. As such, they need to be as robust as other production data engineering processes. Data quality is crucial in any ML application, so ML data pipelines should employ systematic approaches to monitoring and mitigating data quality issues. Avoid tools that make it difficult to join data from ML predictions, model monitoring, etc., with the rest of your data. The simplest way to achieve this is to develop ML applications on the same platform used to manage production data. For example, instead of downloading training data to a laptop, where it is hard to govern and reproduce results, secure the data in cloud storage and make that storage available to your training process.

Implement MLOps in a modular fashion

As with any software application, code quality is paramount for an ML application. Modularized code enables testing of individual components and mitigates difficulties with future code refactoring. Define clear steps (e.g., training, evaluation or deployment), supersteps (e.g., training-to-deployment pipeline) and responsibilities to clarify the modular structure of your ML application.

Process should guide automation

We automate processes to improve productivity and lower risk of human error, but not every step of a process can or should be automated. People still determine the business question, and some models will always need human oversight before deployment. Therefore, the development process is primary and each module in the process should be automated as needed. This allows incremental build-out of automation and customization. Furthermore, when it comes to particular automation tools, choose those that align to your people and process. For example, instead of building a model logging framework around a generic database, you can choose a specialized tool like MLflow, which has been designed with the ML model lifecycle in mind.

CHAPTER 2:

Fundamentals of MLOps

Note: In our experience with customers, there can be variations in these three stages, such as splitting staging into separate "test" and "QA" substages. However, the principles remain the same and we stick to a dev, staging and prod setup within this paper.

Semantics of dev, staging and prod

ML workflows include the following key assets: code, models and data. These assets need to be developed (dev), tested (staging) and deployed (prod). For each stage, we also need to operate within an execution environment. Thus, all the above — execution environments, code, models and data — are divided into dev, staging and prod.

These divisions can best be understood in terms of quality guarantees and access control. On one end, assets in prod are generally business critical, with the highest guarantee of quality and tightest control on who can modify them. Conversely, dev assets are more widely accessible to people but offer no guarantee of quality.

For example, many data scientists will work together in a dev environment, freely producing dev model prototypes. Any flaws in these models are relatively low risk for the business, as they are separate from the live product. In contrast, the staging environment replicates the execution environment of production. Here, code changes made in the dev environment are tested prior to code being deployed to production. The staging environment acts as a gateway for code to reach production, and accordingly, fewer people are given access to staging. Code promoted to production is considered a live product. In the production environment, human error can pose the greatest risk to business continuity, and so the least number of people have permission to modify production models.

One might be tempted to say that code, models and data each share a one-to-one correspondence with the execution environment — e.g., all dev code, models and data are in the dev environment. That is often close to true but is rarely correct. Therefore, we will next discuss the precise semantics of dev, staging and prod for execution environments, code, models and data. We also discuss mechanisms for restricting access to each.

Execution environments

An execution environment is the place where models and data are created or consumed by code. Each execution environment consists of compute instances, their runtimes and libraries, and automated jobs. With Databricks, an "environment" can be defined via dev/staging/prod separation at a few levels. An organization could create distinct environments across multiple cloud accounts, multiple Databricks workspaces in the same cloud account, or within a single Databricks workspace. These separation patterns are illustrated in Figure 2 below.

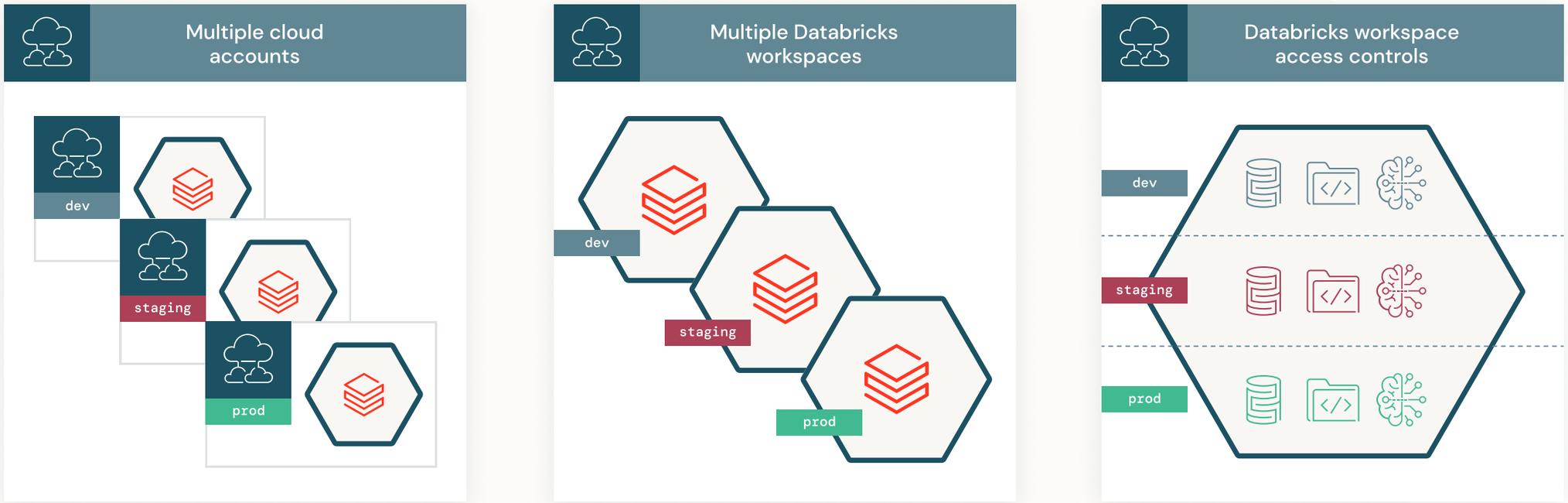


Figure 2

Databricks released Delta Lake to the open source community in 2019. Delta Lake provides all the data lifecycle management functions that are needed to make cloud-based object stores reliable and performant. This design allows clients to update multiple objects at once and to replace a subset of the objects with another, etc., in a serializable manner that still achieves high parallel read/write performance from the objects — while offering advanced capabilities like time travel (e.g., query point-in-time snapshots or rollback of erroneous updates), automatic data layout optimization, upserts, caching and audit logs.

Code

ML project code is often stored in a version control repository (such as Git), with most organizations using branches corresponding to the lifecycle phases of development, staging or production. There are a few common patterns. Some use only development branches (dev) and one main branch (staging/prod). Others use main and development branches (dev), branches cut for testing potential releases (staging), and branches cut for final releases (prod). Regardless of which convention you choose, separation is enforced through Git repository branches.

As a best practice, code should only be run in an execution environment that corresponds to it or in one that's higher. For example, the dev environment can run any code, but the prod environment can only run prod code.

Models

While models are usually marked as dev, staging or prod according to their lifecycle phase, **it is important to note that model and code lifecycle phases often operate asynchronously**. That is, you may want to push a new model version before you push a code change, and vice versa. Consider the following scenarios:

- To detect fraudulent transactions, you develop an ML pipeline that retrains a model weekly. Deploying the code can be a relatively infrequent process, but each week a new model undergoes its own lifecycle of being generated, tested and marked as "production" to predict on the most recent transactions. In this case the code lifecycle is slower than the model lifecycle.
- To classify documents using large deep neural networks, training and deploying the model is often a one-time process due to cost. Updates to the serving and monitoring code in the project may be deployed more frequently than a new version of the model. In this case the model lifecycle is slower than the code.

