DATA+AI
SUMMIT 2022

MLOps on Databricks

A how-to guide

ORGANIZED BY databricks

Outline

Part I

- Introduction
  - What is MLOps
  - Why care about MLOps?
  - People and process
  - Guiding principles
- Fundamentals
  - Semantics of dev, staging, prod
  - ML deployment patterns
- Reference architectures

Part II

- The use case
- Overview of pipelines
- Tooling
  - DevOps
  - DataOps
  - ModelOps
- Project structure
- Demo
- Getting started

Part I

Introduction

- What is MLOps?
- Why care about MLOps
- People and process
- Guiding principles

Hardest Part of ML isn't ML, it's Data

"Hidden Technical Debt in Machine Learning Systems," Google NIPS 2015

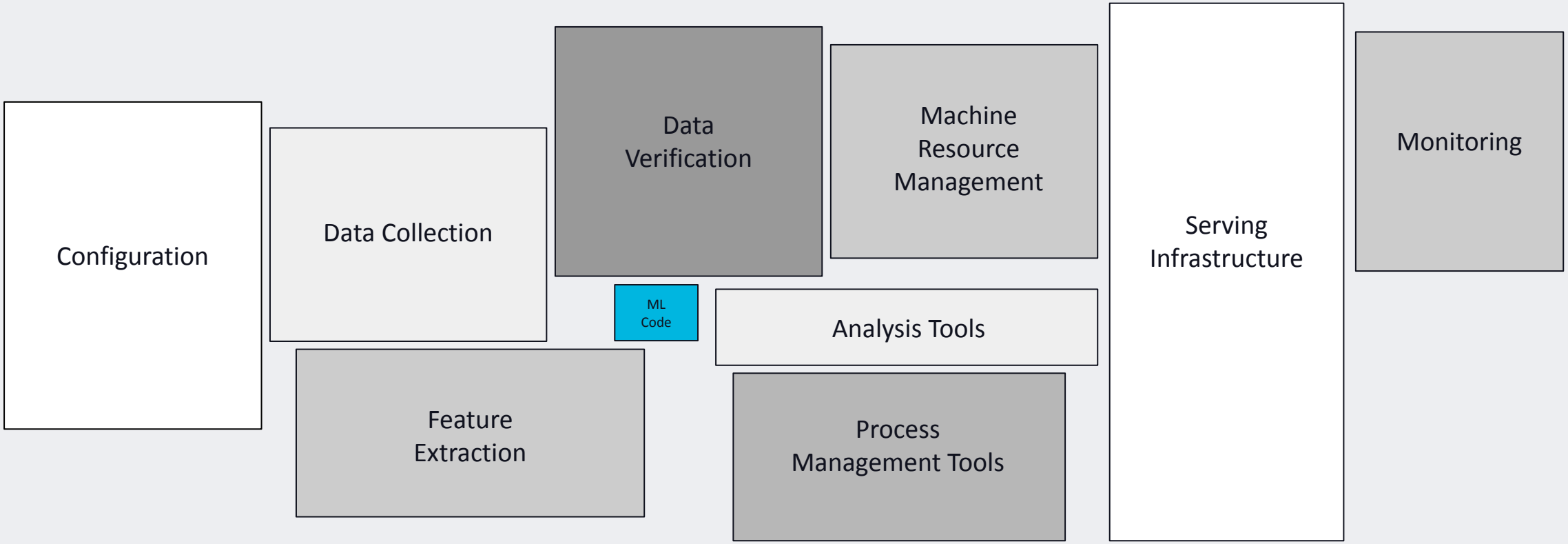


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small green box in the middle. The required surrounding infrastructure is vast and complex.

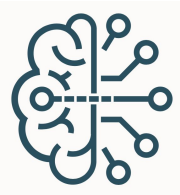
What is MLOps?

MLOps is a set of **processes and automation**

for **managing code, data, and models**

to **improve performance, stability and long-term efficiency** of ML systems

MLOps = DevOps + DataOps + ModelOps



Why should I care about MLOps?

MLOps helps you reduce risk

- Technical risk - poorly performing models, fragile infrastructure
- Compliance risk - violating regulatory or corporate policies

MLOps improves long-term efficiency through automation

- Catch errors before they hit production
- Avoid slow, manual processes

People and process

Business Stakeholder → Responsible for business value of the ML solution

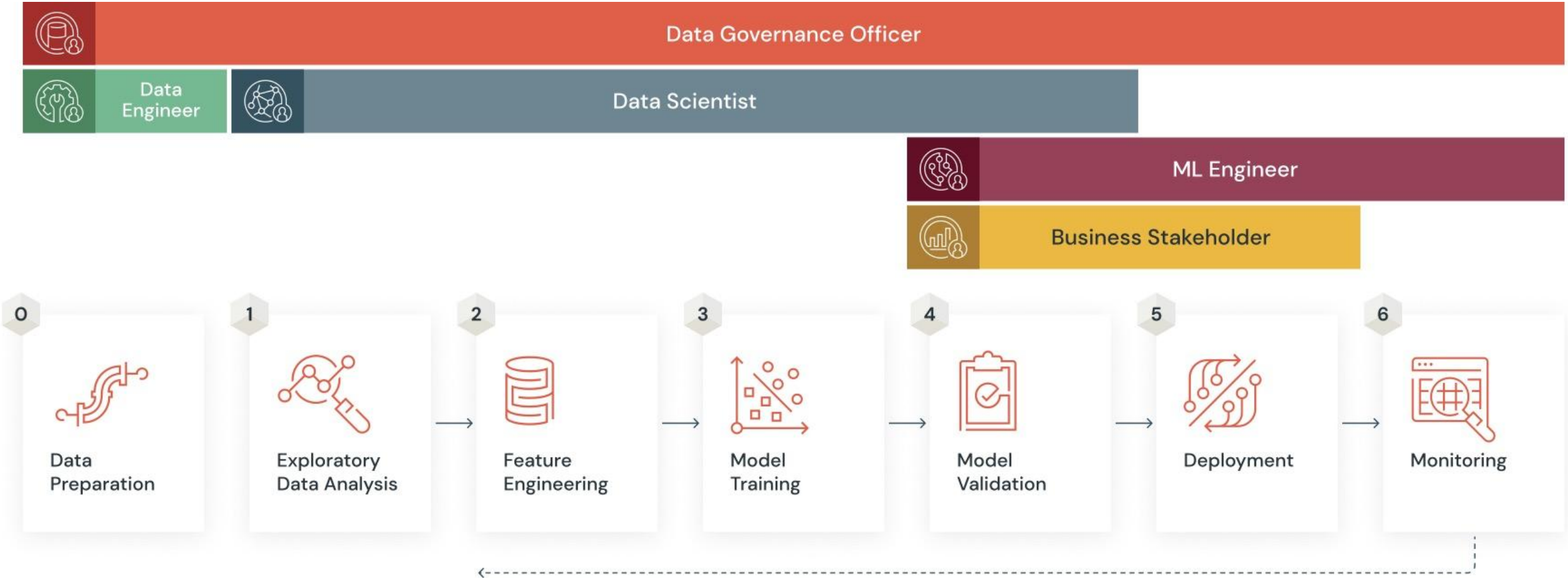
Data Engineer → Builds data pipelines

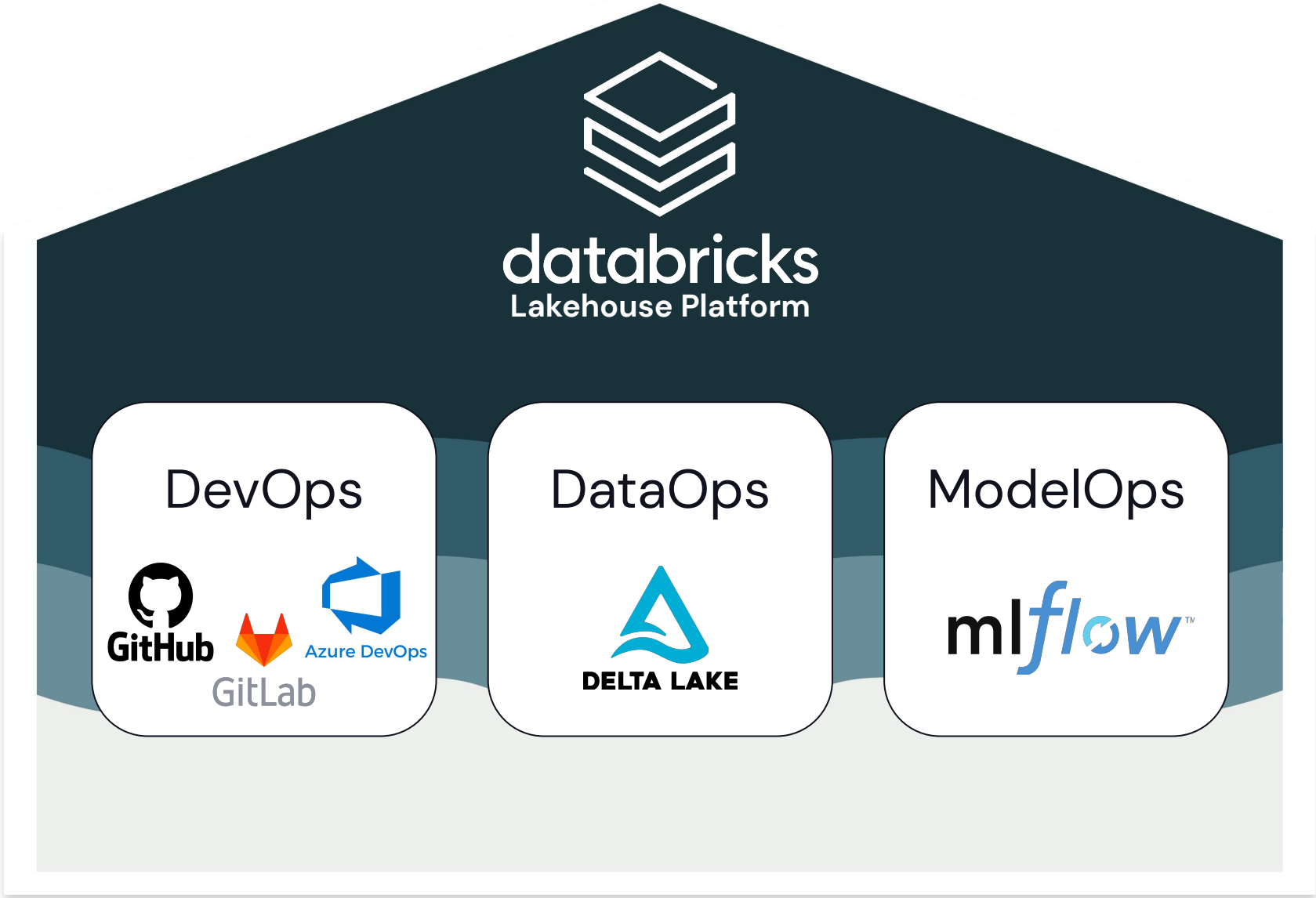
Data Scientist → Translates business problem; trains, tunes model

ML Engineer → Deploys ML model to production

Data Governance Officer → Responsible for data governance and compliance

People and process





Data-centric ML platform

Guiding principles

Always keep your business goals in mind

Take a data-centric approach to machine learning

Implement MLOps in a modular fashion

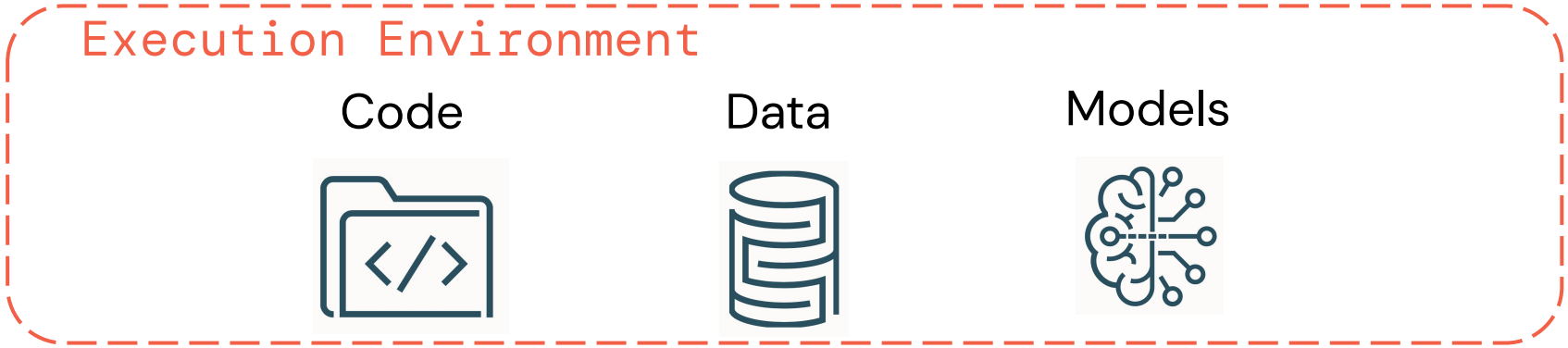
Process should guide automation

Fundamentals of MLOps

- Semantics of dev, staging and prod
- Deployment patterns

Semantics of dev, staging, and prod

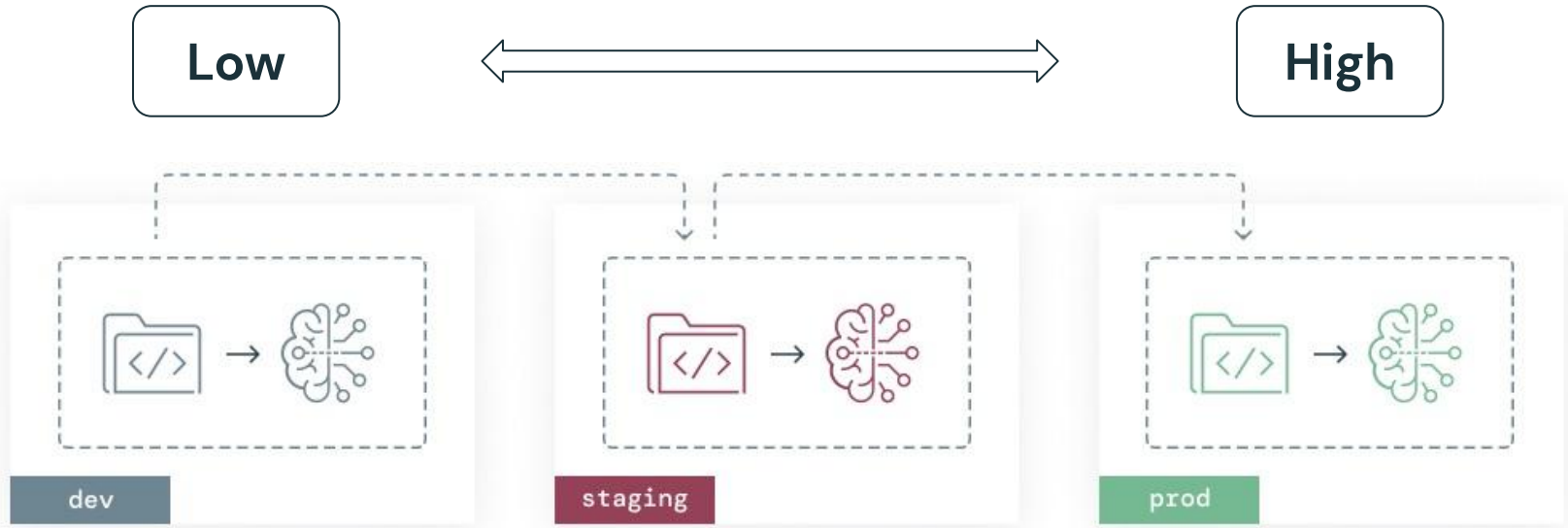
ML Workflow Assets:



