MLOps Reference Handbook

A source-grounded guide structured from The Big Book of MLOps

Asjad

Contents

Introduction

This handbook is a working reference for building and operating machine learning systems on Databricks. Its structure follows the table of contents of *The Big Book of MLOps, 2nd Edition*, using that book as the organizing backbone. The short PDF, `MLOps Questions (1).pdf`, and the archived web sources in `sources/` enrich the relevant sections with Databricks-specific implementation detail.

The book is intended to be a compact operating manual: concepts first, then repeatable processes.

Key Topics

This handbook focuses on the operational controls that make machine learning reproducible and promotable. MLflow provides the tracking layer for experiments, parameters, metrics, artifacts, model signatures, input examples, and model packages. Reproducibility comes from connecting each run to the code version, data snapshot, execution environment, model configuration, and evaluation evidence that produced it.

Model versioning turns model artifacts into governed assets. In Databricks, Unity Catalog registered models provide immutable versions, metadata, access control, lineage, and aliases such as `@champion` and `@challenger`. Git remains the source of truth for pipeline code, validation logic, deployment configuration, and infrastructure definitions. The preferred promotion workflow is therefore code-first: move reviewed code from development to staging to production, then train, validate, register, and deploy the model inside the target environment.

Execution environments are the boundary that makes this workflow trustworthy. Development should support exploration, staging should prove that the full pipeline works under production-like constraints, and production should run approved jobs with controlled permissions, observability, and rollback paths.

Source grounding

The local knowledgebase for this handbook is:

- The Big Book of MLOps, 2nd Edition
- MLOps Questions (1).pdf
- Local source manifest, containing 22 archived sources extracted from the short PDF
- Archived source snapshots under sources/

Substantive sections include “Sources used” notes. These notes point to local files, so a reader can trace the claim back into the available knowledgebase.

How to read this handbook

Read the main chapters in order if you want the operating model: MLOps foundations, platform changes, design decisions, reference architecture, implementation, LLMOps, and conclusion. Use the recipes when you need a repeatable process for a concrete scenario, such

as registering a model, building CI/CD, rolling out a challenger, monitoring production, or operating a RAG system.

MLOps = DataOps + DevOps + ModelOps

MLOps is the discipline of making machine learning reliable as software and trustworthy as a data product. The Big Book frames MLOps as the combination of DataOps, DevOps, and ModelOps. That framing is useful because production ML fails at the boundaries: a model may be statistically sound but trained on stale data, deployed by unreviewed code, registered without lineage, or monitored only after business users notice a decline.

The practical goal is to create a system where data, code, models, execution environments, and monitoring all move through controlled lifecycles. The system should answer ordinary production questions without heroics:

- Which code trained this model?
- Which data snapshot did it use?
- Which validation checks passed before promotion?
- Which model version is serving traffic now?
- How do we roll back?
- How do we know the model is still performing?
- Who approved the promotion, and where is that decision recorded?

The handbook uses one consistent operating principle: do not make machine learning production readiness depend on memory, notebooks, or manual handoffs. Encode conventions into experiments, registries, bundles, CI/CD workflows, monitors, and templates.

Sources used: Big Book pp. 5-6; MLflow on Databricks; Manage model lifecycle in Unity Catalog.

Foundations

Why should I care about MLOps?

MLOps matters because ML systems degrade in more ways than normal software. Code can break, but so can data contracts, feature distributions, labels, model behavior, serving latency, monitoring assumptions, and governance metadata. A model that was correct at training time can become unreliable in production without a code change.

The Big Book's central claim is that effective MLOps accelerates time to business value while reducing ongoing operational burden. That is not because MLOps adds process for its own sake. It is because repeatable promotion, validation, deployment, and monitoring let teams move faster without losing control.

The AWS Databricks workflow documentation makes the operating scope explicit: MLOps is the automated management of code, data, and models so ML systems remain performant, stable, and efficient over time. That definition is useful because it prevents a narrow reading of MLOps as "model deployment." A production ML system is governed by the interaction between source code, data assets, model versions, execution environments, validation evidence, monitoring signals, and retraining decisions.

A mature MLOps workflow gives data scientists room to experiment, gives ML engineers a tested path to production, and gives platform or governance teams a way to enforce lineage, access control, and auditability without turning every release into a manual negotiation.

Guiding principles

The Big Book emphasizes a data-centric view of machine learning. In practice, most ML work is a set of data pipelines: feature engineering, training, validation, inference, monitoring, and retraining all consume and produce data. Treating those pipelines as first-class production assets is more reliable than treating the model artifact as the only important output.

Three principles follow:

1. **Unify data and ML governance.** Data, features, runs, model versions, inference logs, and monitoring tables should live in a governance layer where permissions and lineage are visible.
2. **Promote code through environments.** Training and deployment logic should move through review and CI/CD. Production models should be produced by production-approved code, not manually carried from development.
3. **Close the loop.** Production predictions should feed monitoring, evaluation, and retraining decisions. If inference is a one-way output, the system cannot learn when it is becoming stale.

Semantics of development, staging and production

The Big Book describes an ML solution as data, code, models, and execution environments moving through development, staging, and production. These stages are not just labels. They encode access controls and quality guarantees.

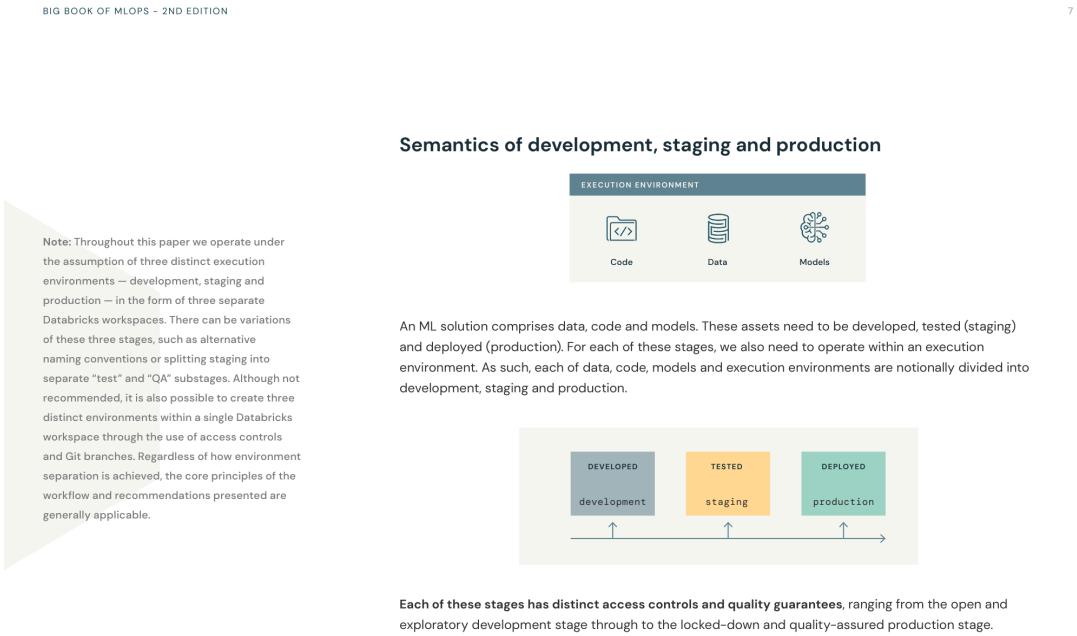

Figure 1: Development, staging, and production semantics from the Big Book of MLOps, p. 7.

- **Development** should allow exploration and fast iteration.
- **Staging** should mirror production enough to validate integration and deployment behavior.
- **Production** should be locked down, automated, monitored, and governed.

The short PDF applies this to Databricks by pairing workspaces with Unity Catalog catalogs. Workspaces provide execution and access isolation. Catalogs provide data and model isolation. The pattern can vary by organization, but the underlying discipline stays the same: separate exploratory work from production authority.

In Databricks terminology, an execution environment includes compute, runtimes, libraries, and automated jobs. Keeping development, staging, and production as distinct execution environments lets a team vary permissions and quality guarantees deliberately: open-ended exploration in development, production-like integration testing in staging, and tightly controlled automated execution in production.