Since model lifecycles do not correspond one-to-one with code lifecycles, it makes sense for model management to have its own service. MLflow and its Model Registry support managing model artifacts directly via UI and APIs. The loose coupling of model artifacts and code provides flexibility to update production models without code changes, streamlining the deployment process in many cases. Model artifacts are secured using MLflow access controls or cloud storage permissions.

Data

Some organizations label data as either dev, staging or prod, depending on which environment it originated in. For example, all prod data is produced in the prod environment, but dev and staging environments may have read-only access to them. Marking data this way also indicates a guarantee of data quality: dev data may be temporary or not meant for wider use, whereas prod data may offer stronger guarantees around reliability and freshness. Access to data in each environment is controlled with table access controls (AWS | Azure | GCP) or cloud storage permissions.

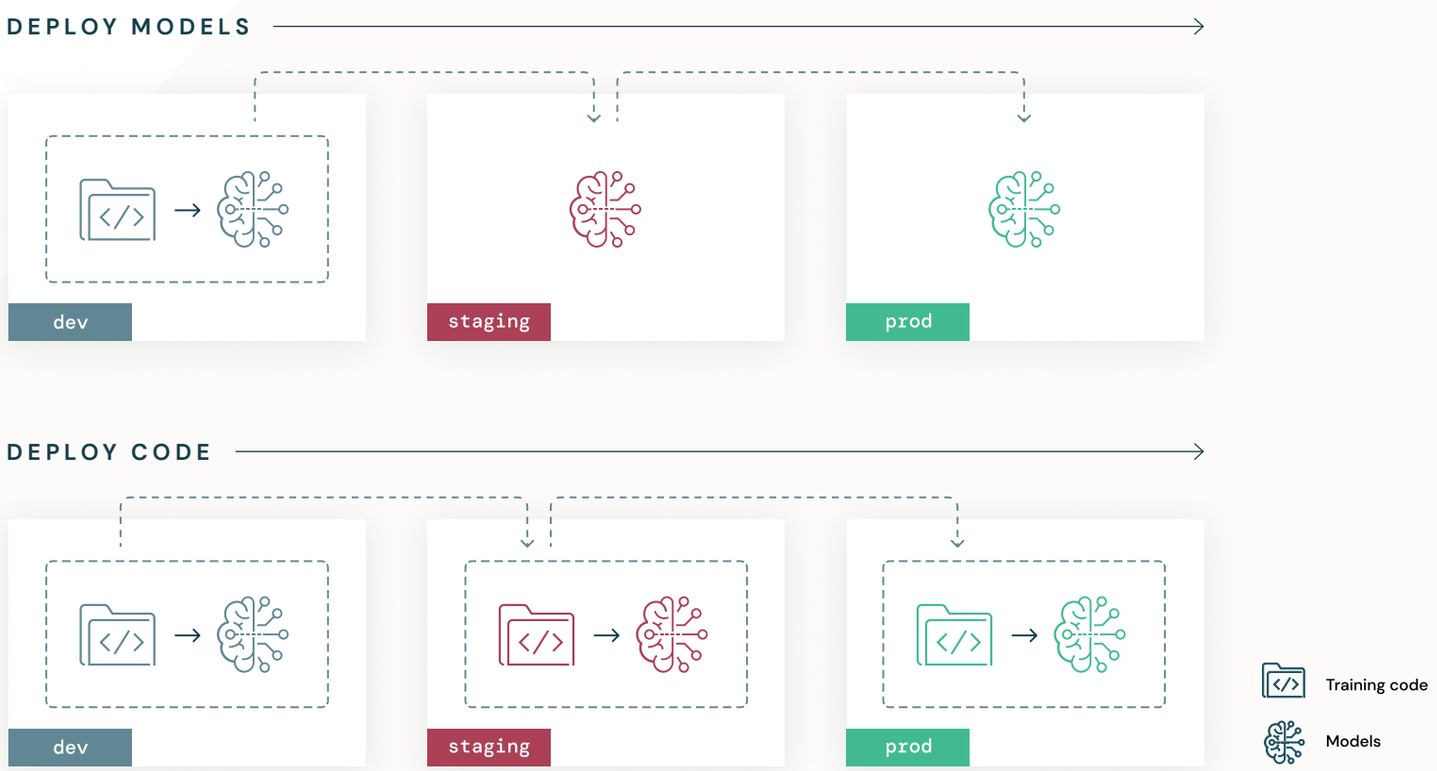
In summary, when it comes to MLOps, you will always have operational separation between dev, staging and prod. Assets in dev will have the least restrictive access controls and quality guarantees, while those in prod will be the highest quality and tightly controlled.

ASSET	SEMANTICS	SEPARATED BY
Execution environments	Labeled according to where development, testing and connections with production systems happen	Cloud provider and Databricks Workspace access controls
Models	Labeled according to model lifecycle phase	MLflow access controls or cloud storage permissions
Data	Labeled according to its origin in dev, staging or prod execution environments	Table access controls or cloud storage permissions
Code	Labeled according to software development lifecycle phase	Git repository branches

Table 1

ML deployment patterns

The fact that models and code can be managed separately results in multiple possible patterns for getting ML artifacts through staging and into production. We explain two major patterns below.



These two patterns differ in terms of whether the model artifact or the training code that produces the model artifact is promoted toward production.

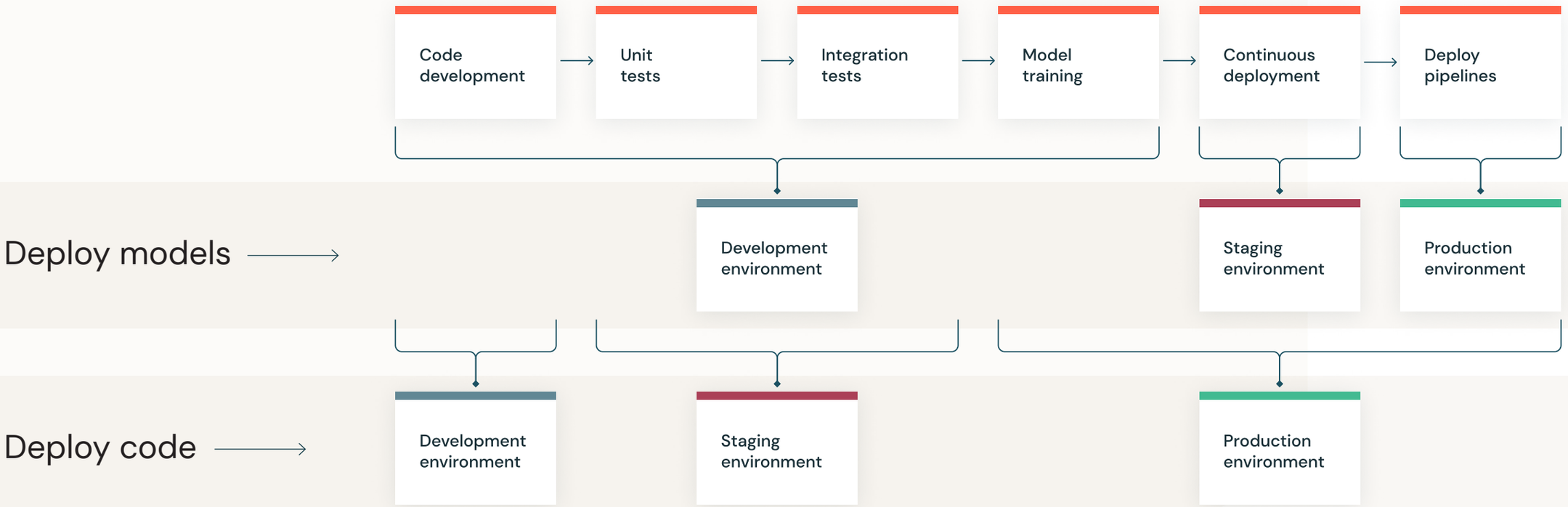
Deploy models

In the first pattern, the model artifact is generated by training code in the development environment. This artifact is then tested in staging for compliance and performance before finally being deployed into production. This is a simpler handoff for data scientists, and in cases where model training is prohibitively expensive, training the model once and managing that artifact may be preferable. However, this simpler architecture comes with limitations. If production data is not accessible from the development environment (e.g., for security reasons), this architecture may not be viable. This architecture does not naturally support automated model retraining. While you could automate retraining in the development environment, you would then be treating "dev" training code as production ready, which many deployment teams would not accept. This option hides the fact that ancillary code for featurization, inference and monitoring needs to be deployed to production, requiring a separate code deployment path.