Assets need to be:

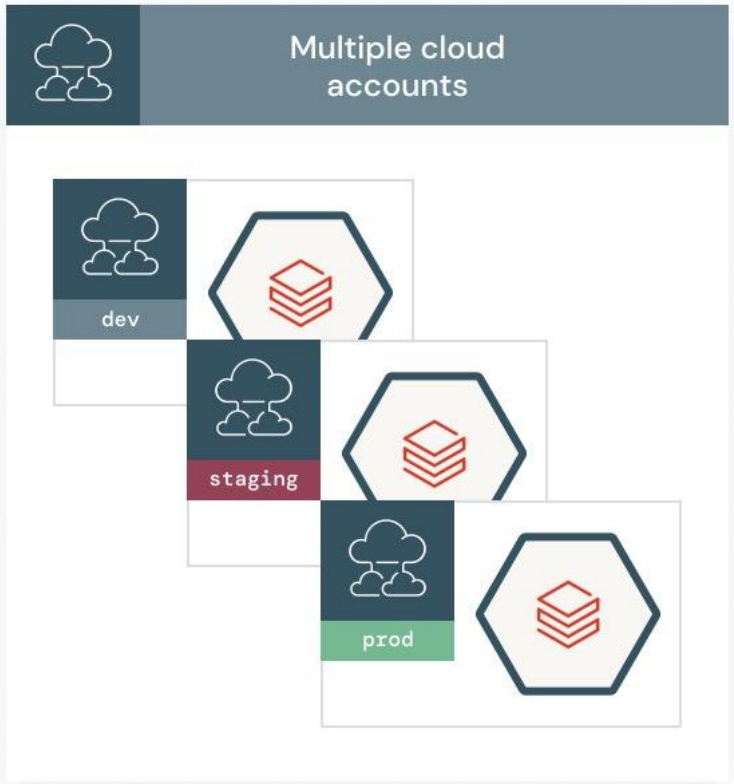
Dev vs. staging vs. prod

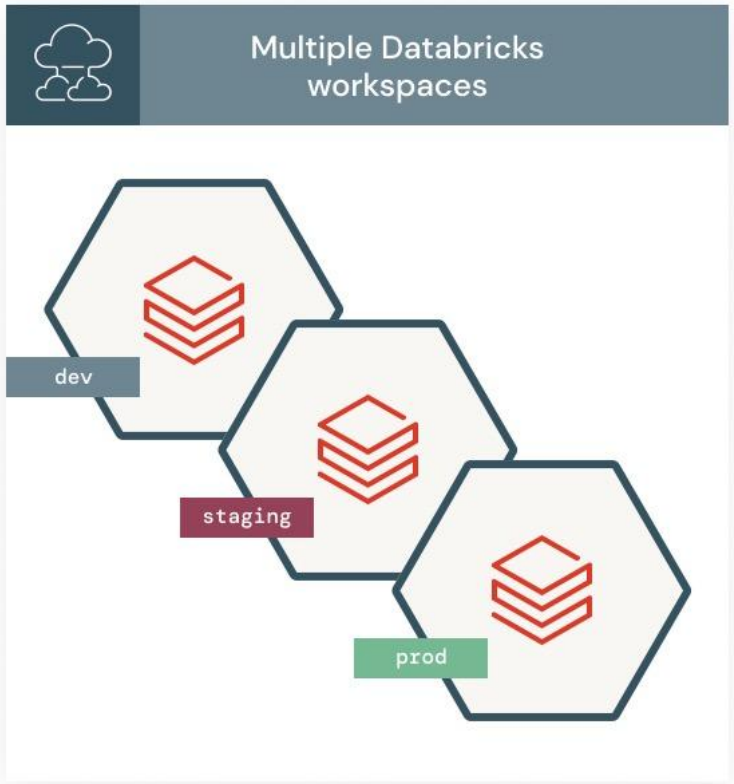
Level of trust, quality and testing:

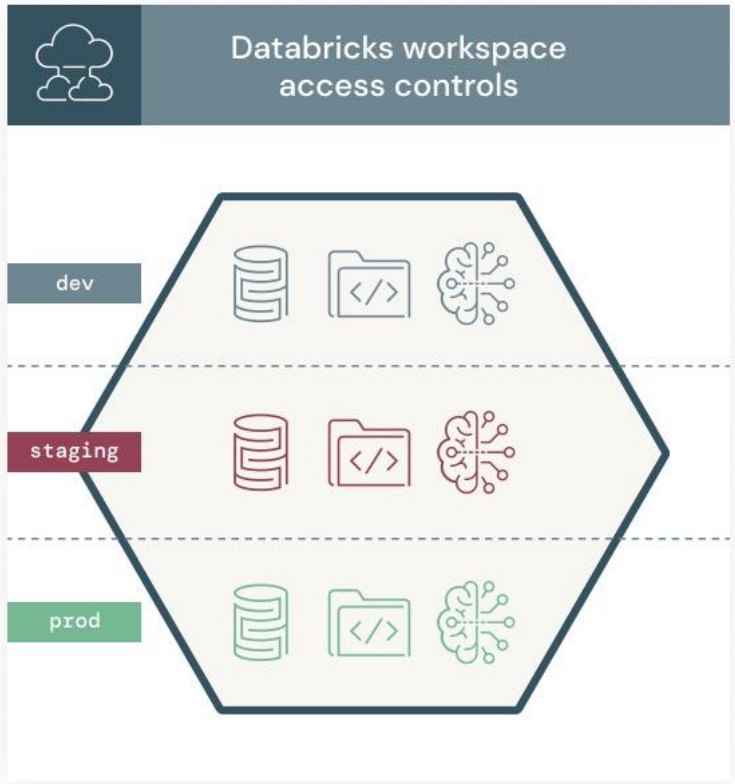


Openness of access:

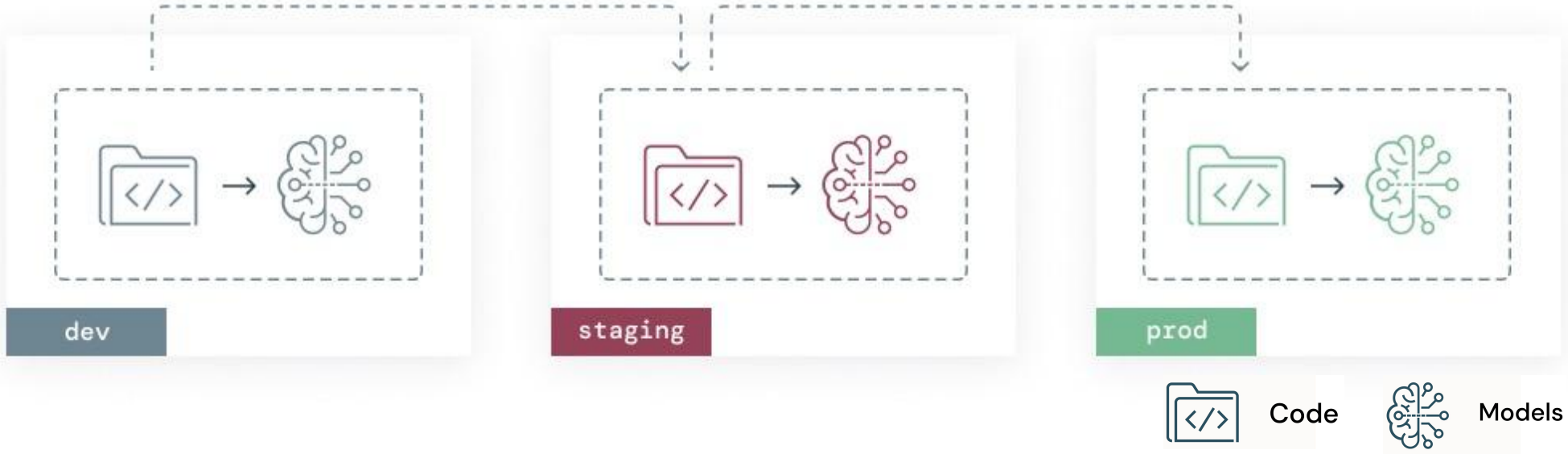
Environments: dev vs. staging vs. prod







Model vs. code lifecycles



Model and code lifecycles often operate asynchronously

- Weekly fraud detection model
  - Model update; no code change
- Computer vision model, large language model
  - No model update; code change

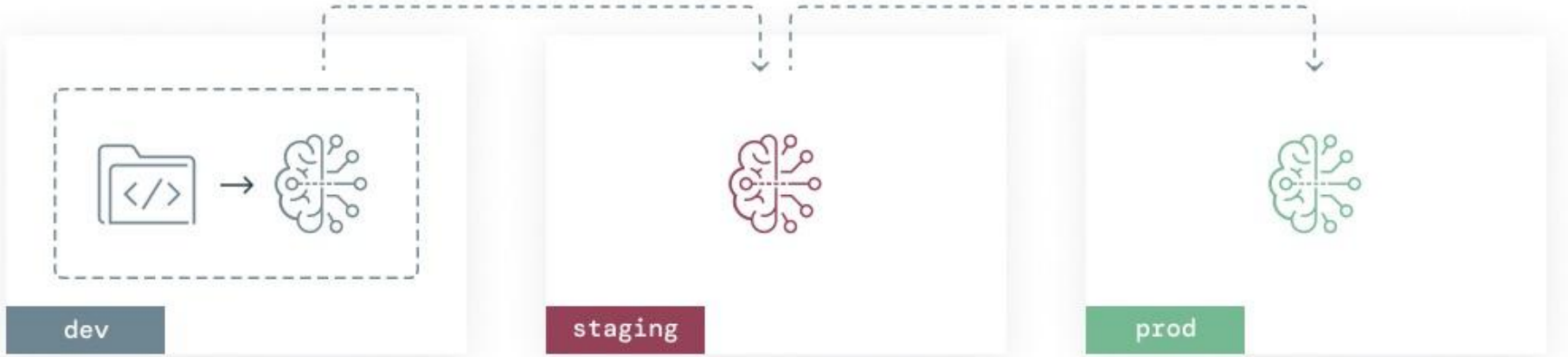
→ mlflow™

Dev vs. staging vs. prod: managing assets

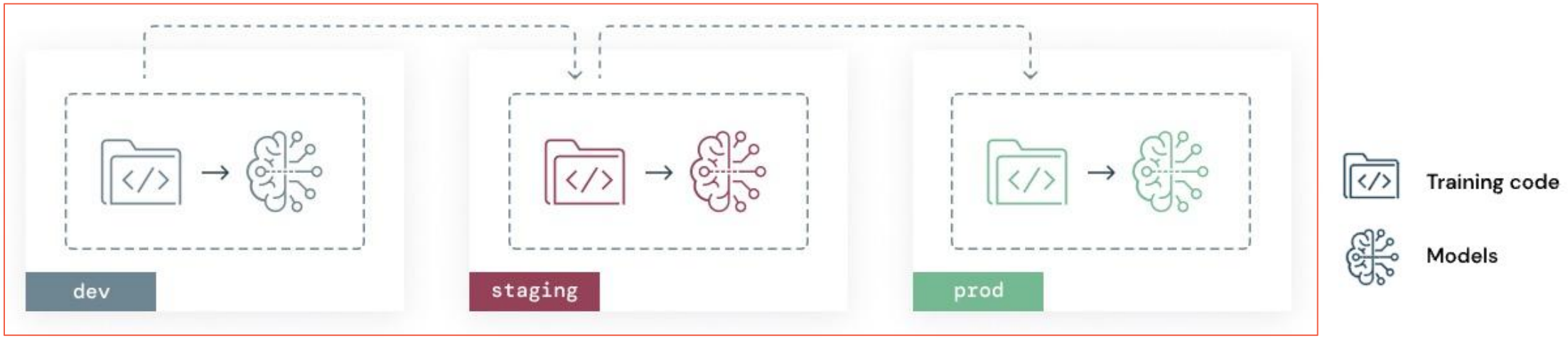
ASSET	SEMANTICS	SEPARATED BY
Execution environments	Labeled according to where development, testing and connections with production systems happen	Cloud provider and Databricks Workspace access controls
Models	Labeled according to model lifecycle phase	MLflow access controls or cloud storage permissions
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