ML deployment patterns

The Big Book distinguishes two promotion strategies:

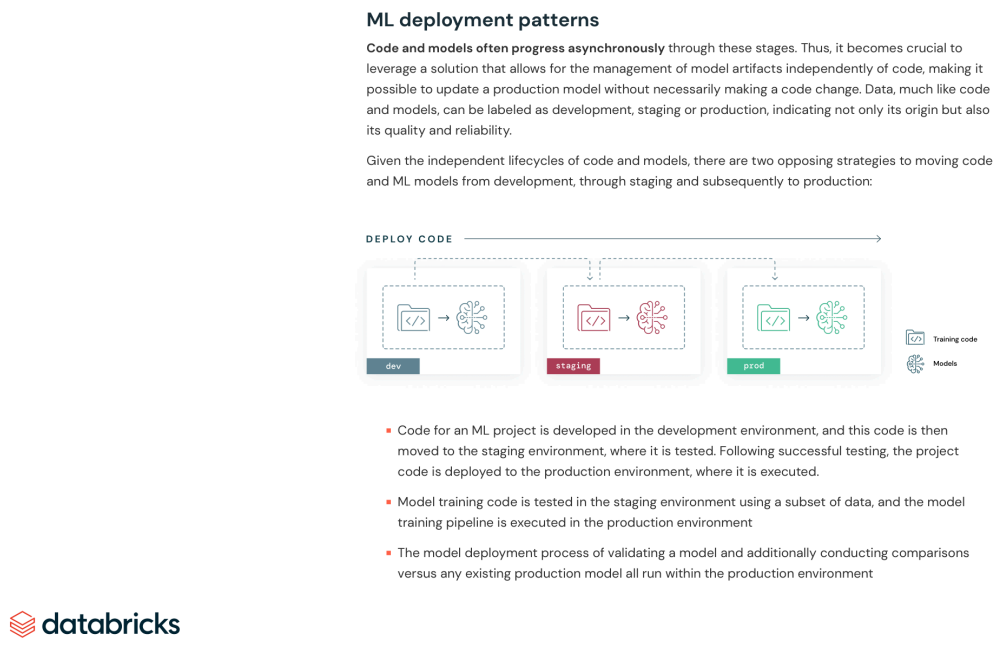

Figure 2: Deploy-code pattern from the Big Book of MLOps, p. 8.

- **Deploy code.** Promote code from dev to staging to prod, then train/register the model in each environment using that environment's data and permissions.
- **Deploy models.** Promote a trained model artifact from one environment to another.

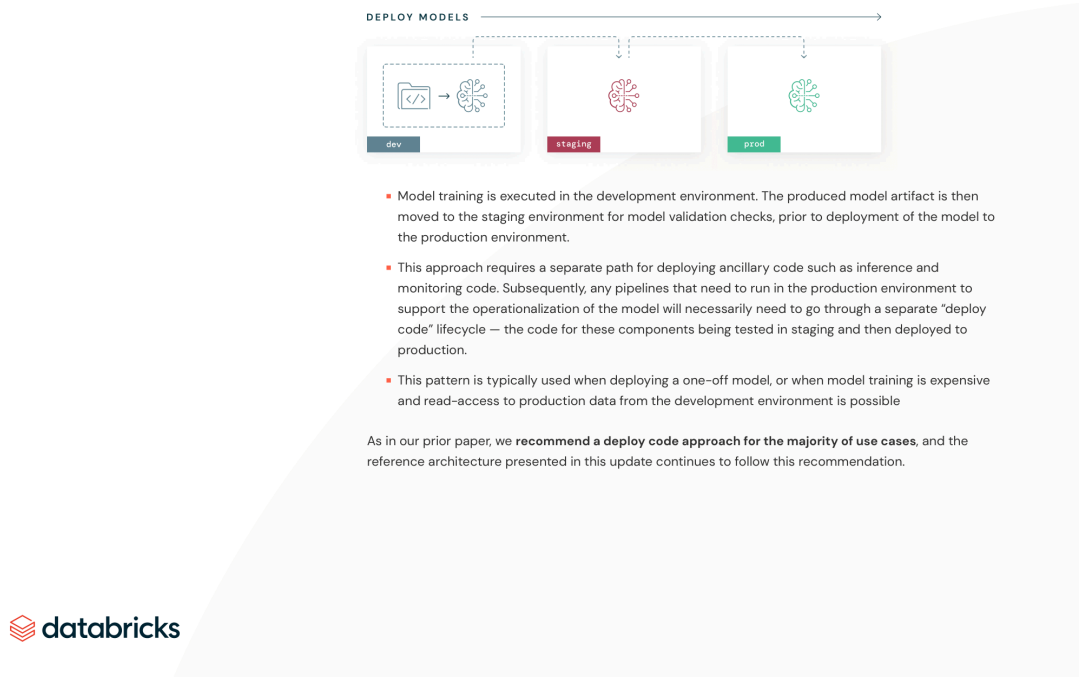

Figure 3: Deploy-model pattern from the Big Book of MLOps, p. 9.

The recommended pattern in the source material is code-first promotion. This creates a cleaner audit chain because the production model is produced by reviewed production code. Model aliases then provide a controlled way to move serving traffic between immutable model versions. Code-first deployment also makes automated retraining safer: the training code, feature code, inference code, and monitoring support code all pass through the same review and integration path before they can affect production.

Model-artifact promotion can still appear in some organizations, especially when training is expensive, hard to reproduce, or constrained to a single workspace. It is the exception rather than the default because it shifts more burden onto provenance and operational controls. If artifact promotion is used, the process must preserve the training code version, data snapshot, run metadata, validation record, approval trail, and any supporting feature, inference, and monitoring code that still needs to move separately.

Recipe: Choose a promotion pattern

Scenario: A team needs to decide whether to promote code or manually promote trained model artifacts.

When to use it: Use this during platform design, onboarding a new ML project, or replacing a notebook-driven release process.

Inputs and prerequisites: Environment map; data access model; model registry; CI/CD system; expected training cost; validation requirements.

Process steps:

1. Identify which assets change independently: feature code, training code, model artifact, inference code, prompt/config, endpoint configuration.
2. Prefer code-first promotion when production can train or validate from governed production data.
3. Use immutable model versions and mutable aliases to separate model identity from serving choice.
4. If artifact promotion is required, store the full provenance chain: code version, data snapshot, run ID, validation result, approver, and target environment.
5. Define rollback as either reverting code or repointing an alias to a previous immutable version.

Design checks and best practices: Production models should be reproducible from source-controlled code and governed data. The live model should be referenced by alias, not by a hard-coded version.

Common failure modes: Hand-carrying an artifact without provenance; retraining in dev and serving in prod; changing inference code and model version together without independent rollback.

Outputs/artifacts: Promotion policy; environment-specific registry naming; alias convention; CI/CD release workflow.

Sources used: Big Book pp. 7-9; MLOps workflows on Azure Databricks; MLOps workflows on Databricks AWS; Model deployment patterns on Databricks AWS; Manage model lifecycle in Unity Catalog.

Platform Capabilities

Modern MLOps is less a hand-built collection of scripts and more a governed platform workflow. Data, models, serving endpoints, inference logs, and monitoring outputs can be managed within a single lakehouse-oriented control plane. The capabilities in this section explain the platform building blocks used throughout the architecture.

Summary: Databricks Tooling

Databricks supports the machine learning lifecycle as a connected platform rather than a loose set of notebooks. The same operating model can ingest data, transform it into governed tables, develop candidate models, track experiments, register versions, serve predictions, monitor behavior, and trigger retraining workflows.

Capability	Role in the MLOps workflow
Auto Loader and Spark	Ingest and transform streaming or incremental data for feature, training, and inference pipelines.
Delta tables	Store raw data, features, labels, predictions, and monitoring outputs with ACID transactions, schema management, and time travel.
Delta Live Tables	Define reliable data pipelines with managed dependencies and data quality expectations.

Capability	Role in the MLOps workflow
Unity Catalog	Govern data, features, functions, and registered models across environment-specific catalogs and schemas.
Databricks SQL, dashboards, and notebooks	Explore data, review metrics, and communicate production behavior to technical and business users.
Workflows	Schedule and orchestrate training, validation, deployment, inference, monitoring, and retraining jobs.
Feature Store / Feature Engineering	Reuse governed feature definitions and keep training and inference feature computation aligned.
AutoML and AutoGluon	Generate baseline models and candidate workflows during exploration while still requiring tracking, validation, and registry controls.
MLflow	Track experiments, log metrics and artifacts, package models, compare runs, and connect training outputs to registered model versions.
Optuna and tuning libraries	Search hyperparameter spaces while recording trial metadata and best-run evidence through MLflow.
Git integration	Version pipeline code, tests, notebooks, bundle configuration, and deployment logic.
Lakehouse Monitoring and MLflow Evaluate	Profile data and predictions, evaluate model quality, detect drift, and provide evidence for retraining decisions.