Deploy code

In the second pattern, the code to train models is developed in the dev environment, and this code is moved to staging and then production. Models will be trained in each environment: initially in the dev environment as part of model development, in staging (on a limited subset of data) as part of integration tests, and finally in the production environment (on the full production data) to produce the final model. If an organization restricts data scientists' access to production data from dev or staging environments, deploying code allows training on production data while respecting access controls. Since training code goes through code review and testing, it is safer to set up automated retraining. Ancillary code follows the same pattern as model training code, and both can go through integration tests in staging. However, the learning curve for handing code off to collaborators can be steep for many data scientists, so opinionated project templates and workflows are helpful. Finally, data scientists need visibility into training results from the production environment, for only they have the knowledge to identify and fix ML-specific issues.

The diagram below contrasts the code lifecycle for the above deployment patterns across the different execution environments.



In general we recommend following the "deploy code" approach, and the reference architecture in this document is aligned to it. Nevertheless, there is no perfect process that covers every scenario, and the options outlined above are not mutually exclusive. Within a single organization, you may find some use cases deploying training code and others deploying model artifacts. Your choice of process will depend on the business use case, resources available and what is most likely to succeed.

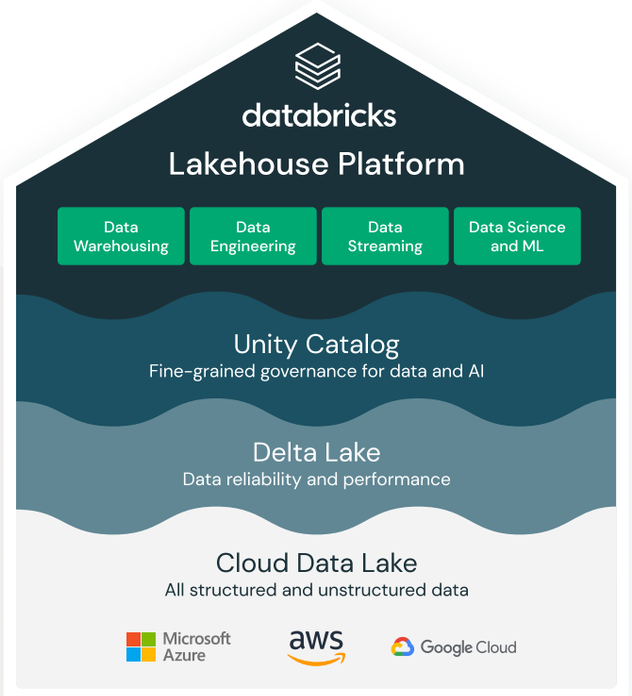
		DEPLOY MODELS	DEPLOY CODE
Process	**Dev**	Develop training code. Develop ancillary code.[1] Train model on prod data. → Promote model and ancillary code.	Develop training code. Develop ancillary code. → Promote code.
	Staging	Test model and ancillary code. → Promote model and ancillary code.	Train model on data subset. Test ancillary code. → Promote code.
	Prod	Deploy model. Deploy ancillary pipelines.	Train model on prod data. Test model. Deploy model. Deploy ancillary pipelines.
Trade-offs	**Automation**	↓ Does not support automated retraining in locked-down env.	↑ Supports automated retraining in locked-down env.
	Data access control	↓ Dev env needs read access to prod training data.	↑ Only prod env needs read access to prod training data.
	Reproducible models	↓ Less eng control over training env, so harder to ensure reproducibility.	↑ Eng control over training env, which helps to simplify reproducibility.
	Data science familiarity	↑ DS team builds and can directly test models in their dev env.	↓ DS team must learn to write and hand off modular code to eng.
	Support for large projects	↓ This pattern does not force the DS team to use modular code for model training, and it has less iterative testing.	↑ This pattern forces the DS team to use modular code and iterative testing, which helps with coordination and development in larger projects.
	Eng setup and maintenance	↑ Has the simplest setup, with less CI/CD infra required.	↓ Requires CI/CD infra for unit and integration tests, even for one-off models.
When to use		Use this pattern when your model is a one-off or when model training is very expensive. Use when dev, staging and prod are not strictly separated envs.	**Use this pattern by default.** Use when dev, staging and prod are strictly separated envs.

Table 2

[1] "Ancillary code" refers to code for ML pipelines other than the model training pipeline. Ancillary code could be featurization, inference, monitoring or other pipelines.

CHAPTER 3:

MLOps Architecture and Process



Architecture components

Before unpacking the reference architecture, take a moment to familiarize yourself with the Databricks features used to facilitate MLOps in the workflow prescribed.

Data Lakehouse

A Data Lakehouse architecture unifies the best elements of data lakes and data warehouses — delivering data management and performance typically found in data warehouses with the low-cost, flexible object stores offered by data lakes. Data in the lakehouse are typically organized using a "medallion" architecture of Bronze, Silver and Gold tables of increasing refinement and quality.

MLflow

MLflow is an open source project for managing the end-to-end machine learning lifecycle. It has the following primary components:

- **Tracking:** Allows you to track experiments to record and compare parameters, metrics and model artifacts. See documentation for AWS | Azure | GCP.
- **Models ("MLflow flavors"):** Allows you to store and deploy models from any ML library to a variety of model serving and inference platforms. See documentation for AWS | Azure | GCP.
- **Model Registry:** Provides a centralized model store for managing models' full lifecycle stage transitions: from staging to production, with capabilities for versioning and annotating. The registry also provides webhooks for automation and continuous deployment. See documentation for AWS | Azure | GCP.

Databricks also provides a fully managed and hosted version of MLflow with enterprise security features, high availability, and other Databricks workspace features such as experiment and run management and notebook revision capture. MLflow on Databricks offers an integrated experience for tracking and securing machine learning model training runs and running machine learning projects.

Databricks and MLflow Autologging

Databricks Autologging is a no-code solution that extends MLflow automatic logging to deliver automatic experiment tracking for machine learning training sessions on Databricks. Databricks Autologging automatically captures model parameters, metrics, files and lineage information when you train models with training runs recorded as MLflow tracking runs. See documentation for AWS | Azure | GCP.

Feature Store

The Databricks Feature Store is a centralized repository of features. It enables feature sharing and discovery across an organization and also ensures that the same feature computation code is used for model training and inference. See documentation for AWS | Azure | GCP.

MLflow Model Serving

MLflow Model Serving allows you to host machine learning models from Model Registry as REST endpoints that are updated automatically based on the availability of model versions and their stages. See documentation for AWS | Azure | GCP.

Databricks SQL

Databricks SQL provides a simple experience for SQL users who want to run quick ad hoc queries on their data lake, create multiple visualization types to explore query results from different perspectives, and build and share dashboards. See documentation for AWS | Azure | GCP.