ML deployment patterns

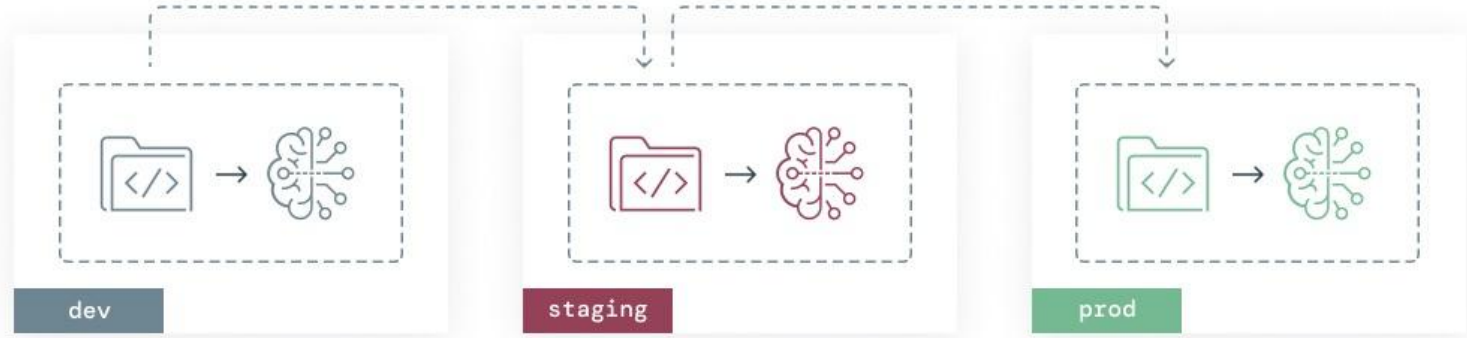
Deploy Models

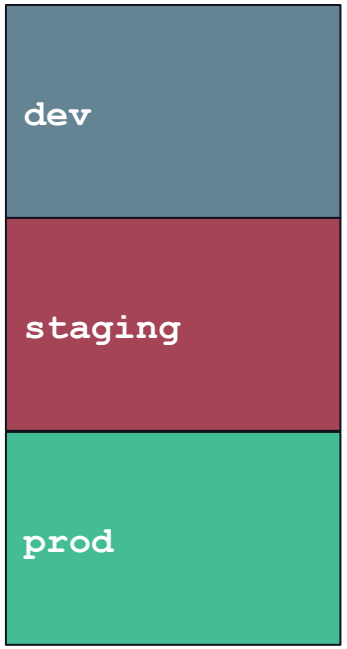


Deploy Code



Deploy code process



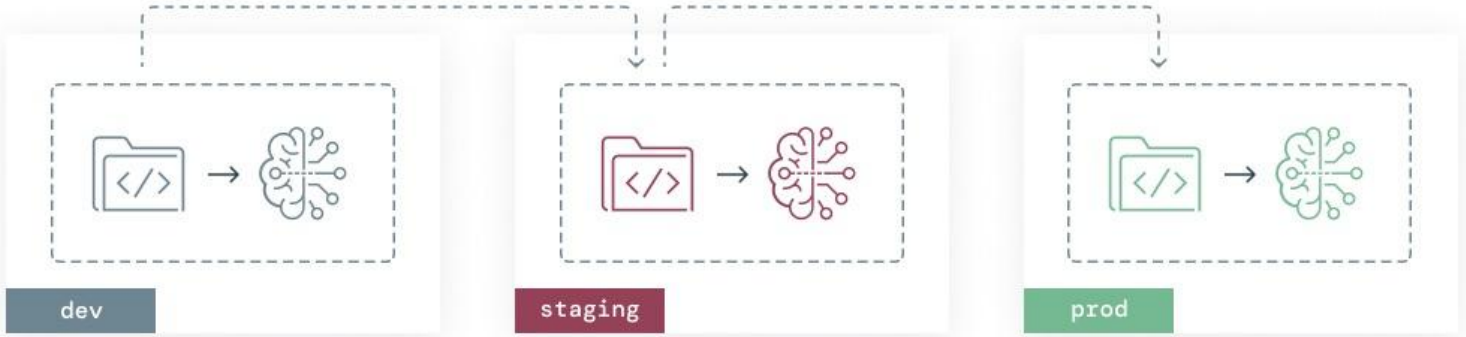


Develop training code
Develop ancillary code
→ Promote code.

✔ Test model training code on subset of data
✔ Test ancillary code
→ Promote code.

✔ Train model on production data
✔ Test model
→ Deploy model
→ Deploy ancillary code

Benefits of deploy code

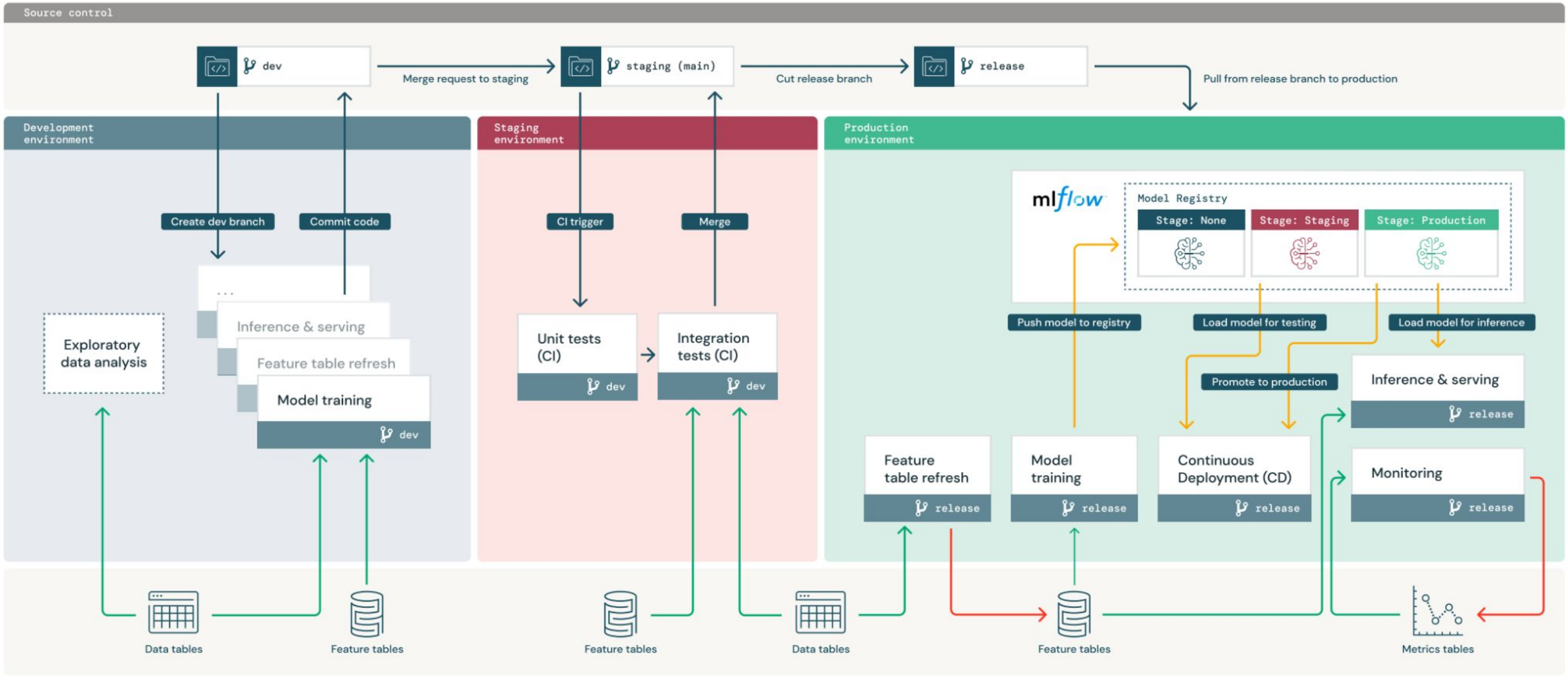


Automation	↑ Supports automated retraining in locked-down env.
Data access control	↑ Only prod env needs read access to prod training data.
Reproducible models	↑ Eng control over training env, which helps to simplify reproducibility.
Support for large projects	↑ This pattern forces the DS team to use modular code and iterative testing, which helps with coordination and development in larger projects.
Data science familiarity	↓ DS team must learn to write & hand off modular code to Eng.
Eng setup & maintenance	↓ Requires CI/CD infra for unit and integration tests, even for one-off models.

Reference Architectures

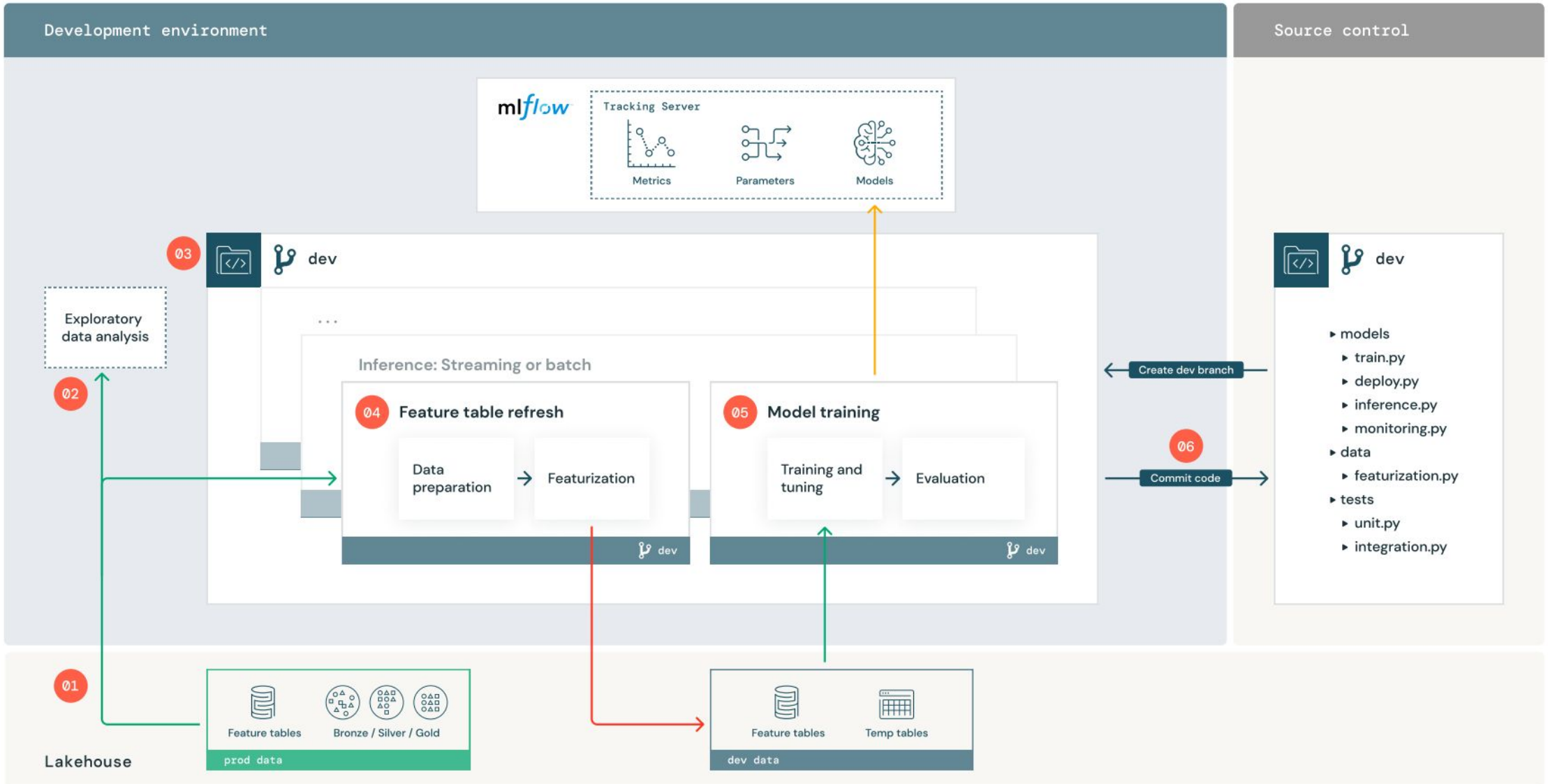
- Multi-environment view
- Dev
- Staging
- Production