The practical pattern is straightforward: use Delta and Unity Catalog for governed data and model assets, MLflow for run-level evidence, Git and Asset Bundles for code and deployment definitions, Workflows for repeatable execution, and monitoring tables for the production feedback loop.

Sources used: MLOps workflows on Databricks AWS; MLflow on Databricks; Data profiling / Lakehouse Monitoring; Machine learning with AutoGluon, an open source AutoML library.

Unity Catalog

Unity Catalog is the governance layer in the source material. It provides a namespace for data and AI assets, including registered models. The short PDF emphasizes a three-level registered-model namespace such as `catalog.schema.model`, with immutable model versions and mutable aliases such as `@champion`, `@challenger`, and optional `@shadow`.

The conceptual move is important: a model is not just a file. It is an asset with lineage, access control, version history, metadata, evaluation evidence, and deployment semantics.

Benefits and architecture implications
Using Unity Catalog as the shared governance layer supports:

- consistent permissions over data and model assets,
- lineage from data to features to runs to model versions,
- model version metadata and descriptions,
- aliases for lifecycle state,
- clearer separation between dev, staging, and production assets.

The main architecture implication is that naming matters. Catalogs, schemas, experiments, registered models, and aliases become part of the operating system of MLOps. Teams should encode naming in templates rather than relying on documentation alone.

Recipe: Register and promote a governed model
Scenario: A trained model needs to move from candidate to production use.

When to use it: Use for supervised ML models, batch inference models, real-time endpoints, and model variants that need auditable lifecycle control.

Inputs and prerequisites: MLflow run; model signature; input example; validation dataset; Unity Catalog registered model; alias convention.

Process steps:

1. Log the model, parameters, metrics, artifacts, signature, and input example to an MLflow run.
2. Register the candidate model to the correct Unity Catalog model name.
3. Attach model version metadata: source run ID, data version, validation status, owner, approval status, and relevant segment metrics.
4. Run validation checks for format, metadata completeness, performance, and domain-specific constraints.
5. If the candidate passes validation, assign or move the `@challenger` alias.
6. Compare `@challenger` against the current `@champion`.
7. Promote by repointing `@champion`; roll back by repointing it to the previous immutable version.

Design checks and best practices: Use aliases for lifecycle state. Avoid deprecated stage-style thinking when working with Unity Catalog model lifecycle. Keep versions immutable and make promotion a metadata operation.

Common failure modes: Treating aliases as documentation instead of deployment control; missing signatures; registering multiple unrelated problems under one model; failing to log the training data version.

Outputs/artifacts: Registered model version; validation report; alias update; promotion record.

Sources used: Manage model lifecycle in Unity Catalog; MLflow on Databricks; `MLOps Questions (1).pdf` Topic 1.

Model Serving

Model Serving provides managed real-time serving endpoints. In the Big Book architecture, it reduces the amount of custom infrastructure required to expose models for low-latency inference. The linked Model Serving source positions serving as a deployment target for models and endpoints that can be governed and monitored.

Benefits and architecture implications

The architecture benefit is separation of concerns. Training pipelines produce and validate model versions. Deployment logic chooses which model version or alias should serve. Serving infrastructure handles request/response operation, traffic management, and endpoint behavior.

For many systems, not every prediction path should be real time. Batch and streaming inference remain valid, often preferable, deployment modes. Real-time serving should be chosen when the product or process needs low-latency decisions.

Lakehouse Monitoring

Lakehouse Monitoring provides a way to monitor data and model behavior using profile and drift metrics. In the short PDF, the relevant monitor type for model monitoring is an inference-log monitor: it works over inputs, predictions, and optionally ground truth.

Benefits and architecture implications

Monitoring changes the architecture from "deploy and hope" to "deploy and observe." A production ML system should record enough inference data to detect changes in input distributions, prediction distributions, data quality, and model quality once labels arrive.

Monitoring is also a governance primitive. Without inference logs and monitoring tables, post-deployment review depends on fragmented logs or custom reporting.

Recipe: Monitor model quality and drift

Scenario: A model is in production and needs ongoing monitoring for drift, quality, and operational behavior.

When to use it: Use after any production deployment, especially when input distributions, labels, or business policies may change over time.

Inputs and prerequisites: Inference table; prediction outputs; optional ground-truth labels; baseline table or baseline period; monitor owner; alert thresholds.

Process steps:

1. Capture inference records with request inputs, prediction outputs, model version or alias, timestamp, and relevant identifiers.
2. Attach an inference-log monitor where supported.
3. Define baseline and comparison windows.
4. Track profile metrics, drift metrics, and quality metrics once labels arrive.
5. Review metrics by important segments, not only globally.
6. Route significant drift or quality loss to an investigation and retraining decision.
7. Store monitoring outputs where they can be joined to model versions and deployment events.

Design checks and best practices: Monitor data behavior and model behavior separately. A stable model can still fail if data changes. A stable data profile can still hide segment-level harm.

Common failure modes: No ground-truth join path; monitoring only aggregate accuracy; missing model version in inference logs; retraining automatically without validation.

Outputs/artifacts: Inference table; profile metrics table; drift metrics table; monitoring review; retraining ticket or run trigger.

Sources used: Data profiling / Lakehouse Monitoring; Monitor model services using inference tables; Big Book pp. 49-50.

Design Decisions

Design decisions are where MLOps becomes concrete. A useful platform has opinions about asset organization, environment separation, CI/CD, validation gates, deployment modes, and monitoring loops. The goal is not to eliminate judgment. The goal is to ensure teams make recurring decisions consistently.

MLflow, Model Registry, and Reproducibility

MLflow is the evidence system for model development. Each meaningful training or evaluation run should record parameters, metrics, artifacts, tags, model signatures, input examples, and model packages. Manual logging is still important even when autologging is available: domain-specific metrics, validation plots, drift reports, feature importance files, and segment-level evaluation artifacts often need explicit `log_metric`, `log_artifact`, or `log_model` calls.

The MLflow UI supports run comparison, metric review, artifact inspection, lineage review, and handoff between experimentation and model registration. The MLflow Client API makes those same actions automatable: search runs to identify the best candidate, register the selected run output to a Unity Catalog model, set or remove tags, update descriptions, and move aliases when validation or deployment decisions change.

Reproducibility depends on recording more than a score. A useful run contains the Git SHA, branch, data version or table snapshot, environment, training configuration, dependency context, evaluator version, and validation result. The registered model version should inherit enough of that context for an operator to understand why it exists and whether it is eligible for deployment.

Unity Catalog is the preferred registry target for governed workflows. It uses a three-level name such as `catalog.schema.model`, stores immutable model versions, and supports aliases for lifecycle roles. A common production flow is to register a new candidate, tag it with validation evidence, assign `@challenger` after checks pass, compare it against the current `@champion`, and promote by moving the `@champion` alias. This keeps deployment decisions reversible without rewriting model artifacts.

Sources used: MLflow on Databricks; Manage model lifecycle in Unity Catalog; `MLOps Questions (1).pdf` Topic 1.

Unity Catalog

Organizing data and AI assets

A practical organization scheme should make environment, ownership, and purpose visible. The short PDF proposes environment-specific catalogs, team or domain schemas, and problem-specific registered model names. That pattern is strong because it maps directly to access control and operational ownership.

For example:

```
catalog: dev | staging | prod
schema:  team_or_domain
model:   problem_variant
alias:   champion | challenger | shadow
```

Experiments should also be structured. Workspace experiments are better than notebook-scoped experiments for production-oriented work because they are discoverable, shared, and easier to govern. A useful convention is one experiment per model problem per environment.

Concepts

The key concepts are:

- **Experiment:** A container for runs for a model problem or project.
- **Run:** A single training or evaluation execution with parameters, metrics, artifacts, tags, and inputs.
- **Registered model:** A governed named asset containing immutable versions.
- **Model version:** A specific registered artifact plus metadata.
- **Alias:** A mutable pointer to a model version, used for lifecycle or deployment intent.
- **Serving endpoint or inference pipeline:** The production consumer of a model alias or version.

