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

MODEL DEPLOYMENT

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

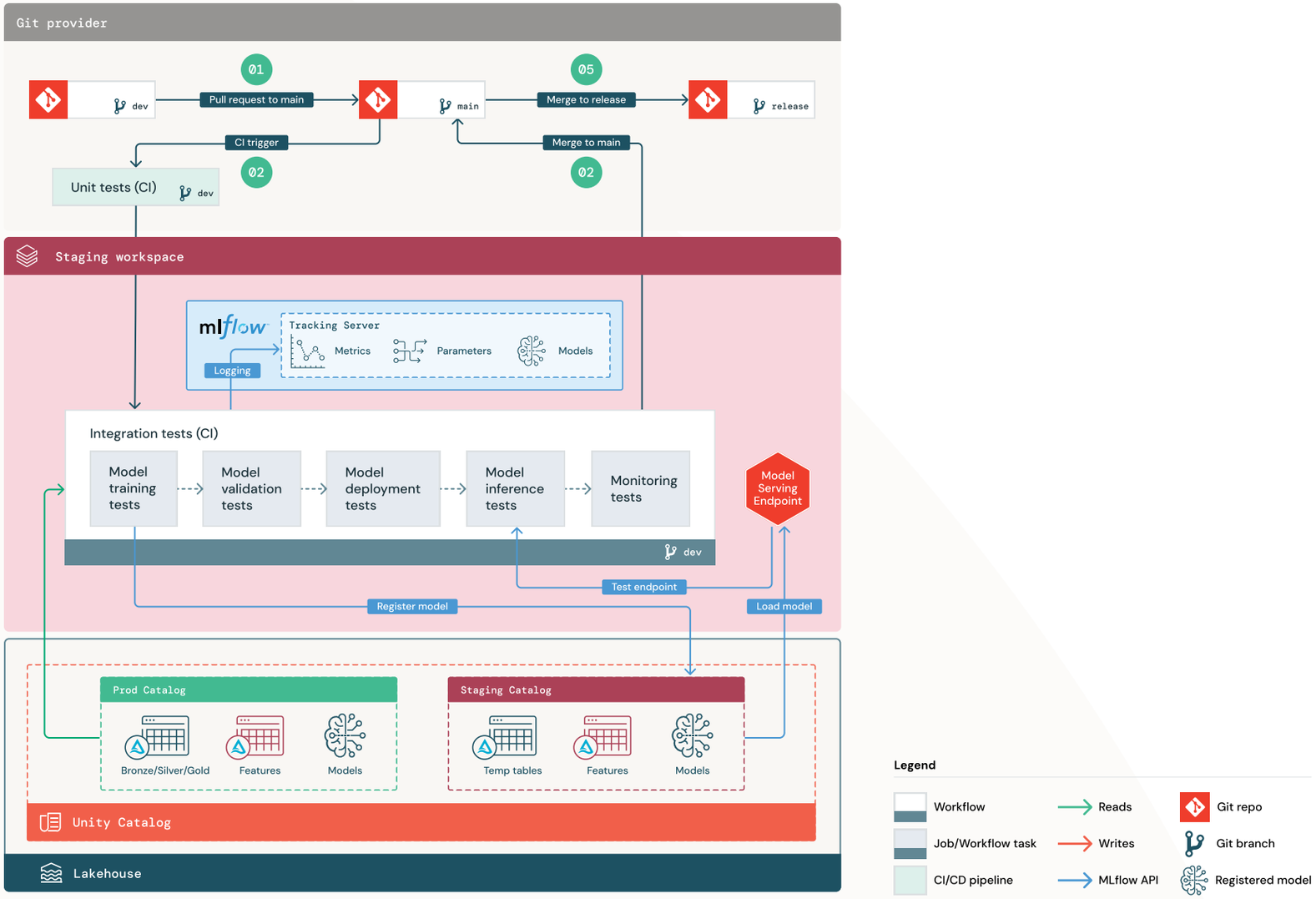
Commit code

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

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

Staging

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



databricks

Data

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

Merge code

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

- PULL REQUEST

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

- UNIT TESTS (CI)

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

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

Integration tests (CI)