Databricks Workflows and Jobs

Databricks Workflows (Jobs and Delta Live Tables) can execute pipelines in automated, non-interactive ways. For ML, Jobs can be used to define pipelines for computing features, training models, or other ML steps or pipelines. See documentation for AWS | Azure | GCP.

Reference architecture

We are now ready to review a general reference architecture for implementing MLOps on the Databricks Lakehouse platform using the recommended "deploy code" pattern from earlier. This is intended to cover the majority of use cases and ML techniques, but it is by no means comprehensive. When appropriate, we will highlight alternative approaches to implementing different parts of the process.

We begin with an overview of the system end-to-end, followed by more detailed views of the process in development, staging and production environments. These diagrams show the system as it operates in a steady state, with the finer details of iterative development cycles omitted. This structure is summarized below.

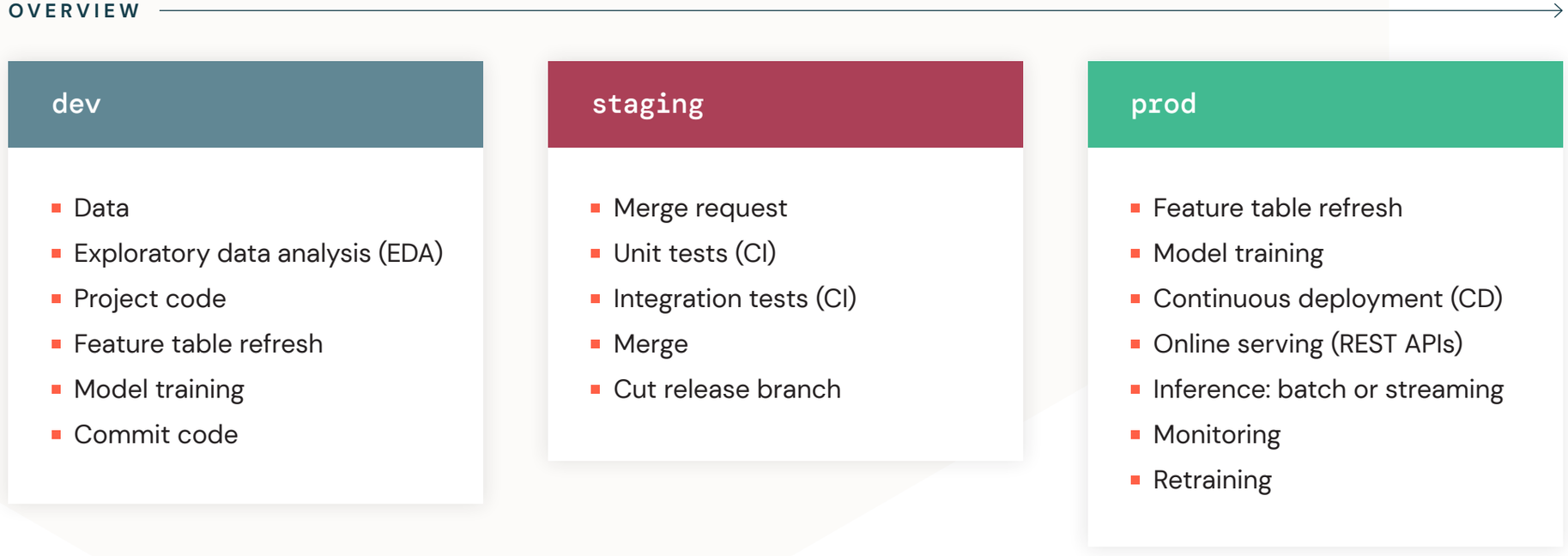
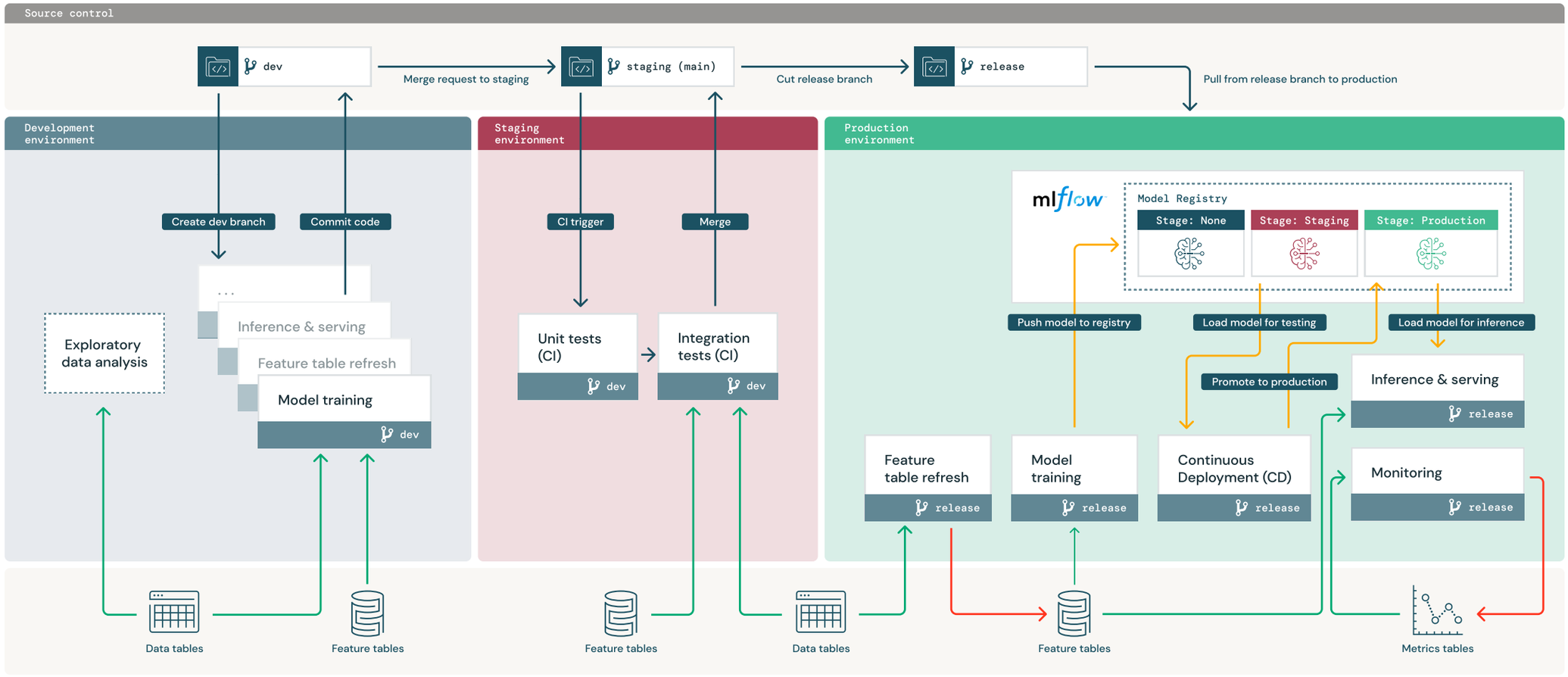