This hierarchy lets a reader trace from a prediction back to the serving configuration, model alias, model version, MLflow run, code version, and data snapshot.

Considerations

The main tradeoff is between freedom and consistency. Development teams need enough flexibility to explore, but production systems need conventions that scale. The best compromise is to encode conventions in templates and automation. Naming rules, required tags, validation checks, and deployment gates should be built into the project skeleton.

Important run tags include environment, Git SHA, branch, author, trigger, data version, model type, and domain-specific segments. Tags are not decoration. They are the connective tissue of traceability.

Recommended organization

Use a layered convention:

- Workspace experiment paths identify team, environment, and problem.
- Unity Catalog catalogs identify environment.
- Schemas identify team or domain ownership.

- Model names identify problem and variant.
- Aliases identify deployment state.
- Tags identify provenance and validation context.

Recipe: Set up experiment tracking and reproducibility

Scenario: A team is beginning a new ML initiative and needs consistent experiment tracking.

When to use it: Use at project creation, before the first serious training run.

Inputs and prerequisites: Project name; team/domain; environment names; Git repository; training data table; MLflow tracking.

Process steps:

1. Create a workspace experiment per model problem and environment.
2. Define a required run tag taxonomy: environment, Git SHA, branch, author, trigger, data version, model type, and business segment.
3. Log parameters, metrics, artifacts, model signature, and input example.
4. Log dataset information or table version so training inputs are traceable.
5. Use parent/child runs for sweeps, variants, or deployment cycles.
6. Store evaluation artifacts such as confusion matrices, SHAP outputs, calibration plots, or segment metrics in governed storage.
7. Make the tracking setup part of the project template, not a wiki page.

Design checks and best practices: A future operator should be able to answer "what trained this model?" from the run record alone.

Common failure modes: Random run names; missing Git SHA; one giant experiment for unrelated models; no data version; metrics logged only globally.

Outputs/artifacts: Experiment path; run tag schema; reusable training template; evaluation artifact convention.

Sources used: MLflow on Databricks; `MLOps Questions (1).pdf` Topic 1; Big Book pp. 34-38.

Model Serving

Pre-deployment testing

Pre-deployment testing should verify both software behavior and model behavior. Unit tests cover transformation logic. Integration tests verify that the pipeline can run in the target environment. Model validation checks the artifact, metadata, signature, input/output shape, and performance thresholds.

For regulated or sensitive workloads, validation should include segment-level metrics. A high global score can hide unacceptable behavior for a subgroup or operating region.

Real-time model deployment

Real-time deployment is appropriate when inference latency affects the user experience or business process. It is less appropriate when predictions are naturally periodic, when inputs arrive in large files, or when decisions are consumed in downstream batch processes.

Implementing in Databricks
Use Model Serving when a governed endpoint is required. Use aliases so the endpoint can refer to the current production model without hard-coding a model version. Combine this with monitoring so serving records are observable after deployment.

Recipe: Choose batch, streaming, or real-time inference
Scenario: A model is ready for production, but the serving mode is undecided.

When to use it: Use during architecture design or when replacing ad hoc notebook inference.

Inputs and prerequisites: Prediction latency requirement; input arrival pattern; output consumption pattern; cost constraints; monitoring needs.

Process steps:

1. If decisions are periodic and consumers can wait, choose batch inference.
2. If inputs arrive continuously and outputs feed near-real-time downstream systems, consider streaming.
3. If a user or service needs a low-latency response, consider real-time serving.
4. Define how each mode records model version, inputs, predictions, and timestamps.
5. Define rollback before deployment: job rollback for batch/streaming or alias/traffic rollback for serving endpoints.
6. Confirm that monitoring can observe the chosen mode.

Design checks and best practices: Serving mode is a product and operations decision, not a prestige decision. Real time is not automatically better.

Common failure modes: Building a REST endpoint for weekly predictions; batch jobs without model version logging; streaming pipelines without data quality checks; real-time endpoints without traffic rollback.

Outputs/artifacts: Serving-mode decision record; inference pipeline or endpoint design; monitoring plan.

Sources used: Big Book pp. 46-49; Model deployment patterns; Deploy models using Model Serving.

Reference Architecture

The Big Book reference architecture is the heart of the handbook. It describes how work moves through development, staging, and production in a multi-environment setup.

Multi-environment view

The architecture separates workspaces and catalogs across environments. Development is optimized for exploration, staging for integration validation, and production for governed execution. Code moves through Git and CI/CD. Data and models remain governed in the environment where they are used.

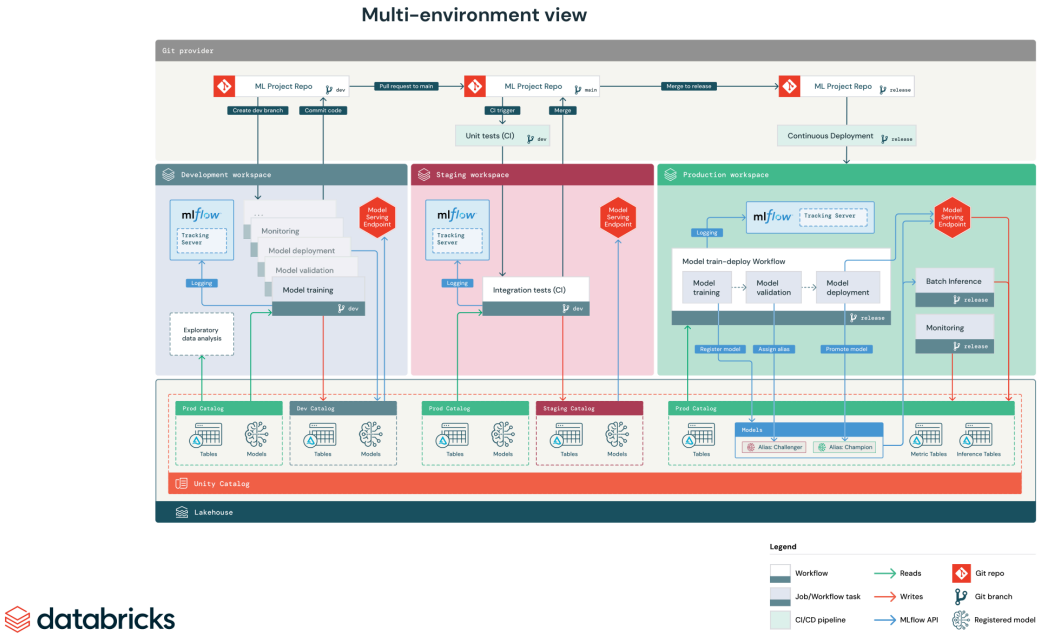

Figure 4: Multi-environment reference architecture overview from the Big Book of MLOps, p. 32.

The short PDF emphasizes that this resolves a common tension: teams want realistic data for validation, but they also need environment separation. Unity Catalog permissions can allow governed read access without collapsing dev, staging, and prod into one uncontrolled space.

Development

Development is where hypotheses become code. It includes data exploration, feature engineering, model training development, validation logic development, and deployment logic development.

Data

Development data should be useful enough to support meaningful experimentation but controlled enough to protect production assets. Where production data is required, access should be governed and read-only unless the project has a deliberate sandbox copy.

Databricks recommends that development workspaces have read-only access to relevant production assets where appropriate. This can include production data, inference tables, metric tables, and production model versions. The point is not to let development mutate production; it is to let data scientists diagnose current behavior, compare candidate models against production models, and understand whether the available data can solve the business problem. If direct read-only access is not possible, a governed production snapshot can be written to the development catalog.

Exploratory data analysis (EDA)

EDA should produce reusable knowledge, not only notebook cells. The durable outputs are data assumptions, candidate features, data quality checks, and failure cases.

AutoML and AutoGluon

AutoML is useful in EDA because it creates baselines quickly. In the Databricks workflow, AutoML can generate trial runs, notebooks, summary statistics, and candidate model code that a team can review and reproduce. The important boundary is that AutoML accelerates exploration; it does not replace source control, MLflow tracking, model registry governance, validation gates, or production monitoring.

AutoGluon is an open source AutoML library that exposes a compact workflow for tabular problems: load data with `Dataset()`, train with `fit()`, and score new data with `predict()`. Behind that simple interface, AutoGluon performs preprocessing, chooses feature handling strategies, trains multiple model families, and combines models through ensembling and stacking. It can also work within a time budget, returning the best models trained within the allowed time.