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

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

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

Merge

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

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

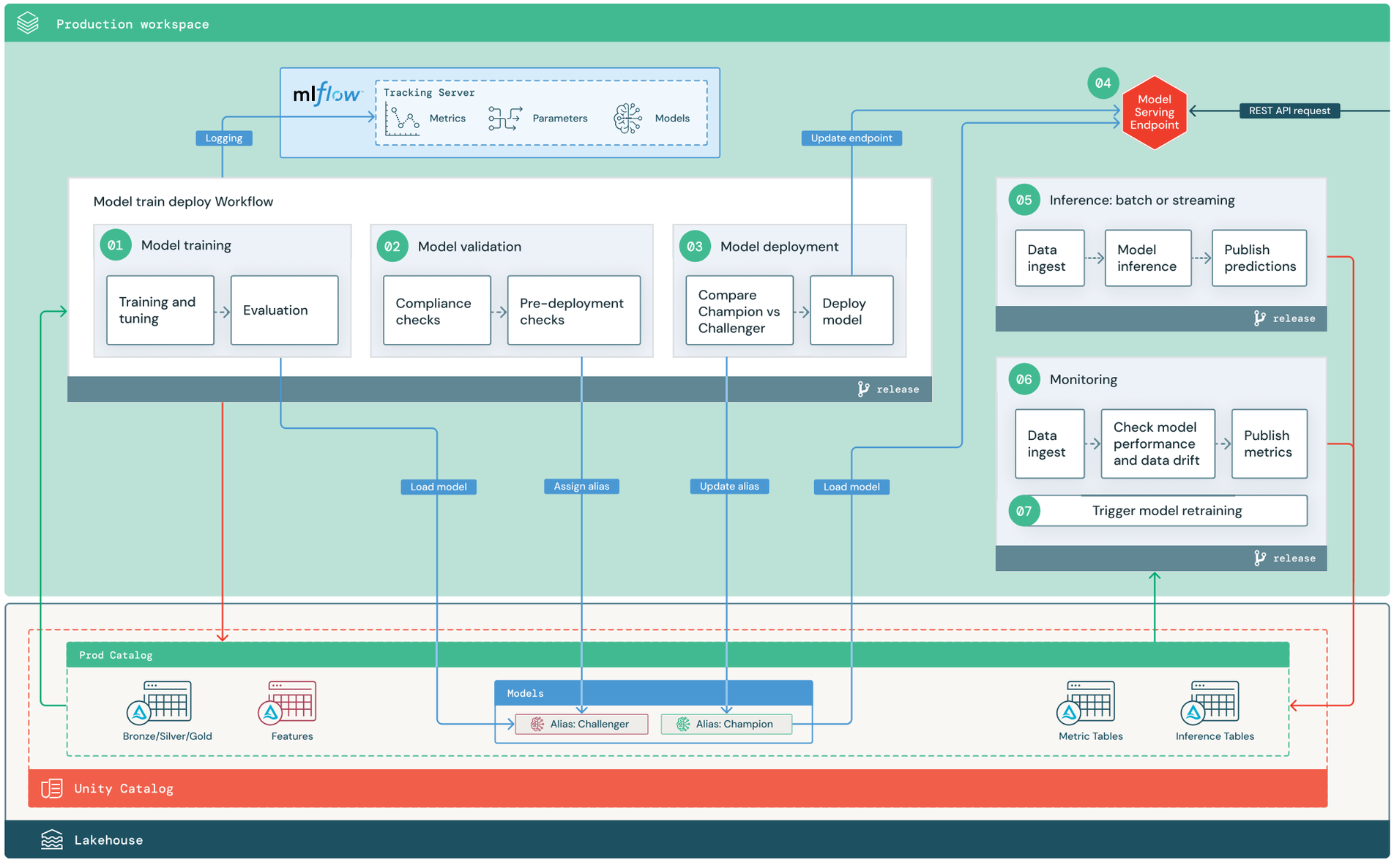
Cut release branch

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

Production

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

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



Legend

Workflow
Job/Workflow task
CI/CD pipeline
Reads
Writes
MLflow API
Git repo
Git branch
Registered model

databricks

Model training

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

- **TRAINING AND TUNING**

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

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

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

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

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

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

- **EVALUATION**

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

- **REGISTER MODEL**

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

Model validation

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

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

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

Model deployment

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

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

- **COMPARE "CHALLENGER" VS. "CHAMPION"**

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

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

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

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

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

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

DEPLOY MODEL

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

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

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

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

Model Serving

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

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

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

Inference: batch or streaming

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

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

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

Lakehouse Monitoring

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

- DATA INGESTION

  This pipeline reads in logs from batch, streaming or online inference.

- CHECK ACCURACY AND DATA DRIFT

  The pipeline then computes metrics about the input data, the model's predictions and the infrastructure performance. Metrics that measure statistical properties are generally chosen by data scientists during development, whereas metrics for infrastructure are generally chosen by ML engineers. Note that custom metrics can be defined and monitored with Lakehouse Monitoring.

- PUBLISH METRICS AND SET UP ALERTS

  The pipeline writes to Lakehouse tables in the prod catalog for analysis and reporting. These tables should be readable into the development environment to allow data scientists to perform granular analysis if required. Tools such as Databricks SQL are used to produce monitoring dashboards, allowing for health checks and diagnostics. The monitoring job or the dashboarding tool issues notifications when health metrics surpass defined thresholds.

- TRIGGER MODEL TRAINING

  When the model monitoring metrics indicate performance issues, or when a model inevitably becomes out of date, the data scientist may need to return to the development environment and develop a new model version. SQL alerts can be used to notify data scientists when this happens.

Retraining

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