Overview

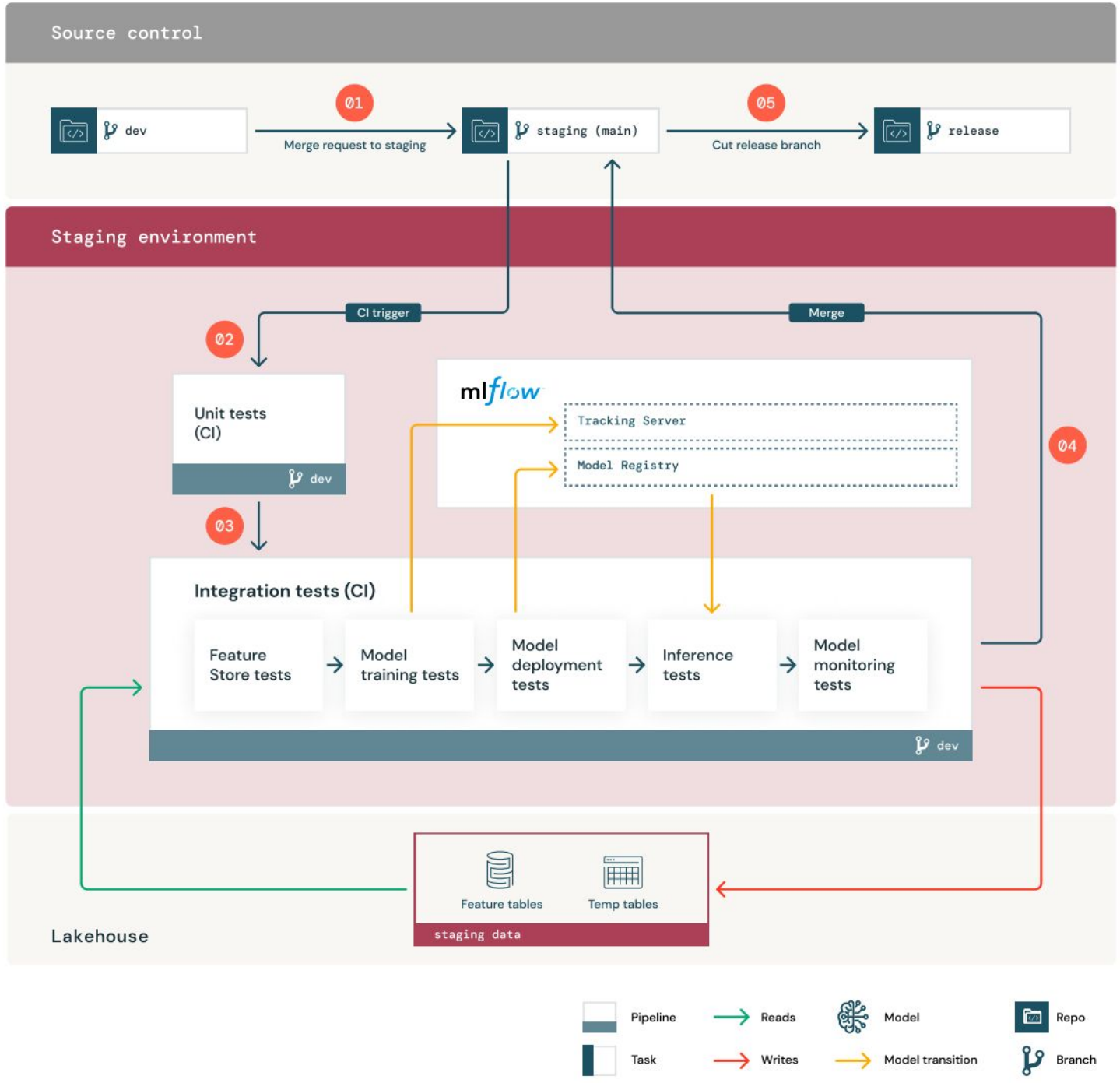


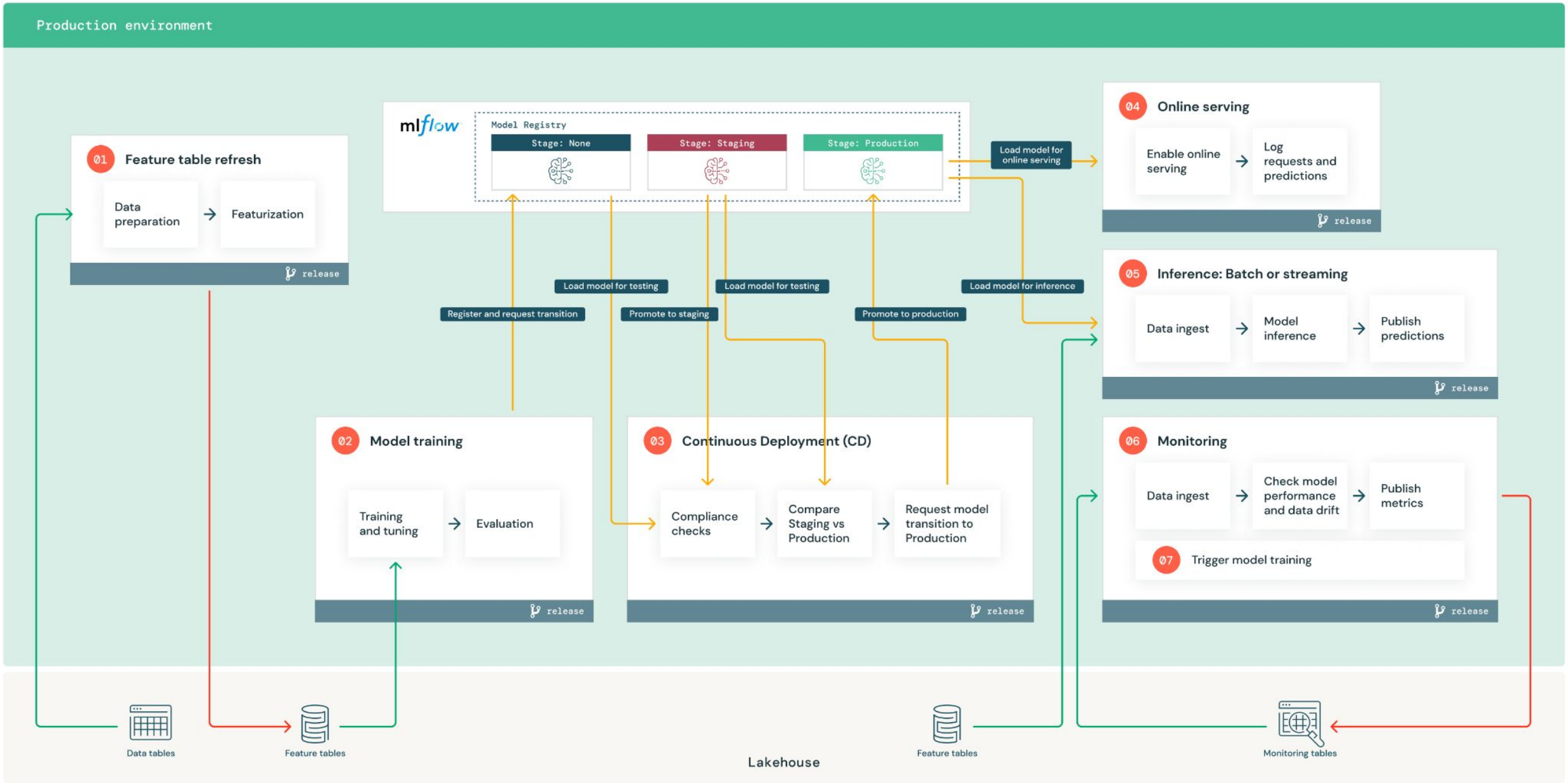
Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch

Dev


Development environment
Source control
mlflow
Tracking Server
Metrics
Parameters
Models
03
dev
Exploratory data analysis
02
...
Inference: Streaming or batch
04 Feature table refresh
Data preparation
Featurization
dev
05 Model training
Training and tuning
Evaluation
dev
Create dev branch
06
Commit code
dev
models
train.py
deploy.py
inference.py
monitoring.py
data
featurization.py
tests
unit.py
integration.py
01
Feature tables
Bronze / Silver / Gold
prod data
Lakehouse
Feature tables
Temp tables
dev data
Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch

Staging


Source control
dev
01
Merge request to staging
staging (main)
05
Cut release branch
release
Staging environment
CI trigger
Merge
02
Unit tests (CI)
dev
mlflow
Tracking Server
Model Registry
04
03
Integration tests (CI)
Feature Store tests
Model training tests
Model deployment tests
Inference tests
Model monitoring tests
dev
Feature tables
Temp tables
staging data
Lakehouse
Pipeline
Reads
Model
Repo
Task
Writes
Model transition
Branch


Production environment
mlflow
Model Registry
Stage: None
Stage: Staging
Stage: Production
01 Feature table refresh
Data preparation
Featurization
release
04 Online serving
Enable online serving
Log requests and predictions
release
Load model for online serving
Load model for testing
Load model for testing
Load model for inference
Register and request transition
Promote to staging
Promote to production
05 Inference: Batch or streaming
Data ingest
Model inference
Publish predictions
release
02 Model training
Training and tuning
Evaluation
release
03 Continuous Deployment (CD)
Compliance checks
Compare Staging vs Production
Request model transition to Production
release
06 Monitoring
Data ingest
Check model performance and data drift
Publish metrics
07 Trigger model training
release
Data tables
Feature tables
Lakehouse
Feature tables
Monitoring tables
Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch

Part II

Outline

Part I

- Introduction
  - People and process
  - Goals of MLOps
  - Guiding principles
- Fundamentals
  - Semantics of dev, staging, prod
  - Moving to production
- Reference Architectures

Part II

- The use case
- Overview of pipelines
- Tooling
  - DevOps
  - DataOps
  - ModelOps
- Project Structure
- Demo
- Getting started

The use case

Churn prediction problem

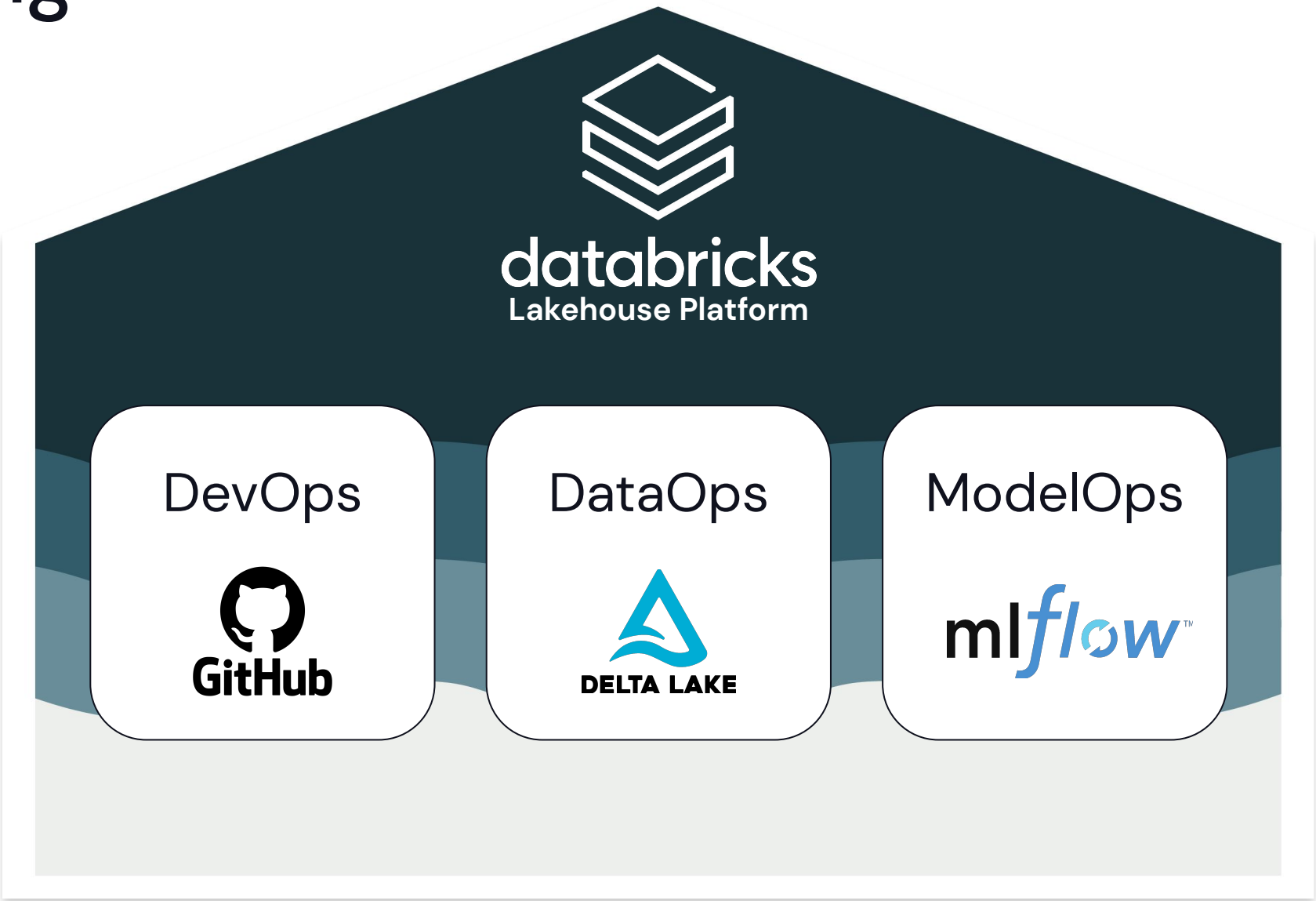
IBM telco customer churn dataset

- Fictional telco company providing phone and internet services
- 7,043 customers
- Churn label column indicates whether customer has left within the last month
- Demographic, and contract-based information