In an MLOps workflow, an AutoGluon model should be treated like any other candidate model. The training run should log the code version, data version, AutoGluon configuration, time budget, metrics, artifacts, and selected model. If the candidate is promising, it should be registered, validated, compared to the current champion, and monitored through the same lifecycle as a hand-built model.

Recipe: Use AutoML and AutoGluon responsibly in an MLOps workflow

Scenario: A team wants a fast baseline or candidate model without bypassing production controls.

When to use it: Use during EDA, early model selection, or benchmark creation for tabular classification/regression problems.

Inputs and prerequisites: Training dataset; target label; evaluation metric; time budget; MLflow tracking; registry location; validation criteria.

Process steps:

1. Start with a clearly versioned training dataset and target label.
2. Run AutoML or AutoGluon to generate baseline trials, using a time budget when cost or iteration speed matters.
3. Log the AutoML configuration, data version, metrics, artifacts, selected model, and generated code or notebook.
4. Review the generated model and preprocessing assumptions; do not promote the output blindly.
5. Package the selected candidate so it can enter the normal registry and validation workflow.
6. Compare it against the current champion using the same holdout, segment, and operational criteria used for hand-built models.
7. Promote only through the normal alias, deployment, and monitoring path.

Design checks and best practices: AutoML is a baseline accelerator, not a governance shortcut. Generated models still need reproducibility, explainability appropriate to the domain, and production validation.

Common failure modes: Treating the leaderboard winner as production-ready; failing to log AutoML settings; ignoring preprocessing assumptions; skipping segment metrics; promoting a model that cannot be reproduced.

Outputs/artifacts: Baseline run set; selected candidate model; generated notebook/code; evaluation report; registered model candidate.

Sources used: MLOps workflows on Databricks AWS; Machine learning with AutoGluon, an open source AutoML library; MLflow on Databricks.

Project code

Project code should move out of notebooks into versioned modules as soon as logic becomes reusable. Notebooks can remain useful interfaces, but production behavior should live in testable code.

Model training development

Training development should create the training pipeline and log its runs. The goal is not only to find a strong model but also to build the repeatable procedure that can train it again.

Model validation and deployment development

Validation logic should be developed with the model, not added after the model is "done." Define the checks that decide whether a candidate can become a challenger and whether a challenger can become champion.

Commit code

Committing code is the boundary between exploratory work and shared engineering. A pull request should include code, tests, configuration, and enough run evidence to review the change responsibly.

Staging

Staging is where the system proves it can run outside a data scientist's personal workflow.

Data

Staging data should mirror production shape and constraints. It may use samples, governed copies, or read access to production-like features depending on the organization's policy. Assets written into the staging catalog are often temporary and retained only long enough to complete testing, but the environment should still be production-like enough to expose integration failures before release.

Merge code

Code is merged only after review and automated checks. The staging environment should catch integration errors that unit tests cannot see.

Integration tests (CI)

Integration tests should deploy or exercise the pipeline in staging, run a smoke train/inference path, and verify that expected artifacts are created. Unit tests should run first against transformation, feature, validation, and packaging code; integration tests should then prove that the pieces work together in the target environment. For a real-time serving system, staging should test the serving infrastructure as well as the model pipeline. Expensive training can be shortened with smaller samples or fewer iterations, but the smoke test should still prove that

feature computation, training, validation, deployment, inference, and monitoring components work together.

Merge

Merge should be a controlled transition. Once code merges to the protected branch, CD can deploy to production using service principal authority rather than individual user authority.

Cut release branch

Release branches are useful when production deployments need stabilization or coordination across teams. In the Databricks workflow pattern, the release branch can be the signal that CI/CD should update production jobs with reviewed code and environment-specific configuration. It should not become a long-lived fork that hides production behavior from mainline development.

Production

Production is where the ML system becomes an operated service.

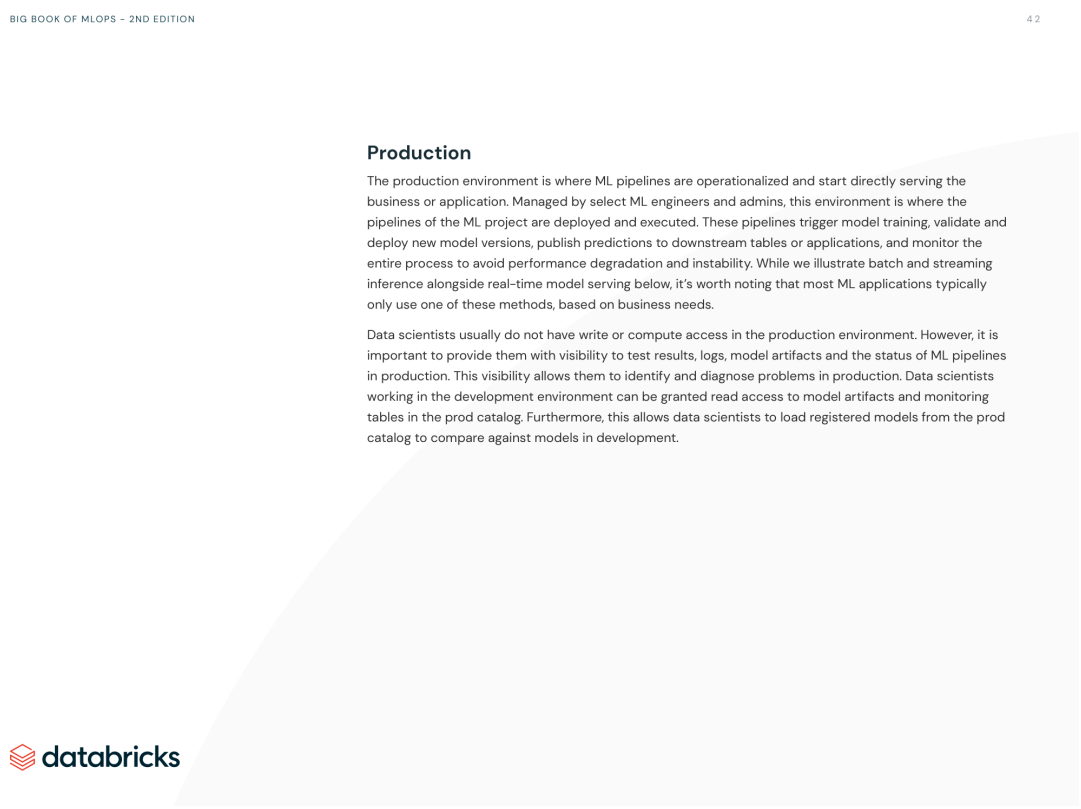

Figure 5: Production workflow from the Big Book of MLOps, p. 42.

Model training

Production training should be run by production-approved code with production permissions. This is what makes code-first promotion auditable. The production training task should log model metrics, parameters, tags, data versions, and the model artifact to the production tracking context, then register the resulting model version to the production catalog.

Model validation

Validation should be a gate, not a report that nobody reads. It should check metadata, schema, performance, segment behavior, and operational readiness. Validation outcomes should be written back as model metadata or tags. A tag such as `model_validation_status` can move from `PENDING` to `PASSED` or `FAILED`, giving operators and data scientists a visible record of why a model did or did not become a challenger.

Model deployment

Deployment should update controlled production pointers such as aliases or endpoint configuration. It should be reversible. A typical deployment step compares the validated challenger with the current champion using offline evaluation or an online comparison. If the challenger wins, deployment repoints the champion alias or updates endpoint traffic. If there is no champion, the challenger should still be compared against a business heuristic or acceptance threshold.

Model Serving

Model Serving is appropriate for real-time inference paths. Production endpoints should have ownership, access controls, monitoring, and rollback. When updating an existing endpoint, Model Serving can keep the current configuration running while the new configuration becomes ready, enabling zero-downtime updates. Endpoints can also serve multiple models with traffic splits, which supports online champion/challenger comparisons.

Inference: batch or streaming

Batch and streaming inference are first-class production patterns. They still need model version logging, input/output capture, monitoring, and retraining hooks. Production inference should load the governed model through an alias such as `@champion` whenever possible, so promotion is decoupled from scoring code and the next run automatically uses the approved model version.

Lakehouse Monitoring

Monitoring should be part of the production architecture from the start. It should observe data quality, drift, prediction behavior, and accuracy once labels are available.