Figure 3

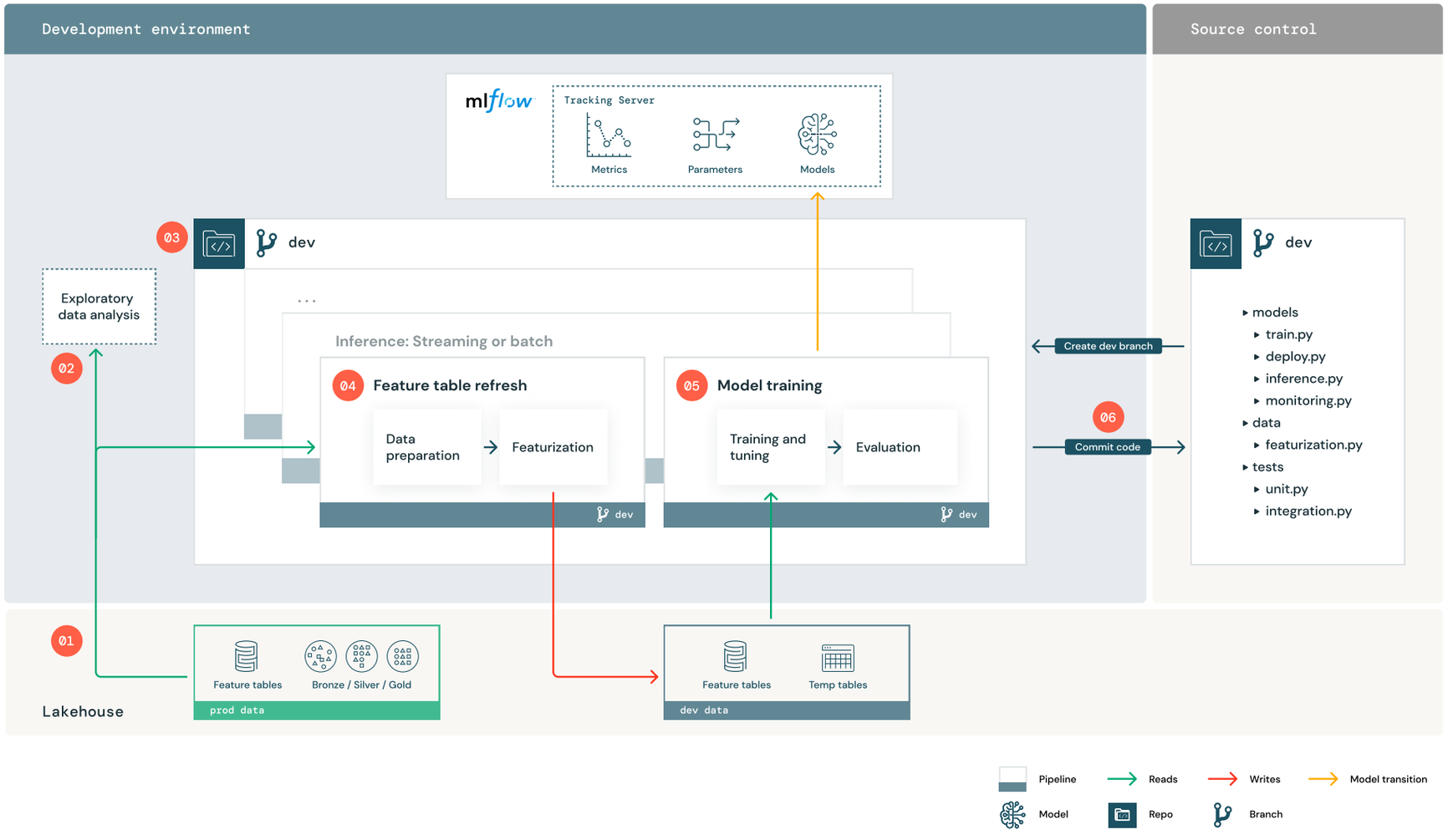


Here we see the overall process for deploying code and model artifacts, the inputs and outputs for pipelines, and model lifecycle stages in production. Code source control is the primary conduit for deploying ML pipelines from development to production. Pipelines and models are prototyped on a dev branch in the development environment, and changes to the codebase are committed back to source control. Upon merge request to the staging branch (usually the "main" branch), a continuous integration (CI) process tests the code in the staging environment. If the tests pass, new code can be deployed to production by cutting a code release. In production, a model is trained on the full production data and pushed to the MLflow Model Registry. A continuous deployment (CD) process tests the model and promotes it toward the production stage in the registry. The Model Registry's production model can be served via batch, streaming or REST API. Ongoing feature engineering and monitoring pipelines also run in production.

Dev

In the development environment, data scientists and ML engineers can collaborate on all pipelines in an ML project, committing their changes to source control. While engineers may help to configure this environment, data scientists typically have significant control over the libraries, compute resources and code that they use.

Figure 4



Data

Data scientists working in the dev environment possess read-only access to production data. They also require read-write access to a separate dev storage environment to develop and experiment with new features and other data tables.

Exploratory data analysis (EDA)

The data scientist explores and analyzes data in an interactive, iterative process. This process is used to assess whether the available data has the potential to address the business problem. EDA is also where the data scientist will begin discerning what data preparation and featurization are required for model training.

This ad hoc process is generally not part of a pipeline that will be deployed in other execution environments.

Project code

This is a code repository containing all of the pipelines or modules involved in the ML system. Dev branches are used to develop changes to existing pipelines or to create new ones. Even during EDA and initial phases of a project, it is recommended to develop within a repository to help with tracking changes and sharing code.

Feature table refresh

This pipeline reads from raw data tables and feature tables and writes to tables in the Feature Store. The pipeline consists of two steps:

- **Data preparation**
  This step checks for and corrects any data quality issues prior to featurization.
- **Featurization**
  In the dev environment, new features and updated featurization logic can be tested by writing to feature tables in dev storage, and these dev feature tables can be used for model prototyping. Once this featurization code is promoted to production, these changes will affect the production feature tables. Features already available in production feature tables can be read directly for development.

In some organizations, feature engineering pipelines are managed separately from ML projects. In such cases, the featurization pipeline can be omitted from this architecture.

Model training

Data scientists develop the model training pipeline in the dev environment with dev or prod feature tables.

- **Training and tuning**
  The training process reads features from the feature store and/or Silver- or Gold-level Lakehouse tables, and it logs model parameters, metrics and artifacts to the MLflow tracking server. After training and hyperparameter tuning, 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.
  If governance requires additional metrics or supplemental documentation about the model, this is the time to add them using MLflow tracking. Model interpretations (e.g., plots produced by SHAP or LIME) and plain text descriptions are common, but defining the specifics for such governance requires input from business stakeholders or a data governance officer.

- **Model output**
  The output of this pipeline is an ML model artifact stored in the MLflow tracking server. When this training pipeline is run in staging or production, ML engineers (or their CI/CD code) can load the model via the model URI (or path) and then push the model to the Model Registry for management and testing.