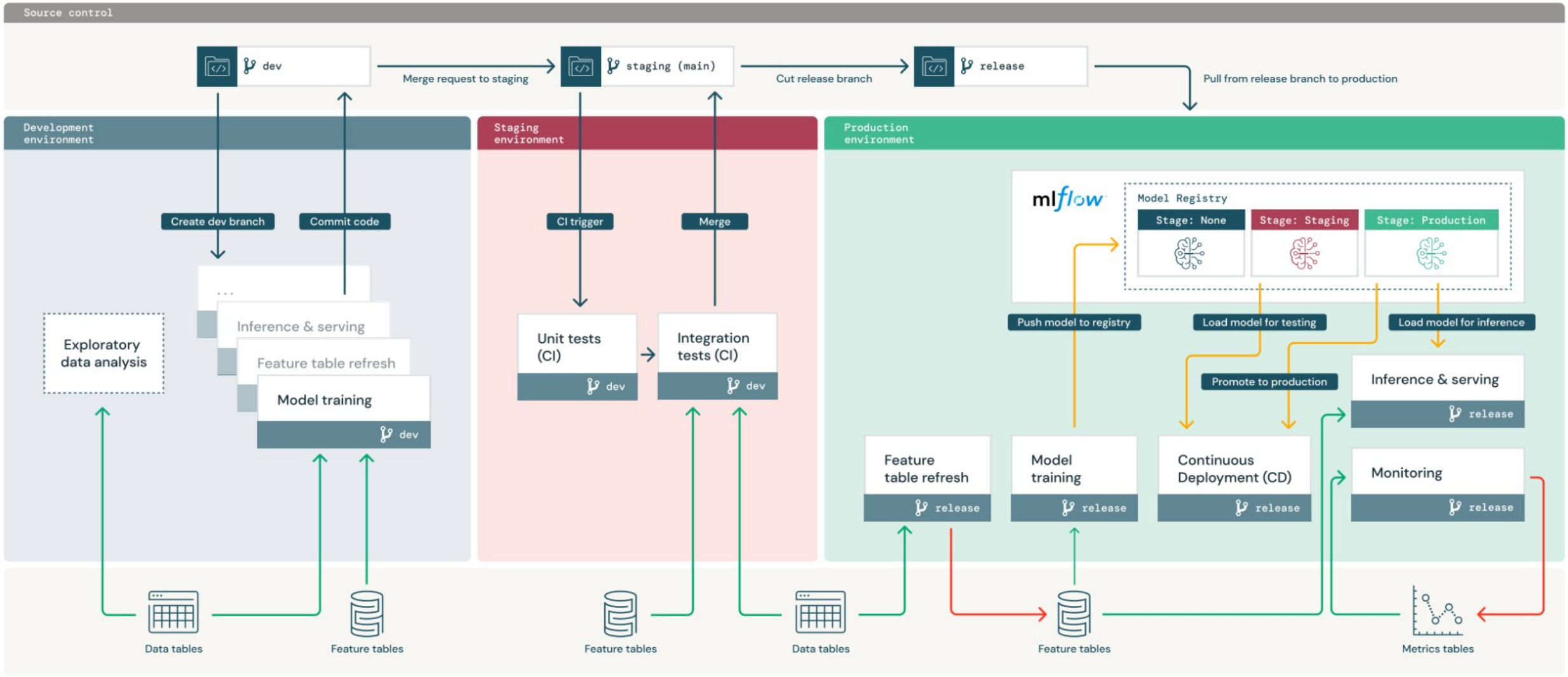
Goal

Create MLOps workflow to continuously deliver
a churn prediction model to production

Tooling


databricks
Lakehouse Platform
DevOps
GitHub
DataOps
DELTA LAKE
ModelOps
mlflow

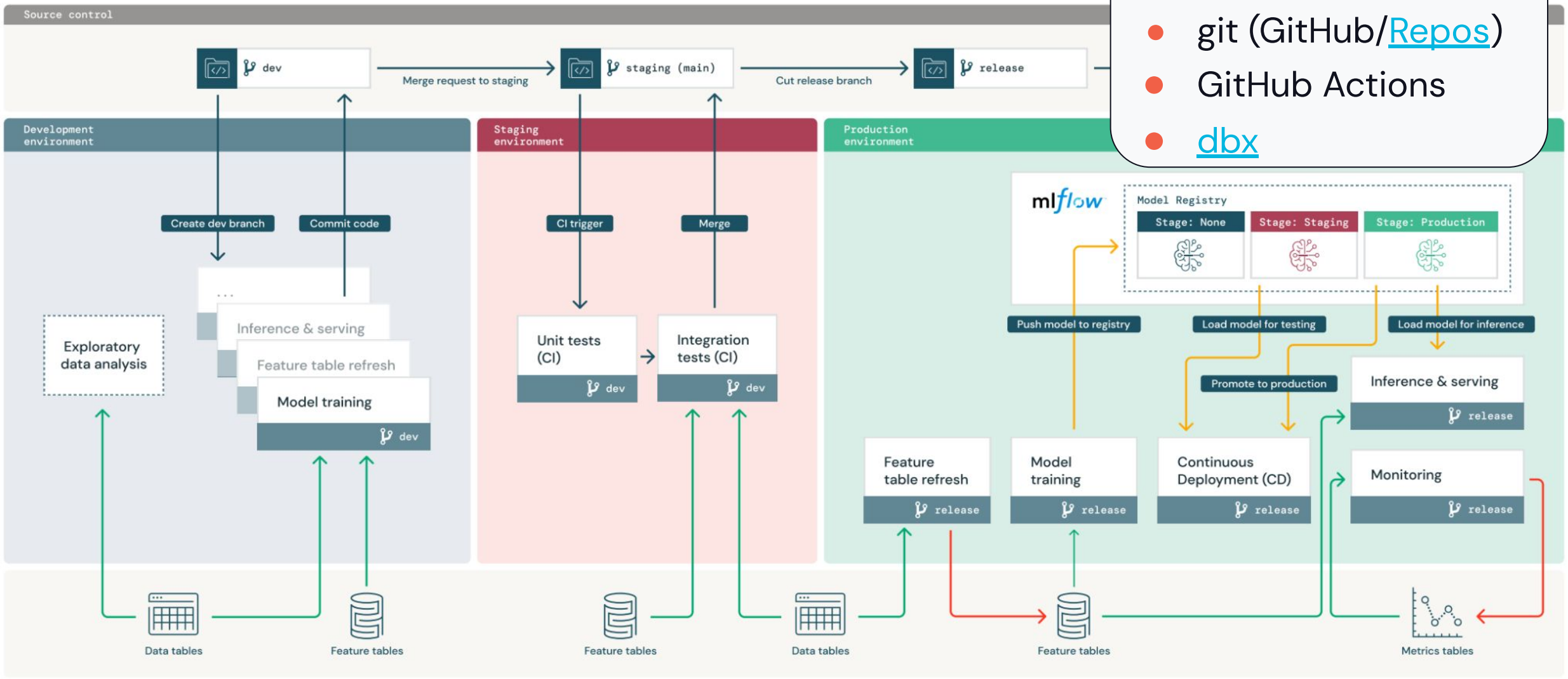
Tooling: Architecture Perspective



Tooling: Architecture Perspective

DevOps

- git (GitHub/Repos)
- GitHub Actions
- dbx

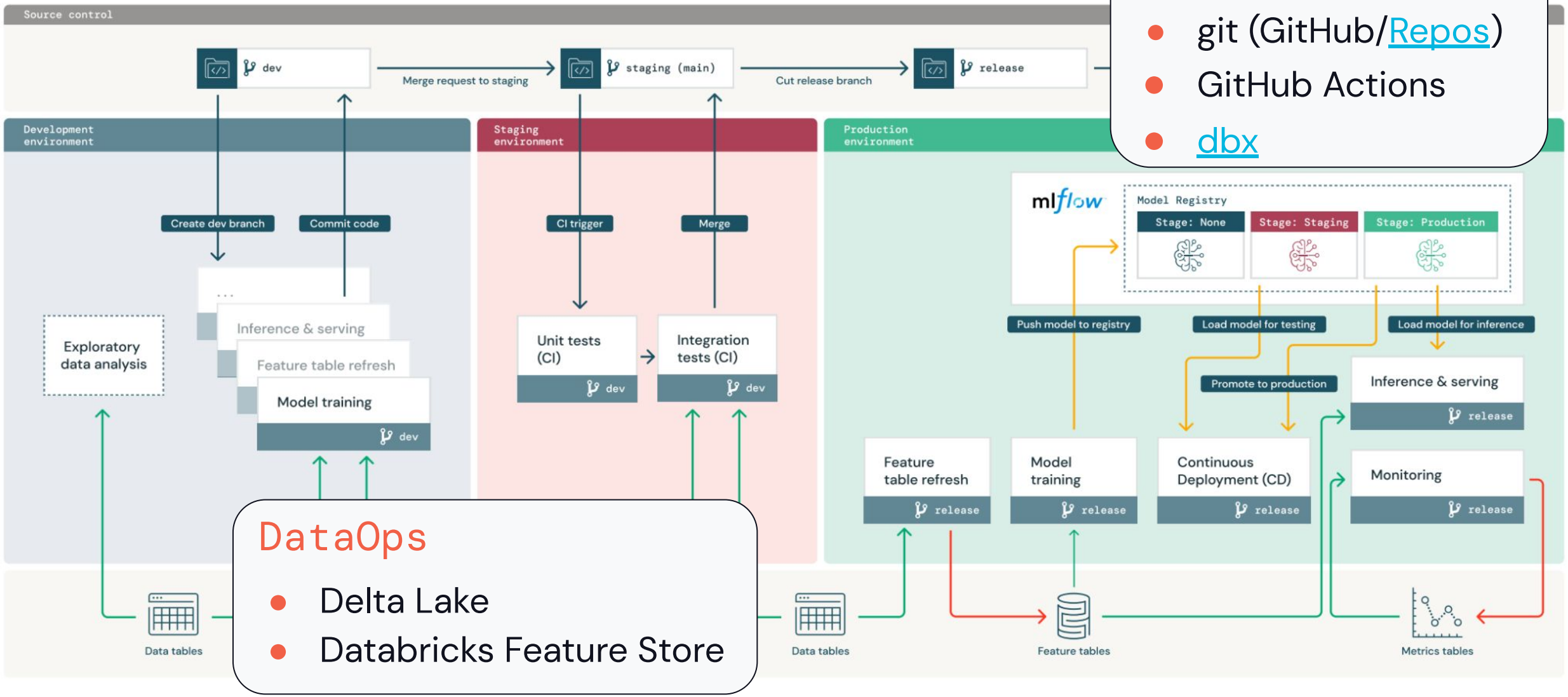


Pipeline · Task · Reads · Writes · Model · Model transition · Repo · Branch

Tooling: Architecture Perspective

DevOps

- git (GitHub/Repos)
- GitHub Actions
- dbx

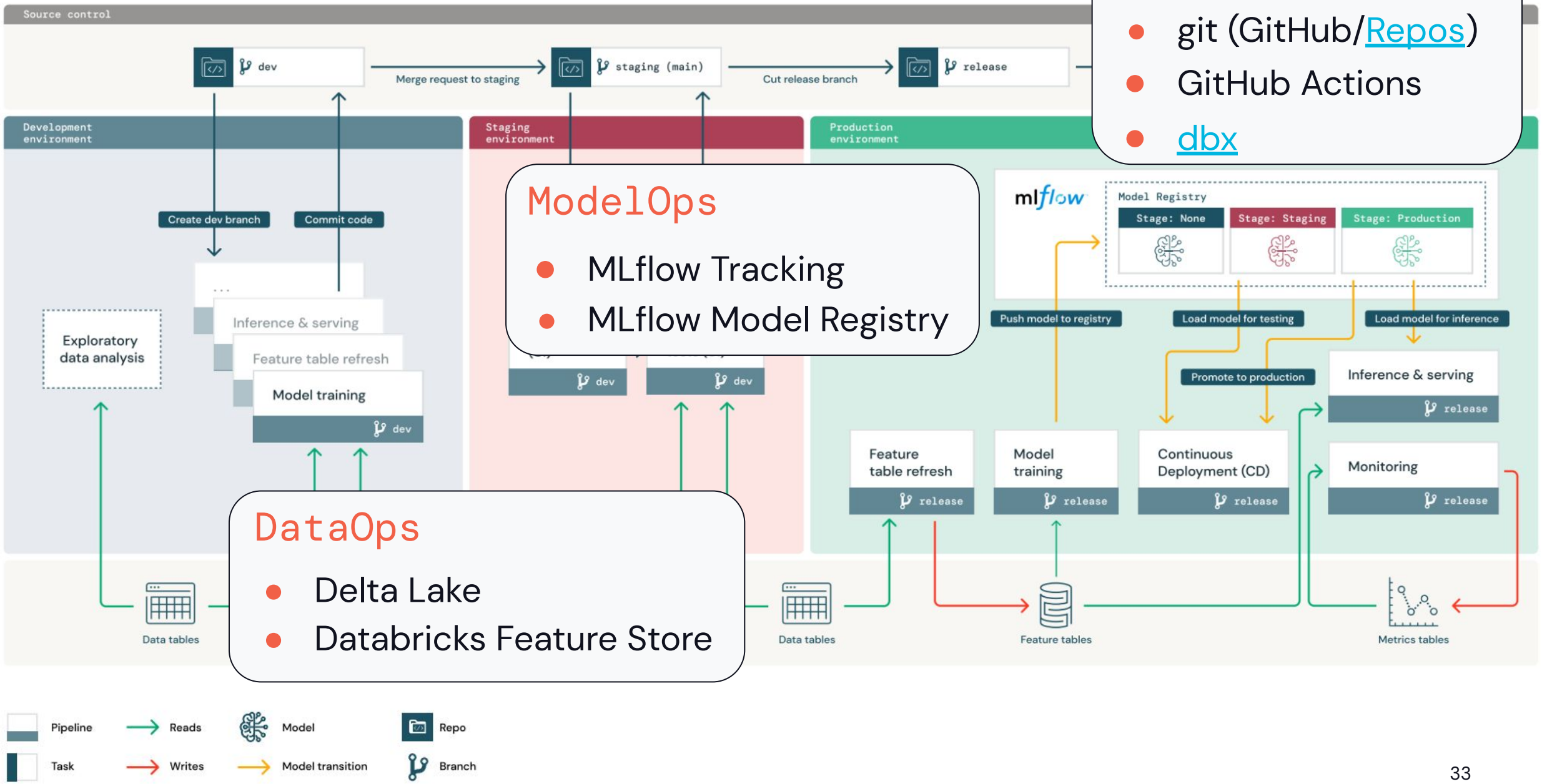


DataOps

- Delta Lake
- Databricks Feature Store

Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch

Tooling: Architecture Perspective



DevOps

- git (GitHub/Repos)
- GitHub Actions
- dbx

ModelOps

- MLflow Tracking
- MLflow Model Registry

DataOps

- Delta Lake
- Databricks Feature Store

Tooling

DevOps

- dbx
  - Simplifies IDE & Databricks integration
  - Job deployment
  - Python Template
- GitHub Actions
  - Automate CI/CD workflows