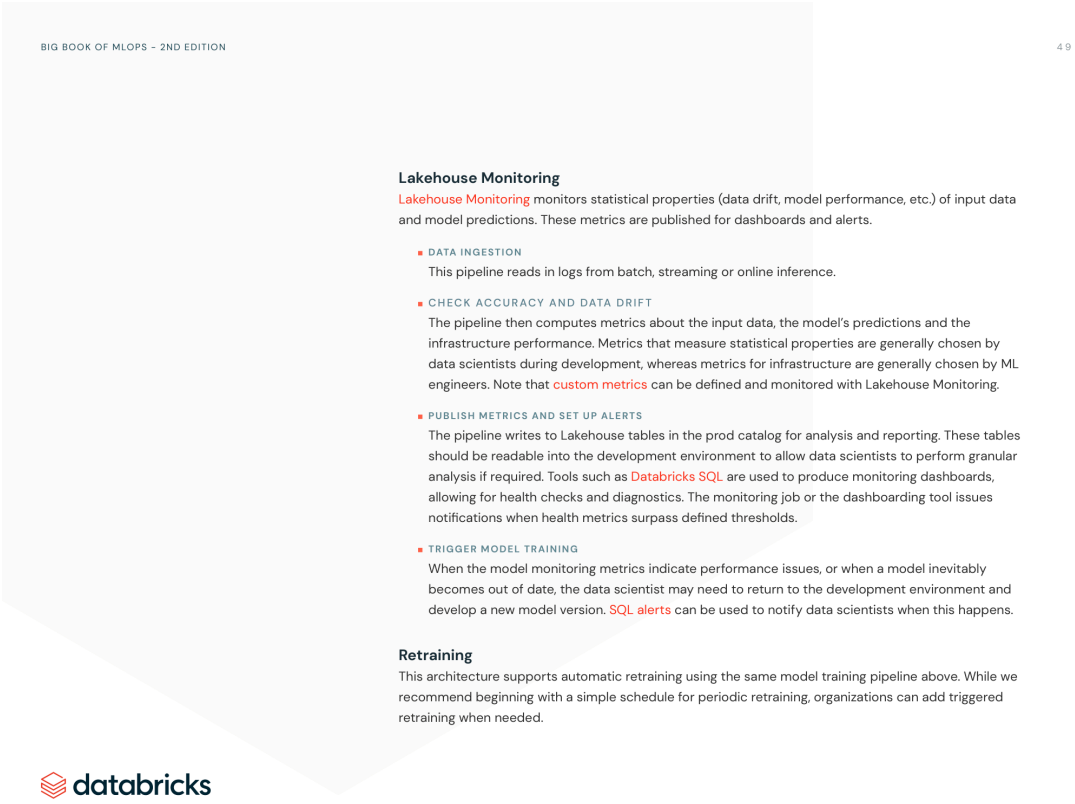

Figure 6: Inference, monitoring, and retraining loop from the Big Book of MLOps, p. 49.

Retraining

Retraining should be a governed response to evidence, not a reflex. A retraining loop should specify the trigger, data window, validation criteria, promotion path, and rollback plan. Scheduled retraining is appropriate when fresh labels or data arrive predictably. Triggered retraining is appropriate when monitoring detects drift, degraded quality, or another anomaly and sends an alert or webhook into the training workflow.

Recipe: Design a retraining loop

Scenario: A production model needs periodic or event-driven retraining.

When to use it: Use when model quality depends on changing data, labels, behavior, products, or policy.

Inputs and prerequisites: Monitoring metrics; label availability; training pipeline; validation gate; registry aliases; deployment workflow.

Process steps:

1. Define retraining triggers: schedule, drift threshold, accuracy drop, business event, SQL alert, webhook, or manual review.
2. Define the training data window and label maturity rule.
3. Run the production training pipeline with logged data version and code version.
4. Register the result as a new immutable model version.
5. Run validation against holdout, recent data, and required segments.
6. Assign `@challenger` only if validation passes.

7. Compare challenger against champion and promote only if the deployment criterion is met.
8. Monitor post-promotion behavior and keep rollback available.

Design checks and best practices: Retraining is not deployment. A retrained model still needs validation and promotion.

Common failure modes: Automatic retraining that silently worsens performance; labels arriving too late or inconsistently; no record of training window; overwriting the champion without comparison.

Outputs/artifacts: Retraining policy; scheduled workflow; validation report; model version; promotion record.

Sources used: Big Book pp. 42-50; MLOps workflows on Azure Databricks; MLOps workflows on Databricks AWS; Data profiling / Lakehouse Monitoring.

Implementing MLOps on Databricks

Implementation should start from templates. Databricks Asset Bundles describe jobs, pipelines, and resources as configuration with environment-specific targets. MLOps Stacks builds on that idea as an opinionated project scaffold. A new stack can be initialized with `databricks bundle init mlops-stacks`, after which the generated project can be customized to match the organization's process.

MLOps Stacks is organized around three components. **ML code** gives data scientists a standardized project structure for notebooks, training, testing, and batch inference. **ML resources as code** defines jobs, pipelines, workspaces, and other deployment resources through bundle configuration. **CI/CD** wires the project into GitHub Actions or Azure DevOps so staging tests and production deployments run through automation. The role split is deliberate: data scientists and ML engineers can develop model code, while ML/platform engineers configure service principals, workspace targets, and production deployment workflows.

Recipe: Build CI/CD for an ML pipeline

Scenario: A team wants to replace manual notebook execution with automated promotion.

When to use it: Use for any ML pipeline that must move from dev to staging to production.

Inputs and prerequisites: Git repository; CI system; Databricks workspaces; service principal; Asset Bundle; tests; environment targets.

Process steps:

1. Define the deployable unit as a Databricks Asset Bundle.
2. Create targets for development, staging, and production.
3. Run unit tests on pull request.
4. Deploy to staging and run an integration smoke test.
5. On merge to the protected branch, deploy to production through a service principal.
6. Run production training, registration, validation, and alias promotion as controlled workflow steps.
7. Keep rollback explicit: revert code, rerun a previous release, or repoint model alias.

Design checks and best practices: The CI/CD service principal should own production deployment authority. Humans approve where judgment is needed; automation executes repeatable steps.

Common failure modes: CI tests only Python importability; production deployment from a user's notebook session; environment-specific configuration hard-coded in code; no smoke test.

Outputs/artifacts: Bundle configuration; CI workflow; CD workflow; service principal permissions; release record.

Sources used: CI/CD for ML; Declarative Automation Bundles; MLOps Stacks; `MLOps Questions (1).pdf` Topics 2-4.

Recipe: Apply MLOps Stacks or Asset Bundles
Scenario: Multiple ML initiatives need a shared delivery pattern.

When to use it: Use when teams repeat similar project structures, environment targets, CI/CD workflows, and governance conventions.

Inputs and prerequisites: Standard environment names; CI provider; workspace hosts; catalog names; team ownership; project template requirements.

Process steps:

1. Start from MLOps Stacks when the project fits the opinionated training and batch inference pattern.
2. Initialize with `databricks bundle init mlops-stacks`.
3. Choose the appropriate project scope: ML code, CI/CD components, or both.
4. Use Asset Bundles to declare jobs, pipelines, models, resources, and environment targets.
5. Encode naming, tags, permissions, and validation gates into the template.
6. Support role separation where useful: project code for data scientists and CI/CD/resource configuration for ML/platform engineers.
7. Roll out in phases: foundation first, CI/CD integration next, stronger quality gates after the workflow is stable.

Design checks and best practices: A template should reduce choices that should not vary by project. It should not prevent legitimate domain-specific validation.

Common failure modes: Treating the stack as a generated one-time copy; allowing each team to fork conventions immediately; putting secrets or environment-specific values in source code.

Outputs/artifacts: Project scaffold; bundle config; environment targets; CI/CD workflows; project onboarding checklist.

Sources used: MLOps Stacks: model development process as code; MLOps Stacks on Databricks AWS; Databricks mlops-stacks GitHub archive; DABs for MLOps Stacks; MLOps Gym Crawl.

LLMOps

LLMOps extends MLOps to systems where the "model" may be a hosted foundation model, a prompt, an agent, a chain, a retrieval pipeline, a fine-tuned model, a serving endpoint, a vector index, an evaluation dataset, or a combination of all of these. The Big Book's LLMOps section is valuable because it treats LLM applications as systems, not only as model calls.