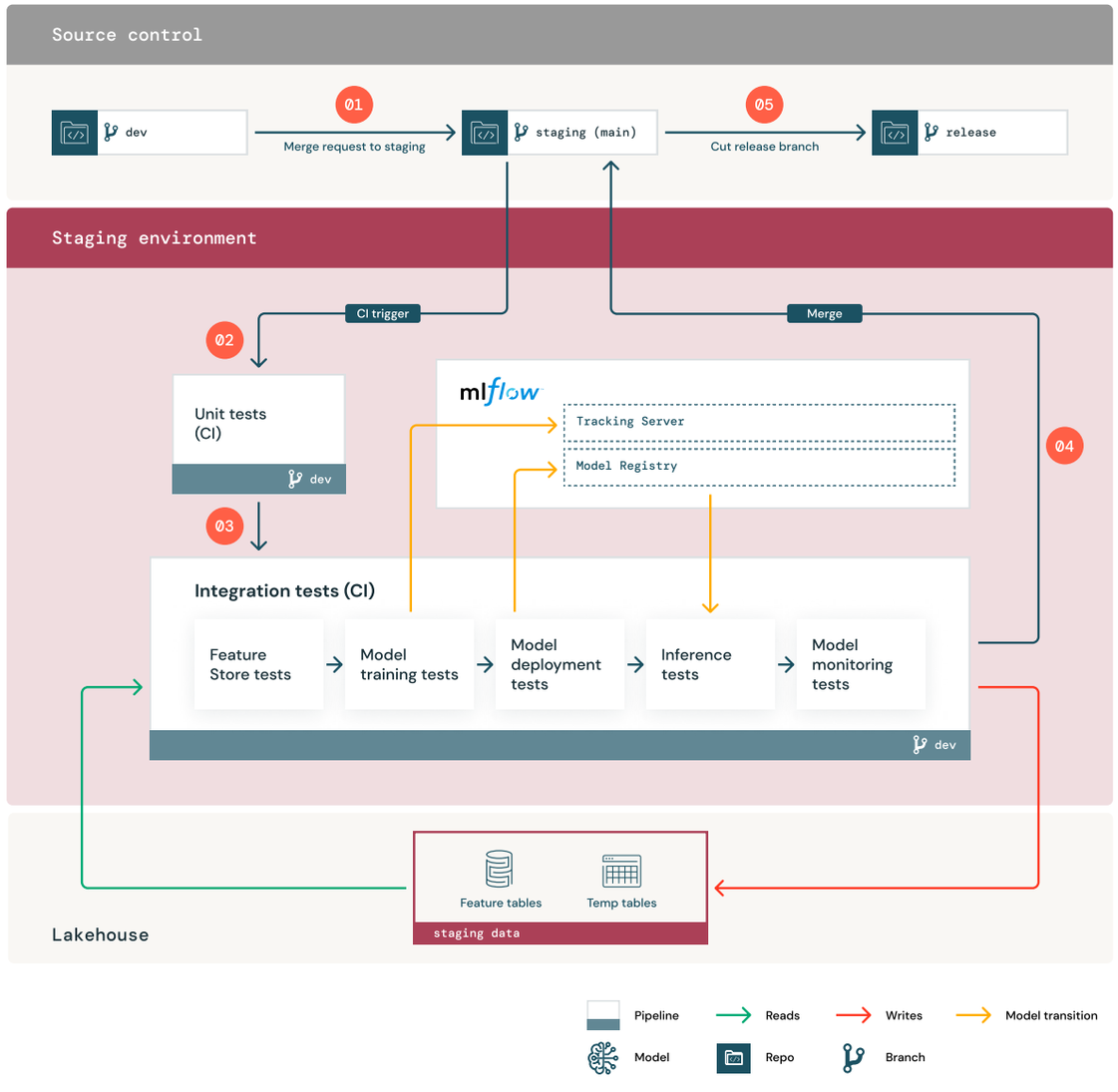
Commit code

After developing code for featurization, training, inference and other pipelines, the data scientist or ML engineer commits the dev branch changes into source control. This section does not discuss the continuous deployment, inference or monitoring pipelines in detail; see the "Prod" section below for more information on those.

Staging

The transition of code from development to production occurs in the staging environment. This code includes model training code and ancillary code for featurization, inference, etc. Both data scientists and ML engineers are responsible for writing tests for code and models, but ML engineers manage the continuous integration pipelines and orchestration.

Figure 5



Data

The staging environment may have its own storage area for testing feature tables and ML pipelines. This data is generally temporary and only retained long enough to run tests and to investigate test failures. This data can be made readable from the development environment for debugging.

Merge code

- **Merge request**
  The deployment process begins when a merge (or pull) request is submitted against the staging branch of the project in source control. It is common to use the "main" branch as the staging branch.

- **Unit tests (CI)**
  This merge request automatically builds source code and triggers unit tests. If tests fail, the merge request is rejected.

Integration tests (CI)

The merge request then goes through integration tests, which run all pipelines to confirm that they function correctly together. The staging environment should mimic the production environment as much as is reasonable, running and testing pipelines for featurization, model training, inference and monitoring.

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 model training on small data sets or for fewer iterations to reduce cost. When models are deployed behind REST APIs, some high-SLA models may merit full-scale load testing within these integration tests, whereas other models may be tested with small batch jobs or a few queries to temporary REST endpoints.

Once integration tests pass on the staging branch, the code may be promoted toward production.

- **Merge**
  If all tests pass, the new code is merged into the staging branch of the project. If tests fail, the CI/CD system should notify users and post results on the merge (pull) request.

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

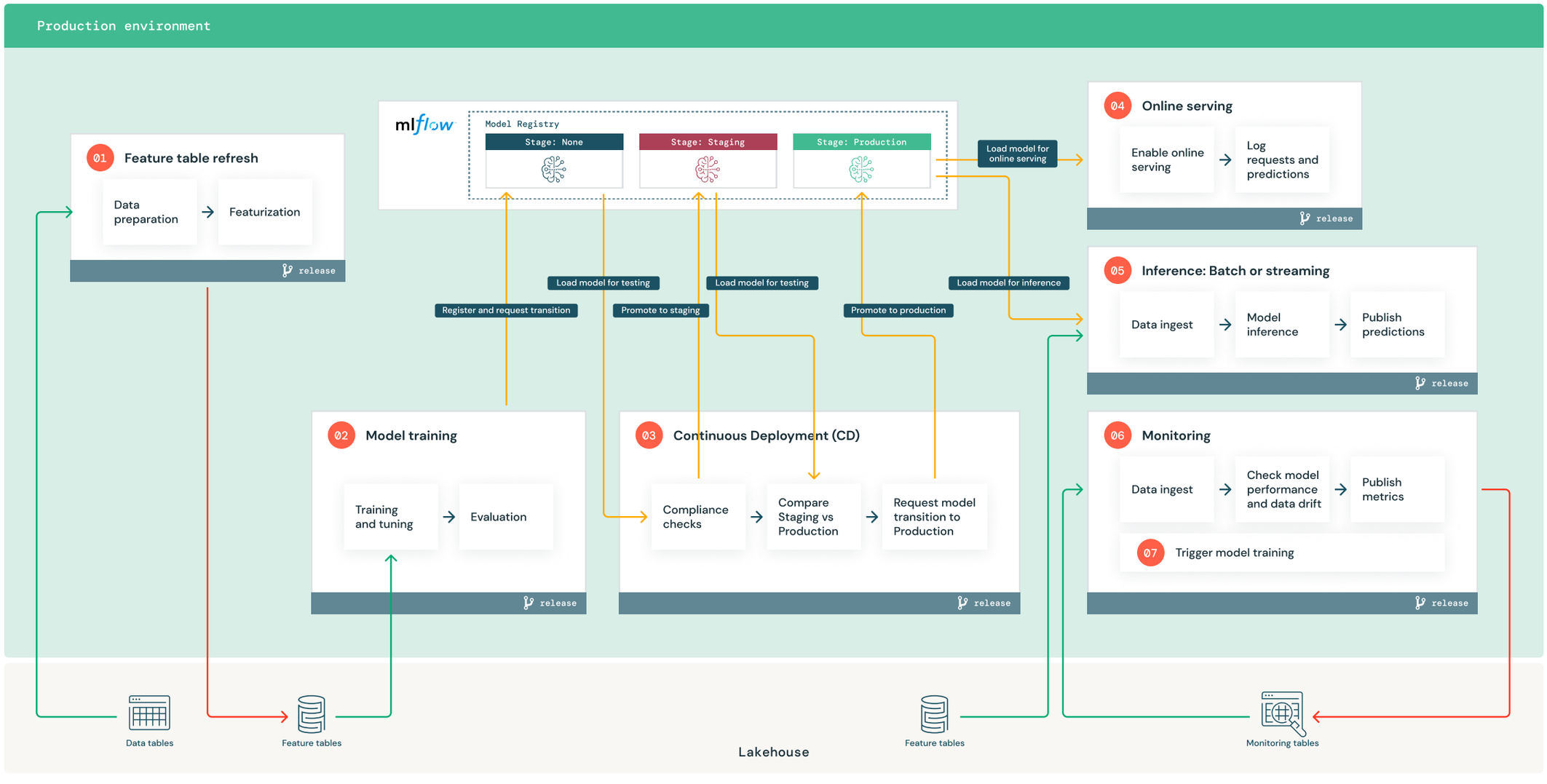
Cut release branch

Once CI tests have passed on a commit in the staging branch, ML engineers can cut a release branch from that commit.