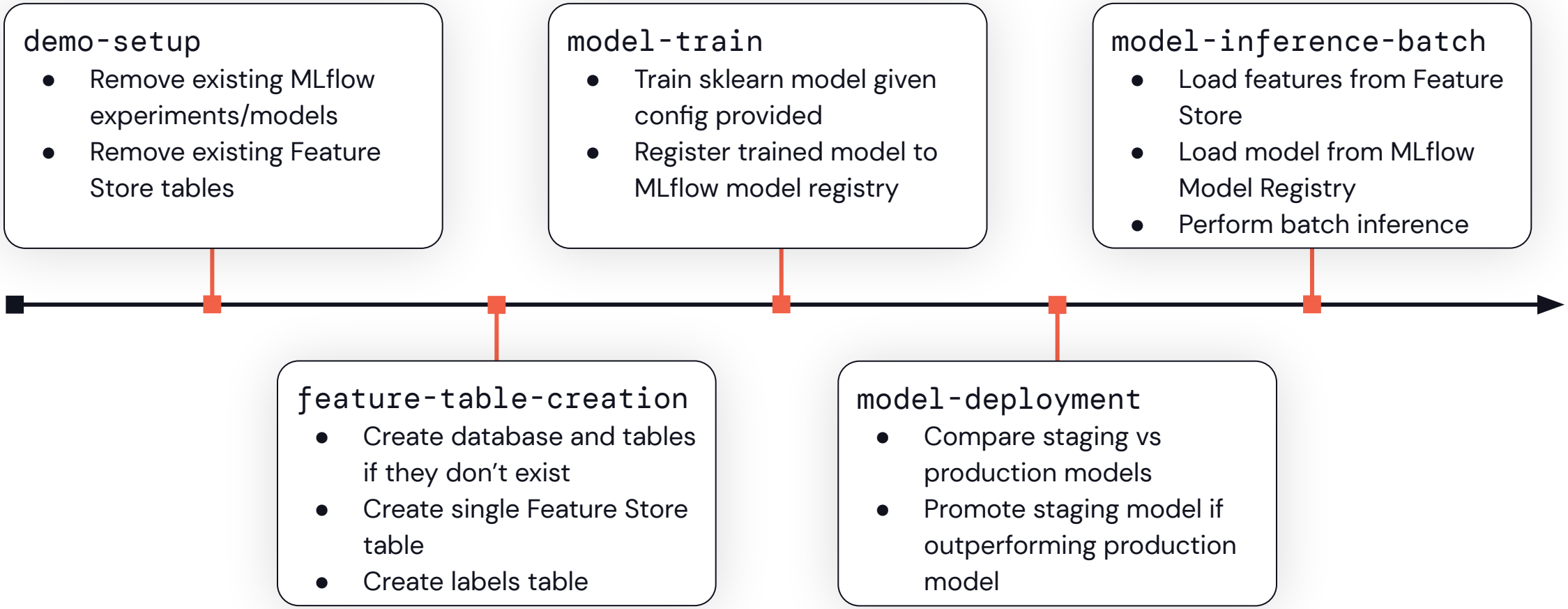
DataOps

- Delta Lake
  - Used as raw data input
- Databricks Feature Store
  - Centralized repository of features
  - Lineage tracking

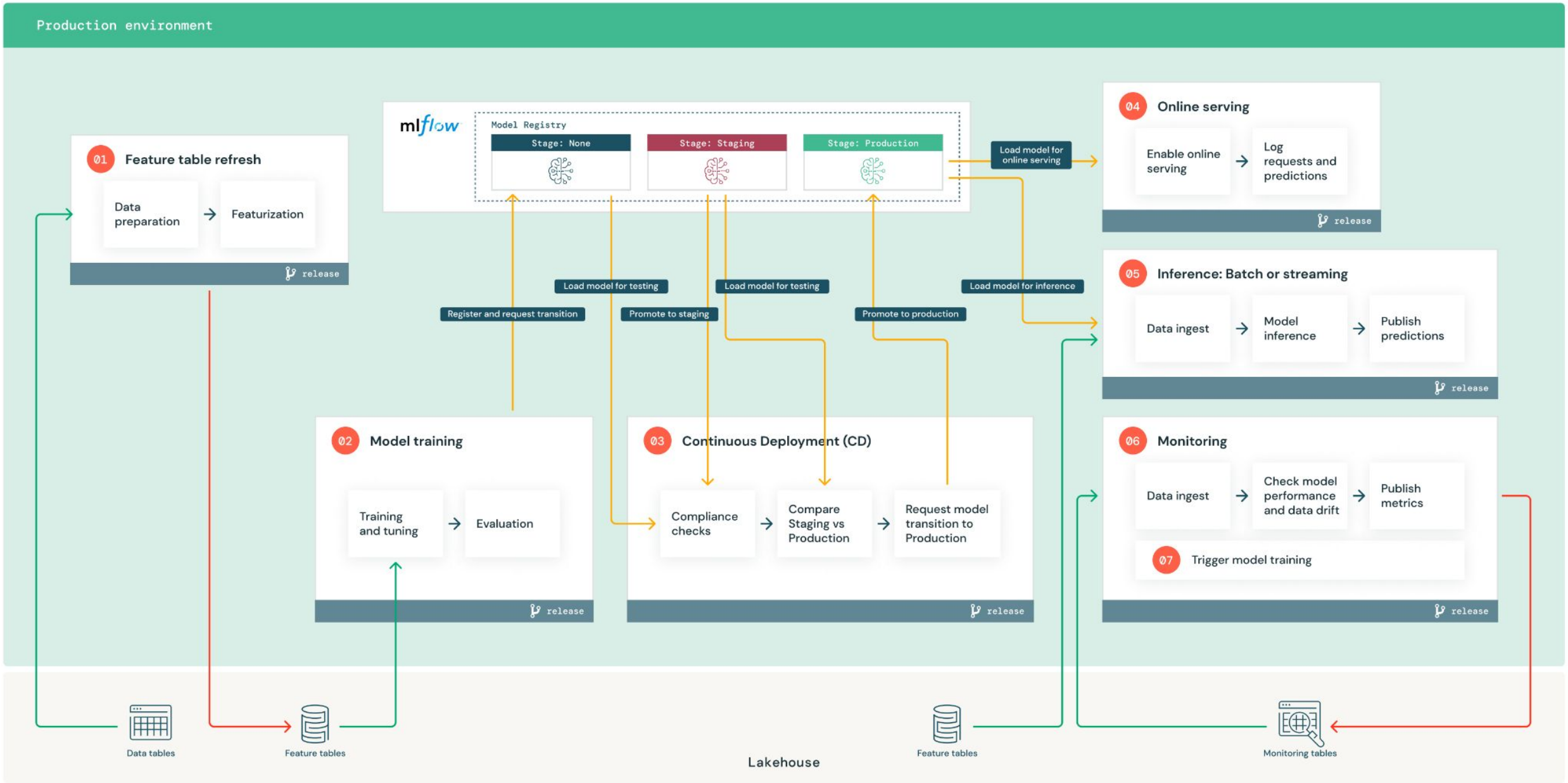
ModelOps

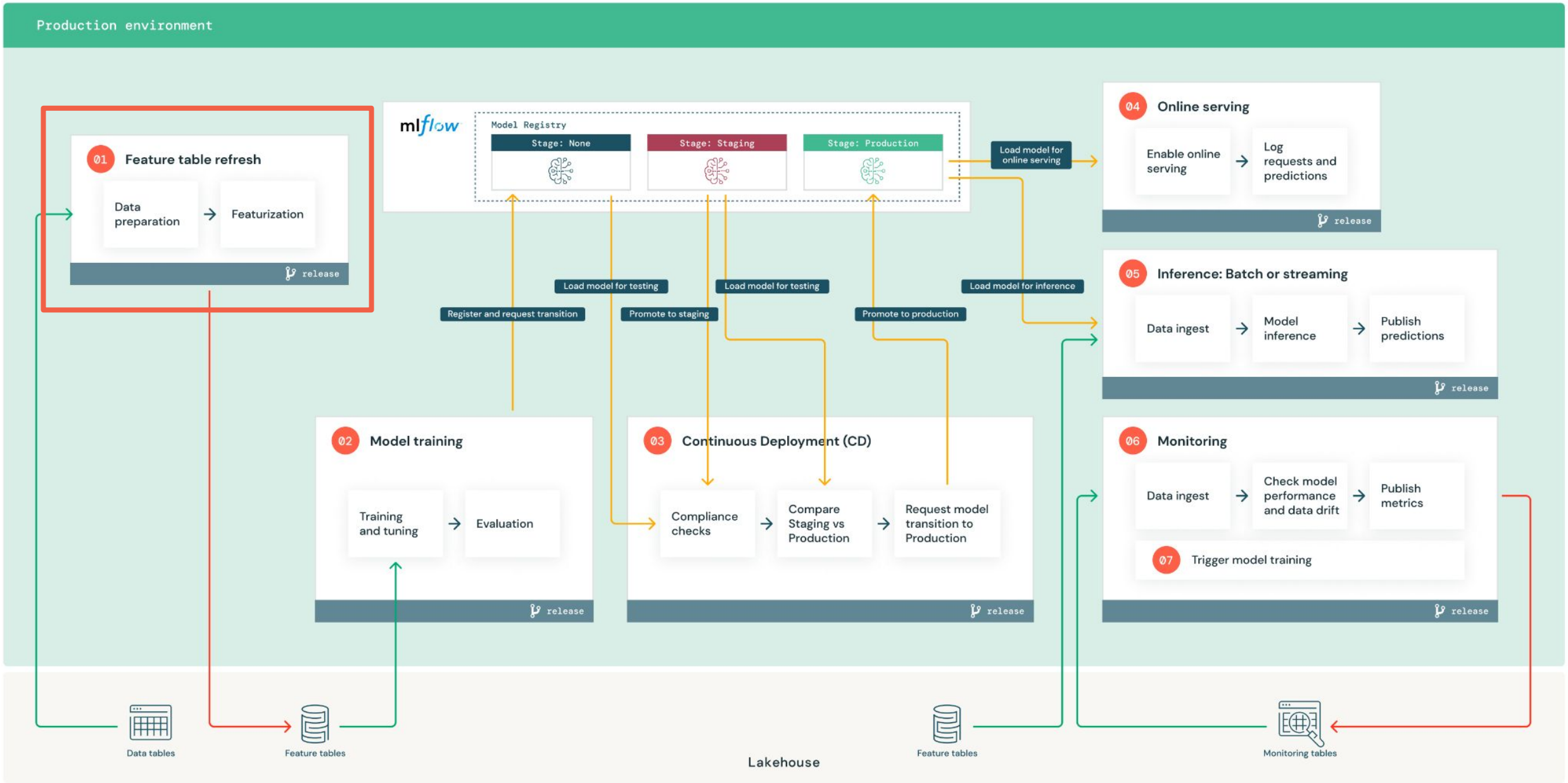
- MLflow Tracking
  - Track params, metrics and model artifacts
- MLflow Model Registry
  - Manage model lifecycle