What changes with LLMs?

LLMs change the unit of control. In classic ML, the registered model artifact often represents the main prediction behavior. In LLM applications, behavior may depend on prompt templates, retrieval data, embeddings, reranking, tool calls, endpoint configuration, safety filters, and evaluation rubrics.

This means version control must cover more than model weights. Prompts, agents, chains, retrieval settings, vector indexes, evaluation datasets, endpoint choices, and cost/performance settings are all production assets. They should move through environments with the same discipline as feature pipelines, training code, and inference code.

The common MLOps foundations still apply. LLM projects still need separate development, staging, and production environments; Git-based version control; MLflow tracking where appropriate; governed data in lakehouse tables; model lifecycle management; and CI/CD. The implementation changes because the asset graph is wider, not because the operating discipline disappears.

Key components of LLM-powered applications

Prompt engineering

Prompts are executable behavior. Treat them like code or configuration: version them, review them, evaluate them, and deploy them through environments.

Leveraging your own data

Enterprise LLM systems often become useful when they can access domain data. That data access path must be governed, refreshed, evaluated, and monitored.

Retrieval augmented generation (RAG)

RAG connects a user query to relevant documents or records, then passes retrieved context to an LLM. It is often preferable to fine-tuning when the goal is to ground responses in changing private knowledge.

Typical RAG workflow

A typical RAG workflow includes ingestion, chunking, embedding, indexing, retrieval, prompt construction, generation, and evaluation. Each stage can fail independently. Bad chunks, stale embeddings, poor retrieval, or weak prompts can all produce poor answers even when the foundation model is strong.

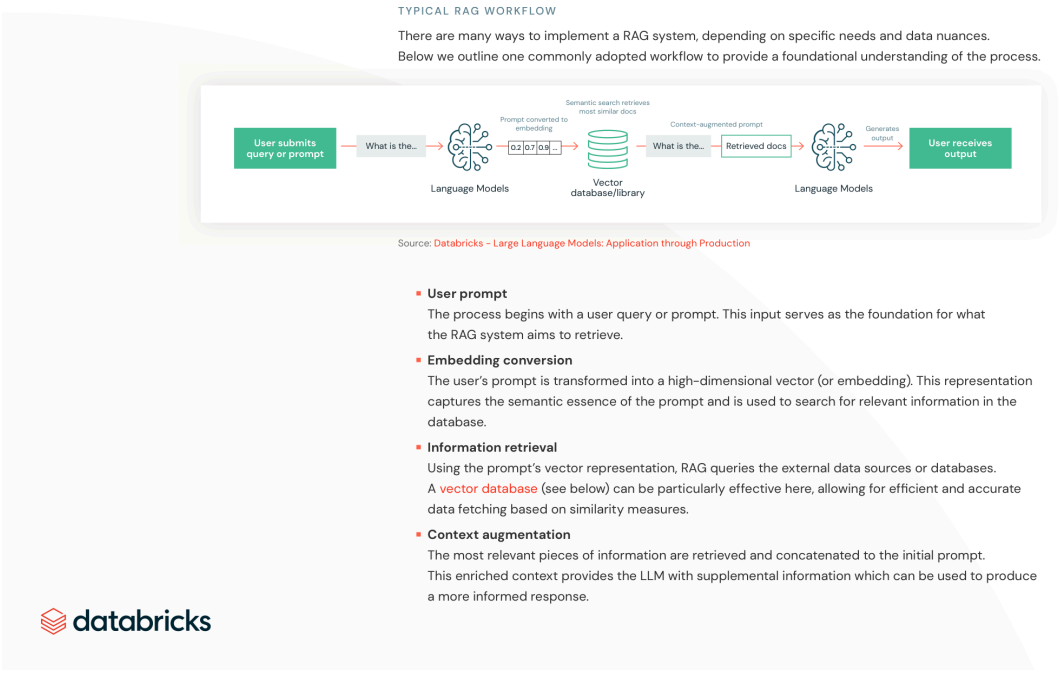

Figure 7: Typical RAG workflow from the Big Book of MLOps, p. 59.

Vector database

The linked source on Databricks AI Search describes a managed retrieval capability built around indexes. The short PDF highlights Delta Sync Index behavior: changed rows can be incrementally embedded, while changing the embedding model usually implies a full re-embedding and index rebuild.

Benefits of vector databases in a RAG workflow

Vector indexes make semantic retrieval practical at scale. They also introduce lifecycle concerns: freshness, embedding model choice, index rebuild cost, access control, and evaluation.

Recipe: Use vector search and retrieval workflows

Scenario: A team needs a RAG workflow over governed enterprise data.

When to use it: Use when answers should be grounded in private or changing knowledge rather than only in a foundation model's pretraining.

Inputs and prerequisites: Source Delta table or document corpus; chunking strategy; embedding model; index type; freshness requirement; evaluation questions; access controls.

Process steps:

1. Define the retrieval corpus and governance boundary.
2. Choose chunking rules that preserve useful context.
3. Choose managed or self-managed embeddings based on control, cost, and compliance needs.

4. Create an index with an appropriate sync mode: continuous for low-latency freshness or triggered for controlled refresh.
5. Evaluate retrieval quality before evaluating generated answers.
6. Version prompt templates and retrieval configuration together.
7. If changing embedding models, plan a full re-embed and dual-index cutover.

Design checks and best practices: Retrieval quality is a first-order system metric. Do not evaluate only final text if the retrieved context is wrong.

Common failure modes: Re-indexing surprise costs; stale indexes; unauthorized data in retrieval results; evaluating answer style but not source relevance.

Outputs/artifacts: Index configuration; embedding policy; retrieval evaluation set; RAG prompt/config version; cutover plan.

Sources used: Big Book pp. 58-61; Databricks AI Search; Create AI Search endpoints and indexes; `MLOps Questions (1).pdf` Topic 5.

Fine-tuning LLMs
Fine-tuning changes model behavior by training on additional examples. It is useful when the desired behavior is not achieved reliably through prompting or retrieval alone, or when style, task format, or domain adaptation must be learned.

When to use fine-tuning?
Use fine-tuning when you have representative training examples and a stable behavior target. Do not use it as a substitute for retrieval when the main requirement is access to changing facts.

Fine-tuning in practice
Fine-tuning still needs the MLOps basics: dataset versioning, run tracking, evaluation gates, registry metadata, deployment controls, and rollback.

Pre-training
Pre-training is a much heavier process than fine-tuning. It is usually relevant only when an organization has large-scale data, infrastructure, and a strategic reason to train a model from scratch or continue training at scale.

When to use pre-training?
Use pre-training only when the organization needs deep control over the model and has the resources to support the full lifecycle.

Pre-training in practice
Pre-training is an infrastructure and data-governance program, not a single modeling task. The operational burden includes data curation, compute planning, evaluation, safety, deployment, and long-term maintenance.

Third-party APIs vs. self-hosted models
Third-party APIs can accelerate experimentation and reduce operational burden. Self-hosted or provisioned deployments can provide stronger control over cost, latency, privacy, customization, and availability. The right choice depends on workload stability, compliance requirements, model choice, and expected traffic. Centralized API governance is valuable be-

cause it gives the organization a control point for credentials, auditing, fallback, and provider switching without scattering provider-specific logic throughout applications.

Model evaluation
LLM evaluation is harder than many classic ML evaluations because outputs can be open-ended. Evaluation often combines automated metrics, LLM-based judging, human feedback, regression test sets, and task-specific rubrics.

LLMs as evaluators
LLMs can help evaluate outputs, but their judgments are themselves model outputs. Treat evaluator prompts and rubrics as versioned assets, and periodically calibrate them against human review.

Human feedback in evaluation
Human feedback remains important for subjective quality, policy alignment, and high-impact decisions. The goal is not to make every judgment manual, but to create a reliable calibration loop. Human review should feed evaluation sets, monitoring review, future fine-tuning decisions, and regression tests for prompts or chains.

Packaging models or pipelines for deployment
An LLM application package may include prompt templates, retrieval configuration, model endpoint references, tools, safety filters, and evaluation settings. Package these as versioned deployable assets, not as scattered notebook state.

LLM Inference
LLM inference introduces cost, latency, throughput, and quality tradeoffs. The linked Foundation Model API and Provisioned Throughput sources distinguish experimentation-style usage from production capacity planning.