Prod

The production environment is typically managed by a select set of ML engineers and is where ML pipelines directly serve the business or application. These pipelines compute fresh feature values, train and test 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 online serving below, most ML applications will use only one of these methods, depending on the business requirements.

Figure 6



Though data scientists may not have write or compute access in the production environment, 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.

Feature table refresh

This pipeline transforms the latest production Lakehouse data into production feature tables. It can use batch or streaming computation, depending on the freshness requirements for downstream training and inference. The pipeline can be defined as a Databricks Job which is scheduled, triggered or continuously running.

Model training

The model training pipeline runs either when code changes affect upstream featurization or training logic, or when automated retraining is scheduled or triggered. This pipeline runs on the full production data.

- **Training and tuning**

  During the training process, logs are recorded to the MLflow tracking server. These include model metrics, parameters, tags and the model itself.

  During development, 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 in automated retraining, and it can make training and tuning faster.

- **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 will have selected meaningful evaluation metrics for the use case, and those metrics or their custom logic will be used in this step.

- **Register and request transition**

  Following model training, the model artifact is registered to the MLflow Model Registry of the production environment, set initially to 'stage=None'. The final step of this pipeline is to request a transition of the newly registered model to 'stage=Staging'.

Continuous deployment (CD)

The CD pipeline is executed when the training pipeline finishes and requests to transition the model to 'stage=Staging'. There are three key tasks in this pipeline:

- **Compliance checks**

  These tests load the model from the Model Registry, perform compliance checks (for tags, documentation, etc.), and approve or reject the request based on test results. If compliance checks require human expertise, this automated step can compute statistics or visualizations for people to review in a manual approval step at the end of the CD pipeline. Regardless of the outcome, results for that model version are recorded to the Model Registry through metadata in tags and comments in descriptions.

  The MLflow UI can be used to manage stage transition requests manually, but requests and transitions can be automated via MLflow APIs and webhooks. If the model passes the compliance checks, then the transition request is approved and the model is promoted to 'stage=Staging'. If the model fails, the transition request is rejected and the model is moved to 'stage=Archived' in the Model Registry.

- **Compare staging vs. production**

  To prevent performance degradation, models promoted to 'stage=Staging' must be compared to the 'stage=Production' models they are meant to replace. The metric(s) for comparison should be defined according to the use case, and the method for comparison can vary from canary deployments to A/B tests. All comparison results are saved to metrics tables in the lakehouse.

  If this is the first deployment and there is no 'stage=Production' model yet, the 'stage=Staging' model should be compared to a business heuristic or other threshold as a baseline. For a new version of an existing 'stage=Production' model, the 'stage=Staging' model is compared with the current 'stage=Production' model.

- **Request model transition to production**
  If the candidate model passes the comparison tests, a request is made to transition it to 'stage=Production' in the Model Registry. As with other stage transition requests, notifications, approvals and rejections can be managed manually via the MLflow UI or automatically through APIs and webhooks. This is also a good time to consider human oversight, as it is the last step before a model is fully available to downstream applications. A person can manually review the compliance checks and performance comparisons to perform checks which are difficult to automate.

Online serving (REST APIs)

For lower throughput and lower latency use cases, online serving is generally necessary. With MLflow, it is simple to deploy models to Databricks Model Serving, cloud provider serving endpoints, or on-prem or custom serving layers.

In all cases, the serving system loads the production model from the Model Registry upon initialization. On each request, it fetches features from an online Feature Store, scores the data and returns predictions. The serving system, data transport layer or the model itself could log requests and predictions.

Inference: batch or streaming

This pipeline is responsible for reading the latest data from the Feature Store, loading the model from 'stage=Production' in the Model Registry, performing inference and publishing predictions. For higher throughput, higher latency use cases, batch or streaming inference is generally the most cost-effective option.

A batch job would likely publish predictions to Lakehouse tables, over a JDBC connection, or to flat files. A streaming job would likely publish predictions either to Lakehouse tables or to message queues like Apache Kafka®

Monitoring

Input data and model predictions are monitored, both for statistical properties (data drift, model performance, etc.) and for computational performance (errors, throughput, etc.). 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.
- **Publish metrics**
  The pipeline writes to Lakehouse tables for analysis and reporting. 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.

Retraining

This architecture supports automatic retraining using the same model training pipeline above. While we recommend beginning with manually triggered retraining, organizations can add scheduled and/or triggered retraining when needed.

- **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.

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

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?

CHAPTER 4:

LLMOps – Large Language Model Operations

Large language models

LLMs have splashed into the mainstream of business and news, and there is no doubt that they will 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 explain what may change for MLOps when introducing LLMs. We will discuss several key topics in detail, from prompt engineering to packaging, to 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. The one-sentence summary is: LLMs are a new class of natural language processing (NLP) models that have significantly surpassed their predecessors in 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 delve into more detail in the next section.

Table 3

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 model registries 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 data refresh, model tuning, model inference, etc.

Discussion of key topics for LLMOps

So far, we have listed top potential changes to MLOps as you introduce LLMs. In this section, we will dive into more details about selected topics.

Prompt engineering

Prompt engineering is the practice of adjusting the text prompt given to an LLM in order to elicit better responses — using engineering techniques. It is a very new practice, but some best practices are emerging. We will cover a few 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 the extreme case, many LLMs have been fine-tuned for specific NLP tasks and do not even 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 Demonstrate-Search-Predict (DSP) Framework is a good example of a tool for prompt tuning.

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

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, specific 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
- If 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

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's 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.

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. In the future, you can use that 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.

Resources

Fine-tuning

→ Fine-Tuning Large Language Models with Hugging Face and DeepSpeed

→ Webinar: Build Your Own Large Language Model Like Dolly: How to fine-tune and deploy your custom LLM

Model distillation, quantization and pruning

→ Gentle Introduction to 8-bit Matrix Multiplication for transformers at scale using Hugging Face Transformers, Accelerate and bitsandbytes

→ Large Transformer Model Inference Optimization

→ Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA

Methods for reducing costs of inference

Use a smaller model

- Pick a different existing model. Try smaller versions of models (such as "t5-small" instead of "t5-base") or alternate architectures.
- Fine-tune a custom model. With the right training data, a fine-tuned model can often be smaller and/or perform better than a 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, so using more concise queries and responses reduces costs.
- 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 low ROI queries are handled by simpler, faster models or methods. 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.

Resources

Reinforcement Learning from Human Feedback (RLHF)

→ Chip Huyen blog post on "RLHF: Reinforcement Learning from Human Feedback"

→ Hugging Face blog post on "Illustrating Reinforcement Learning from Human Feedback (RLHF)"

→ Wikipedia

Human feedback, testing, and monitoring

While human feedback is important in many traditional ML applications, it becomes much more important for LLMs. Since most LLMs output natural language, it is very difficult to evaluate the outputs via traditional metrics. 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.

Humans — ideally your end users — become essential for validating LLM output. While you can pay human labelers to compare or rate model outputs, 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.