Overview of Pipelines



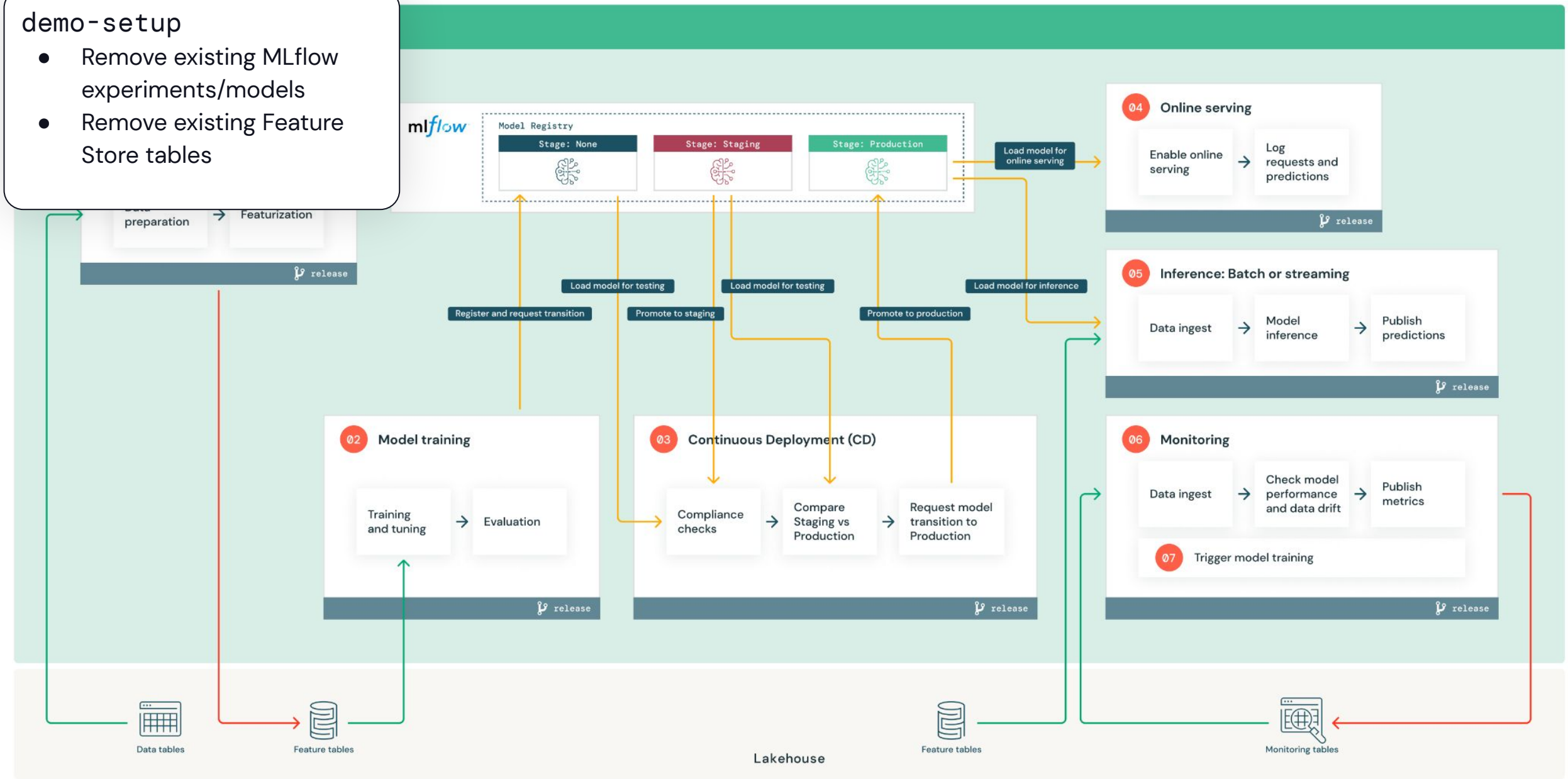
Overview of pipelines


Production environment
mlflow
Model Registry
Stage: None
Stage: Staging
Stage: Production
01 Feature table refresh
Data preparation
Featurization
release
04 Online serving
Enable online serving
Log requests and predictions
release
Load model for online serving
Load model for testing
Load model for testing
Load model for inference
Register and request transition
Promote to staging
Promote to production
05 Inference: Batch or streaming
Data ingest
Model inference
Publish predictions
release
02 Model training
Training and tuning
Evaluation
release
03 Continuous Deployment (CD)
Compliance checks
Compare Staging vs Production
Request model transition to Production
release
06 Monitoring
Data ingest
Check model performance and data drift
Publish metrics
07 Trigger model training
release
Data tables
Feature tables
Lakehouse
Feature tables
Monitoring tables
Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch


Production environment
01 Feature table refresh
Data preparation
Featurization
release
mlflow
Model Registry
Stage: None
Stage: Staging
Stage: Production
Load model for online serving
04 Online serving
Enable online serving
Log requests and predictions
release
Load model for testing
Load model for testing
Load model for inference
Register and request transition
Promote to staging
Promote to production
05 Inference: Batch or streaming
Data ingest
Model inference
Publish predictions
release
02 Model training
Training and tuning
Evaluation
release
03 Continuous Deployment (CD)
Compliance checks
Compare Staging vs Production
Request model transition to Production
release
06 Monitoring
Data ingest
Check model performance and data drift
Publish metrics
07 Trigger model training
release
Data tables
Feature tables
Lakehouse
Feature tables
Monitoring tables
Pipeline
Task
Reads
Writes
Model
Model transition
Repo
Branch

`demo-setup`

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

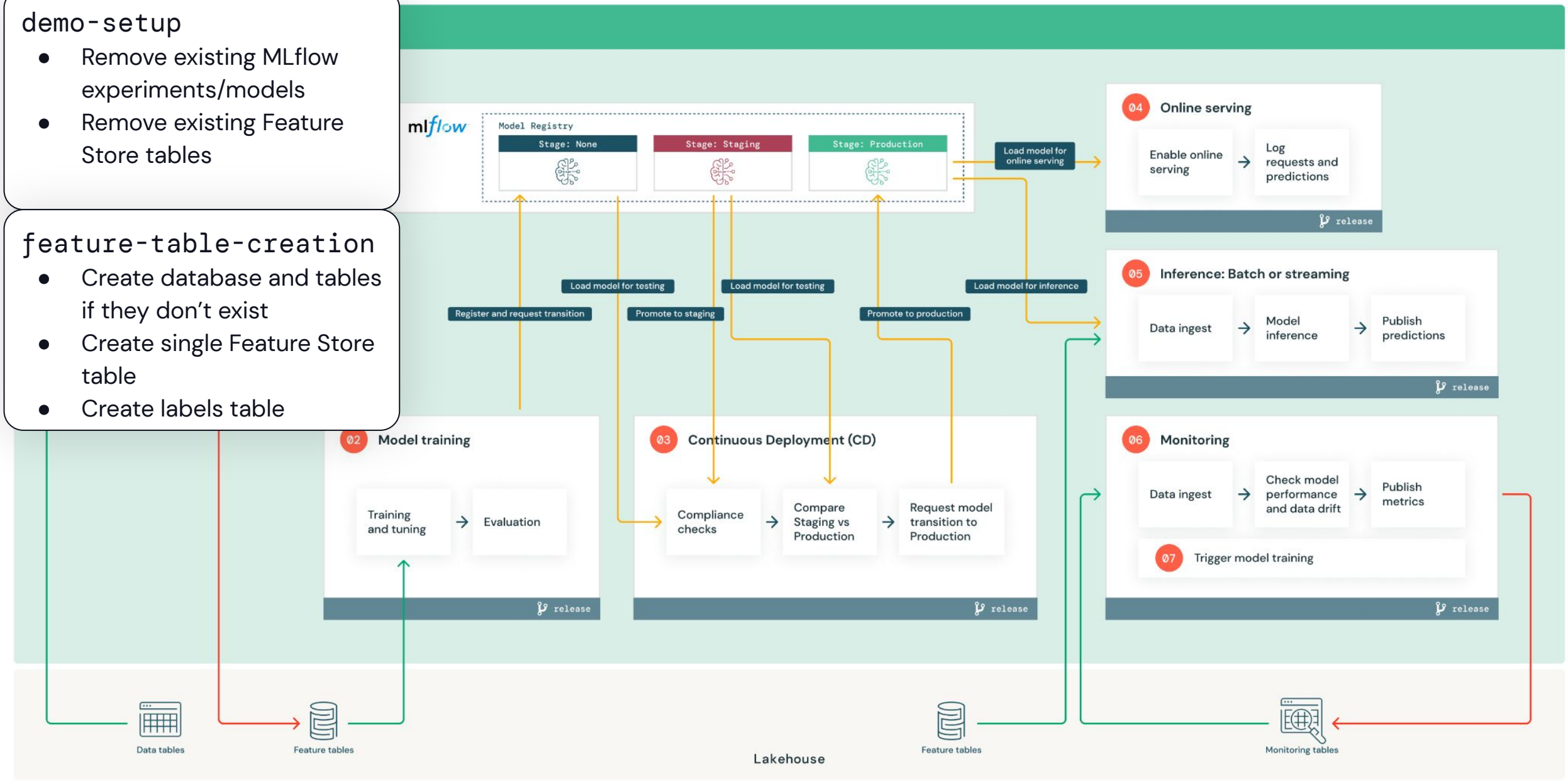


`demo-setup`

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

`feature-table-creation`

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table

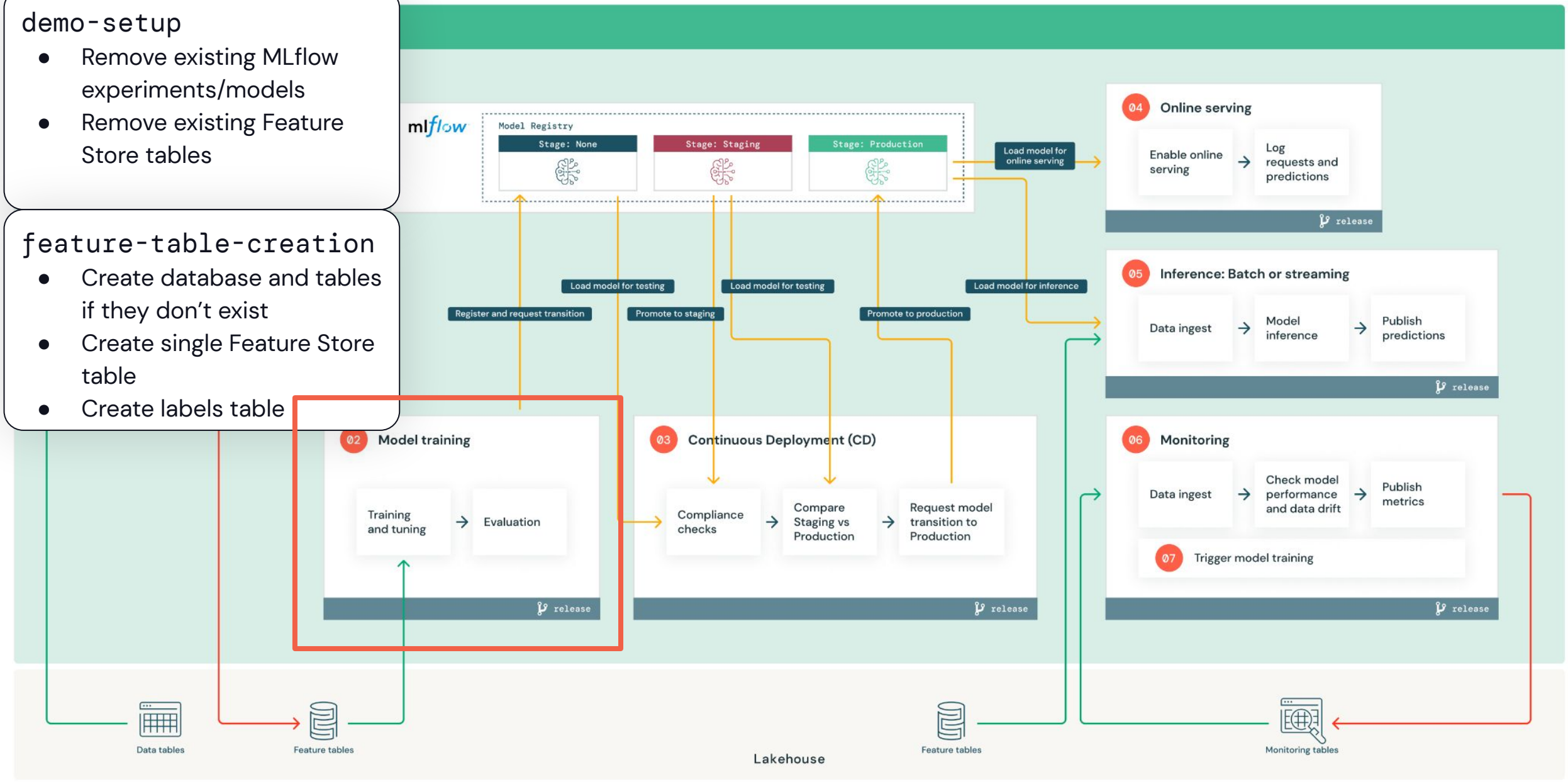


`demo-setup`

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

`feature-table-creation`

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table



Pipeline · Reads · Model · Repo
Task · Writes · Model transition · Branch

`demo-setup`

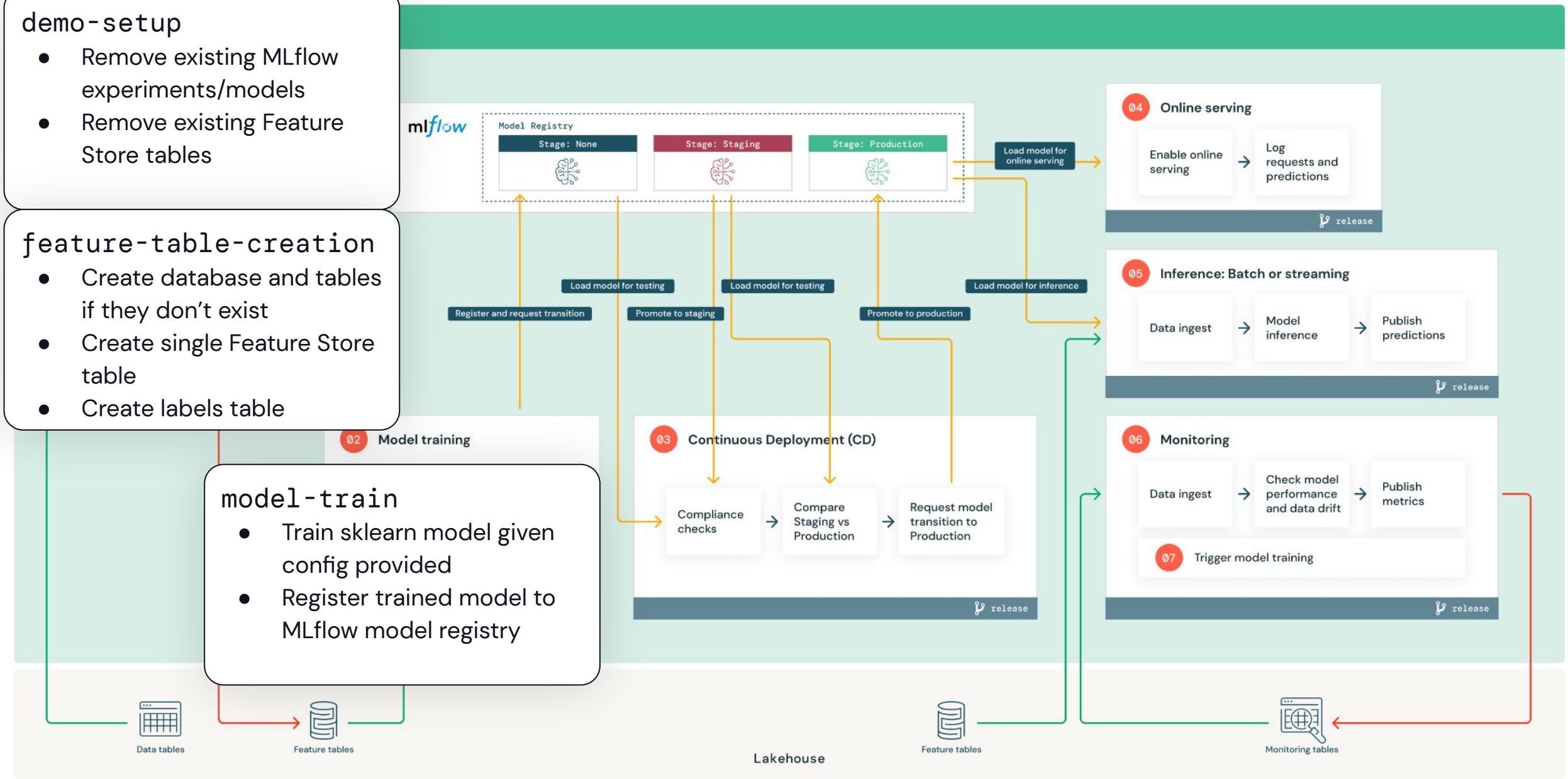
- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