Real-time inference
Real-time LLM inference is appropriate for interactive applications. It needs latency budgets, rate controls, fallback behavior, logging, and evaluation.

Batch inference
Batch inference is useful for offline enrichment, classification, summarization, or periodic generation where latency is less important than throughput and cost control.

Inference with large models
Large models can improve capability but increase latency and cost. A production design should test whether a smaller model, better retrieval, or better prompting can meet the requirement.

Managing cost/performance trade-offs
Cost/performance management is an operating discipline. Track token usage, latency, endpoint capacity, cache opportunities, prompt length, retrieval context length, and model choice.

Methods for reducing costs of inference

Cost controls include shorter prompts, better retrieval filtering, caching, batching, smaller models, provisioned throughput for stable workloads, fallback routing, and regular review of model choices.

Recipe: Operate LLMOps with inference logging and governance

Scenario: A production LLM application needs quality monitoring, cost control, and auditability.

When to use it: Use for RAG systems, agents, prompt-driven applications, or foundation-model API integrations.

Inputs and prerequisites: Endpoint configuration; prompt/chain version; retrieval config; evaluation set; inference table; cost and latency targets; governance policy.

Process steps:

1. Version prompts, chain code, retrieval settings, and endpoint configuration.
2. Log requests and responses to governed inference tables where supported.
3. Track model/endpoint ID, prompt version, retrieval index version, latency, token usage, and error state.
4. Evaluate outputs with a mix of automated checks, LLM-as-judge where appropriate, and human feedback for calibration.
5. Use traffic splitting or fallback for new model or prompt versions.
6. Review cost/performance metrics before scaling provisioned capacity.
7. Keep rollback simple: revert prompt/config or route traffic back to the prior endpoint/model.

Design checks and best practices: Treat prompts and retrieval config as production assets. Logging must support quality review and compliance, not only debugging.

Common failure modes: Changing prompts without versioning; no record of retrieved context; judging output without task-specific rubrics; ignoring token cost until after rollout.

Outputs/artifacts: Versioned LLM app config; inference table; evaluation report; traffic routing plan; rollback record.

Sources used: Big Book pp. 51-74; LLMOps workflows; LLMOps workflows on Databricks AWS; AI governance with Unity AI Gateway; Monitor model services using inference tables; Foundation Model APIs; Provisioned throughput Foundation Model APIs.

Reference architecture

The LLMOps reference architecture follows the same discipline as the rest of MLOps: separate environments, version assets, validate before promotion, observe production behavior, and close the loop. The difference is that the asset graph is wider.

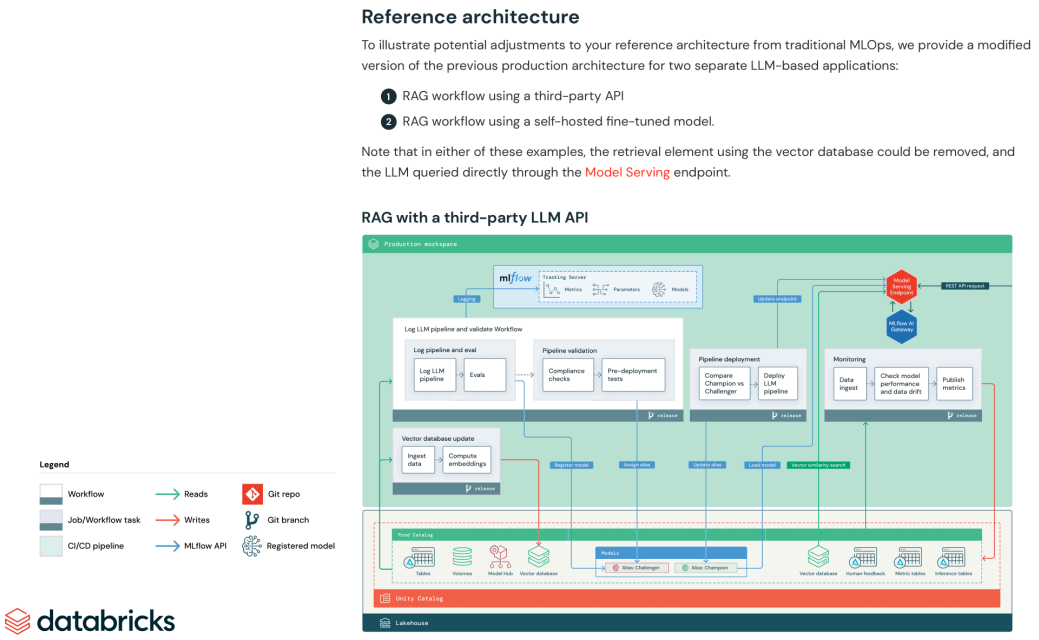

Figure 8: LLMOps RAG reference architecture from the Big Book of MLOps, p. 74.

RAG with a third-party LLM API

In this pattern, the application owns retrieval, prompt construction, evaluation, and governance, while the foundation model is accessed through an API. The key controls are endpoint selection, prompt versioning, retrieval evaluation, inference logging, and fallback.

RAG with a fine-tuned OSS model

In this pattern, the organization has more control over model behavior and hosting, but also more responsibility for training, evaluation, deployment, scaling, and maintenance.

Conclusion

The central lesson of the Big Book sequence is that MLOps is an operating system for ML work. It gives teams a repeatable way to move from exploration to production while preserving governance, lineage, validation, rollback, and monitoring.

For practical adoption, start with the smallest repeatable path:

1. Standardize experiments and run metadata.
2. Register models in a governed namespace.
3. Promote code through dev, staging, and production.
4. Use aliases for model lifecycle control.
5. Add validation gates before promotion.
6. Capture inference logs and monitor drift/quality.
7. Encode the pattern into templates and bundles.
8. Extend the same discipline to LLM prompts, retrieval, endpoints, and evaluation.

The point is not to make every project heavy. The point is to make the safe path the easiest path.

Sources used: Big Book pp. 5-78; MLOps workflows; CI/CD for ML; LLMOps workflows.

Glossary

Alias: A mutable pointer to an immutable model version, commonly used for lifecycle roles such as champion or challenger.

Asset Bundle: A declarative way to define Databricks jobs, pipelines, resources, and environment targets.

Champion: The current production model version or model configuration.

Challenger: A candidate model version being validated against the champion.

Experiment: A container for MLflow runs related to a model problem or project.

Inference table: A governed table of model or LLM request/response records used for monitoring, evaluation, and audit.

Lakehouse Monitoring: Databricks monitoring capability for profile, drift, and model-quality metrics over governed data.

MLOps Stacks: An opinionated Databricks project template built on Asset Bundles for production-oriented ML projects.

Model version: An immutable registered model artifact plus metadata.

AutoML: Automated model development tooling used to generate baselines, candidate models, and reproducible trial artifacts during exploration.

AutoGluon: An open source AutoML library that can train, ensemble, and serve candidate models using a compact `Dataset()`, `fit()`, and `predict()` workflow.

RAG: Retrieval augmented generation, a pattern that retrieves relevant context and passes it to an LLM to ground generated output.

Unity Catalog: Databricks governance layer for data and AI assets, including registered models.

Bibliography and Local Source Index

The source manifest is the authoritative list of downloaded references: sources/source-manifest.json.

Primary local references:

- The Big Book of MLOps, 2nd Edition
- MLOps Questions (1).pdf

Archived web references:

- Manage model lifecycle in Unity Catalog
- MLflow on Databricks
- Big Book of MLOps download page
- Model deployment patterns

- DABs for MLOps Stacks
- MLOps Stacks
- MLOps workflows
- CI/CD for ML
- Declarative Automation Bundles
- Databricks mlops-stacks GitHub archive
- MLOps Gym Crawl
- Provisioned throughput Foundation Model APIs
- Supported foundation models
- AI governance with Unity AI Gateway
- Databricks AI Search
- Create AI Search endpoints and indexes
- Foundation Model APIs
- Deploy models using Model Serving
- Data profiling / Lakehouse Monitoring
- Monitor model services using inference tables
- LLMOps workflows
- MLOps workflows on Databricks AWS
- Model deployment patterns on Databricks AWS
- MLOps Stacks on Databricks AWS
- LLMOps workflows on Databricks AWS
- CI/CD for ML on Databricks AWS
- Machine learning with AutoGluon, an open source AutoML library