In terms of operations, not much changes from traditional MLOps:

- **Data:** Human feedback is simply data, and it should be treated like any other data. Store it in your lakehouse, and process it using the same data pipeline tooling as other data.
- **Testing and monitoring:** A/B testing and incremental rollouts of new models and pipelines may become more important, superceding offline quality tests. If you can collect user feedback, then these rollout methods can validate models before they are fully deployed.
- **Fine-tuning:** Human feedback becomes especially important for LLMs when it can be incorporated into fine-tuning models via techniques like Reinforcement Learning from Human Feedback (RLHF). Even if you start with an existing or generic model, you can eventually customize it for your purposes via fine-tuning.

Other topics

- **Scaling out:** Practices around scaling out training, fine-tuning and inference are similar to traditional ML, but some of your tools may change. Tools like Apache Spark™ and Delta Lake remain general enough for your LLM data pipelines and for batch and streaming inference, and they may be helpful for distributing fine-tuning. To handle LLM fine-tuning and training, you may need to adopt some new tools such as distributed PyTorch, distributed TensorFlow, and DeepSpeed.
- **Model serving:** If you manage the serving system for your LLMs, then you may need to make adjustments to handle larger models. While serving with CPUs can work for smaller deep learning models, most LLMs will benefit from or require GPUs for serving and inference.
- **Vector databases:** Some but not all LLM applications require vector databases for efficient similarity-based lookups of documents or other data. Vector databases may be an important addition to your serving infrastructure. Operationally, it is analogous to a feature store: it is a specialized tool for storing preprocessed data which can be queried by inference jobs or model serving systems.

Reference architecture

To illustrate potential adjustments to your reference architecture from traditional MLOps, we provide a modified version of the previous production architecture.

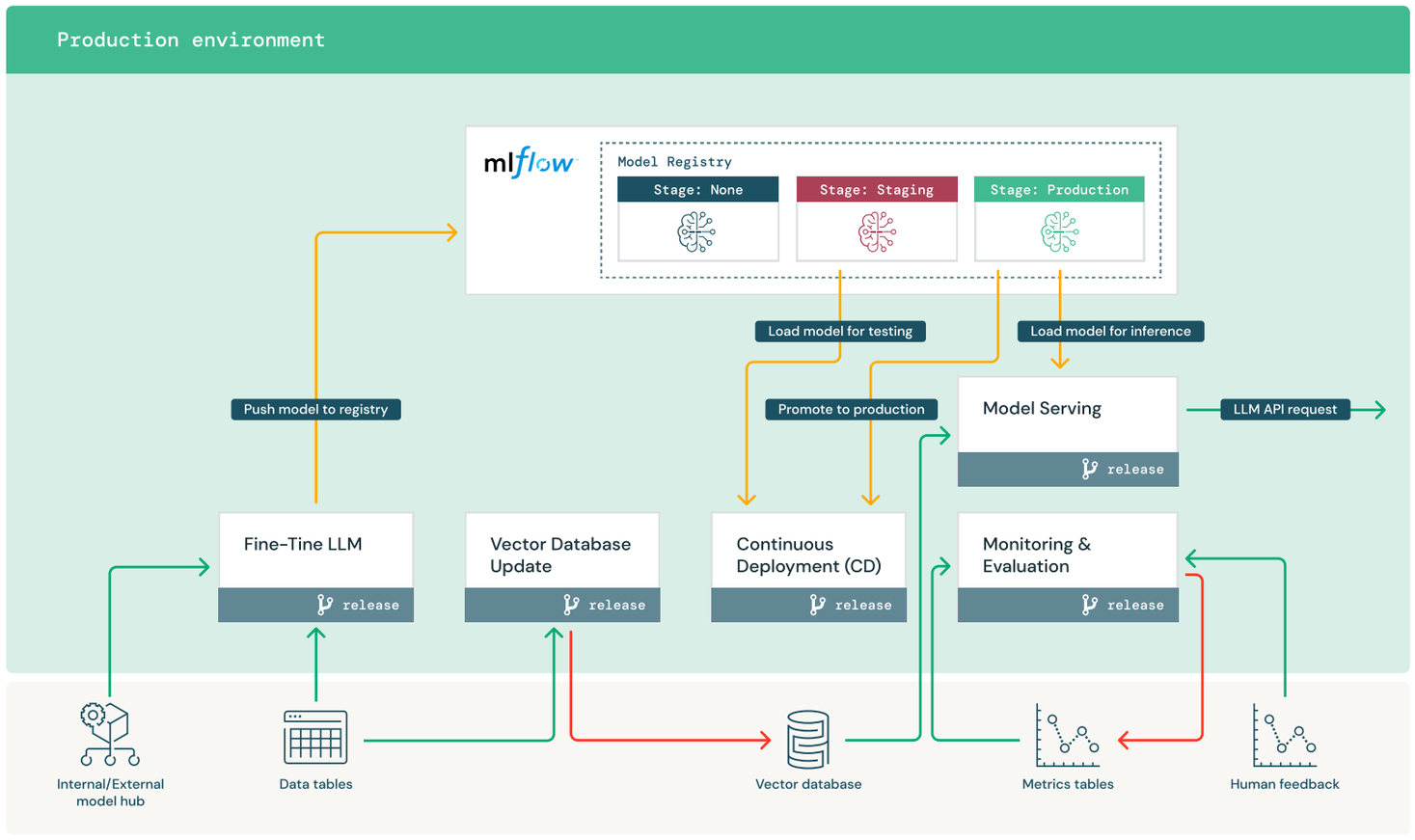


Figure 7

Additional resources

With LLMs being such a novel field, we link to several LLM resources below, which are not necessarily "LLMOps" but may prove useful to you.

→ edX: Professional Certificate in Large Language Models

→ Chip Huyen blog post on "Building LLM applications for production"

LLM lists and leaderboards

→ LMSYS Leaderboard

→ Hugging Face Open LLM Leaderboard

→ Stanford Center for Research on Foundation Models
  → Ecosystem graphs
  → HELM

→ Blog post on "Open Source ChatGPT Alternatives"

The primary changes to this production architecture are:

- **Internal/External 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. It appears here in production to illustrate using an existing base model that is then fine-tuned in production. Without fine-tuning, this hub would mainly be used in development.
- **Fine-Tune LLM:** Instead of de novo Model Training, 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.
- **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. We replaced the Feature Store (and its Feature Table Refresh job) with the Vector Database (and its Vector Database Update job) to illustrate that these data stores and jobs are analogous in terms of operations.
- **Model Serving:** The architectural change illustrated here is that some LLM pipelines will make external API calls, such as to internal or third-party LLM APIs. Operationally, this adds complexity in terms of potential latency or flakiness from third-party APIs, as well as another layer of credential management.
- **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.

Looking ahead

LLMs only became mainstream in late 2022, and countless libraries and technologies are being built to support and leverage LLM use cases. You should expect rapid changes. More powerful LLMs will be open-sourced; tools and techniques for customizing LLMs and LLM pipelines will become more plentiful and flexible; and an explosion of techniques and ideas will gradually coalesce into more standardized practices.

While this technological leap provides us all with great opportunities, the use of cutting-edge technologies requires extra care in LLMOps to build and maintain stable, reliable LLM-powered applications. The good news is that much of your existing MLOps tooling, practices and knowledge will transfer smoothly over to LLMs. With the additional tips and practices mentioned in this section, you should be well set up to harness the power of large language models.

About Databricks

Databricks is the data and AI company. More than 9,000 organizations worldwide — including Comcast, Condé Nast and over 50% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

Sign up for a free trial

databricks