`feature-table-creation`

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table

`model-train`

- Train sklearn model given config provided
- Register trained model to MLflow model registry



demo-setup

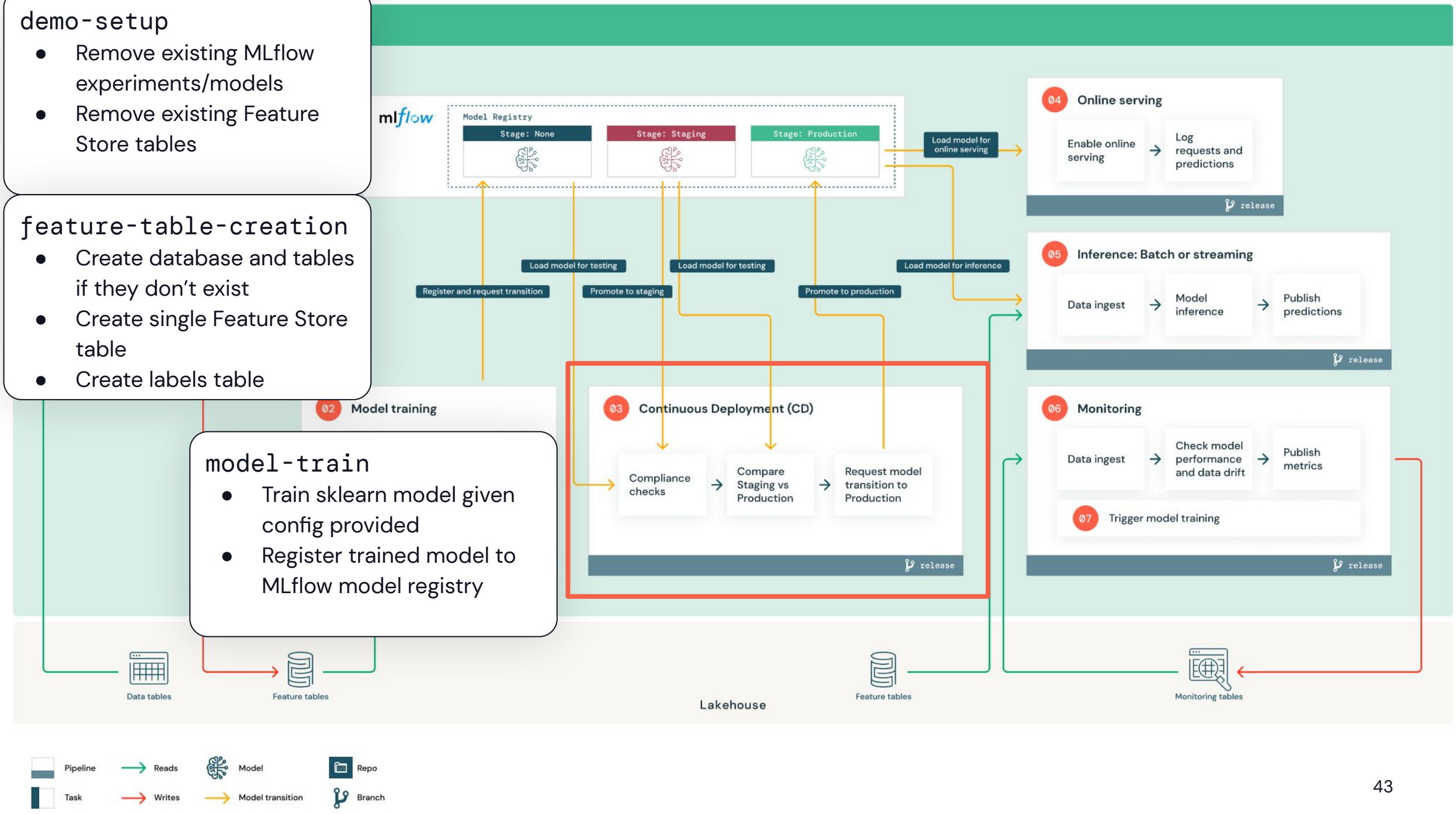
- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

feature-table-creation

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table

model-train

- Train sklearn model given config provided
- Register trained model to MLflow model registry

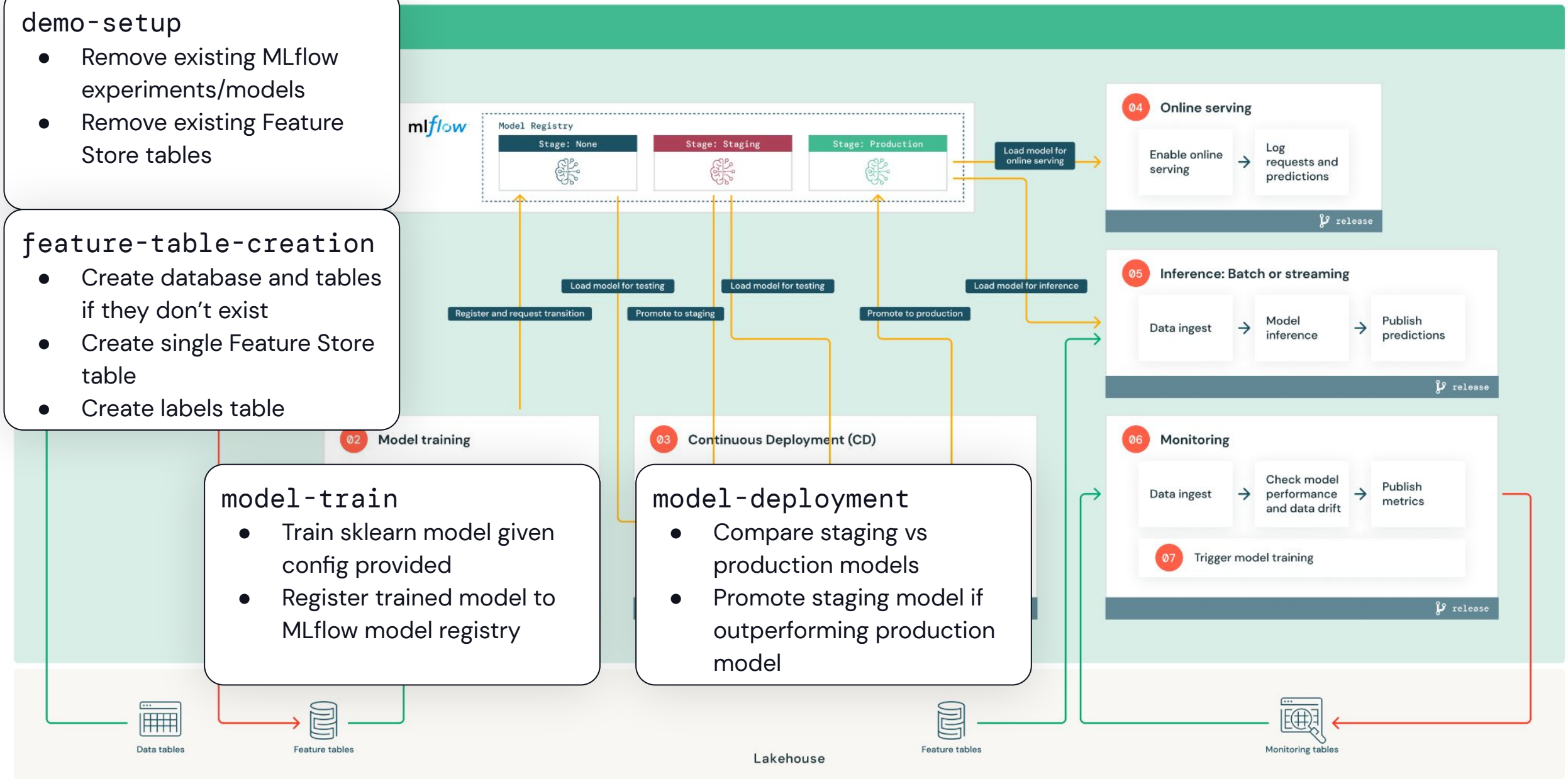


demo-setup

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

feature-table-creation

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table



model-train

- Train sklearn model given config provided
- Register trained model to MLflow model registry

model-deployment

- Compare staging vs production models
- Promote staging model if outperforming production model

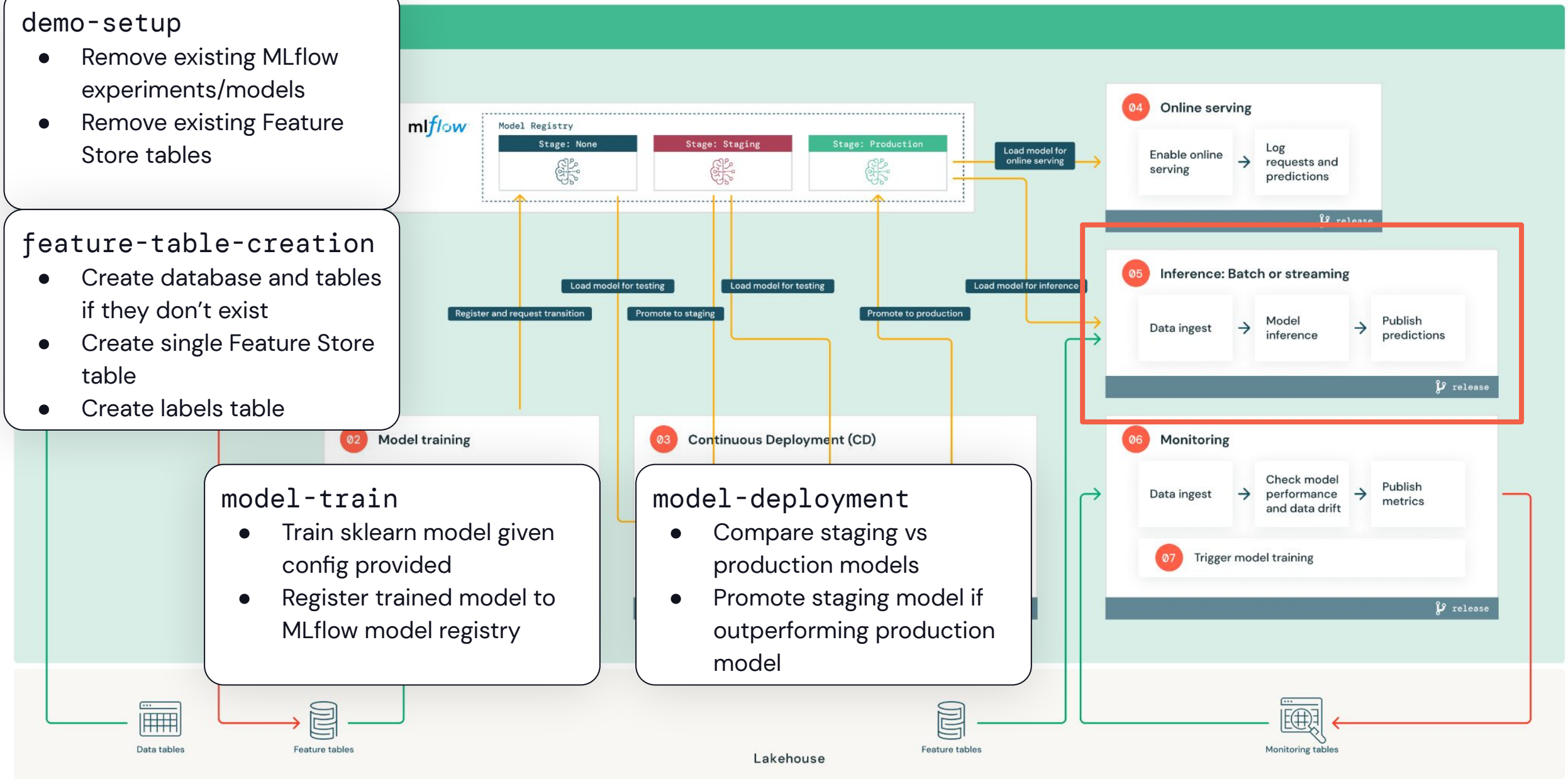
Pipeline · Reads · Model · Repo
Task · Writes · Model transition · Branch

demo-setup

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

feature-table-creation

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table



model-train

- Train sklearn model given config provided
- Register trained model to MLflow model registry

model-deployment

- Compare staging vs production models
- Promote staging model if outperforming production model

Pipeline · Reads · Model · Repo
Task · Writes · Model transition · Branch

demo-setup

- Remove existing MLflow experiments/models
- Remove existing Feature Store tables

feature-table-creation

- Create database and tables if they don't exist
- Create single Feature Store table
- Create labels table

model-train

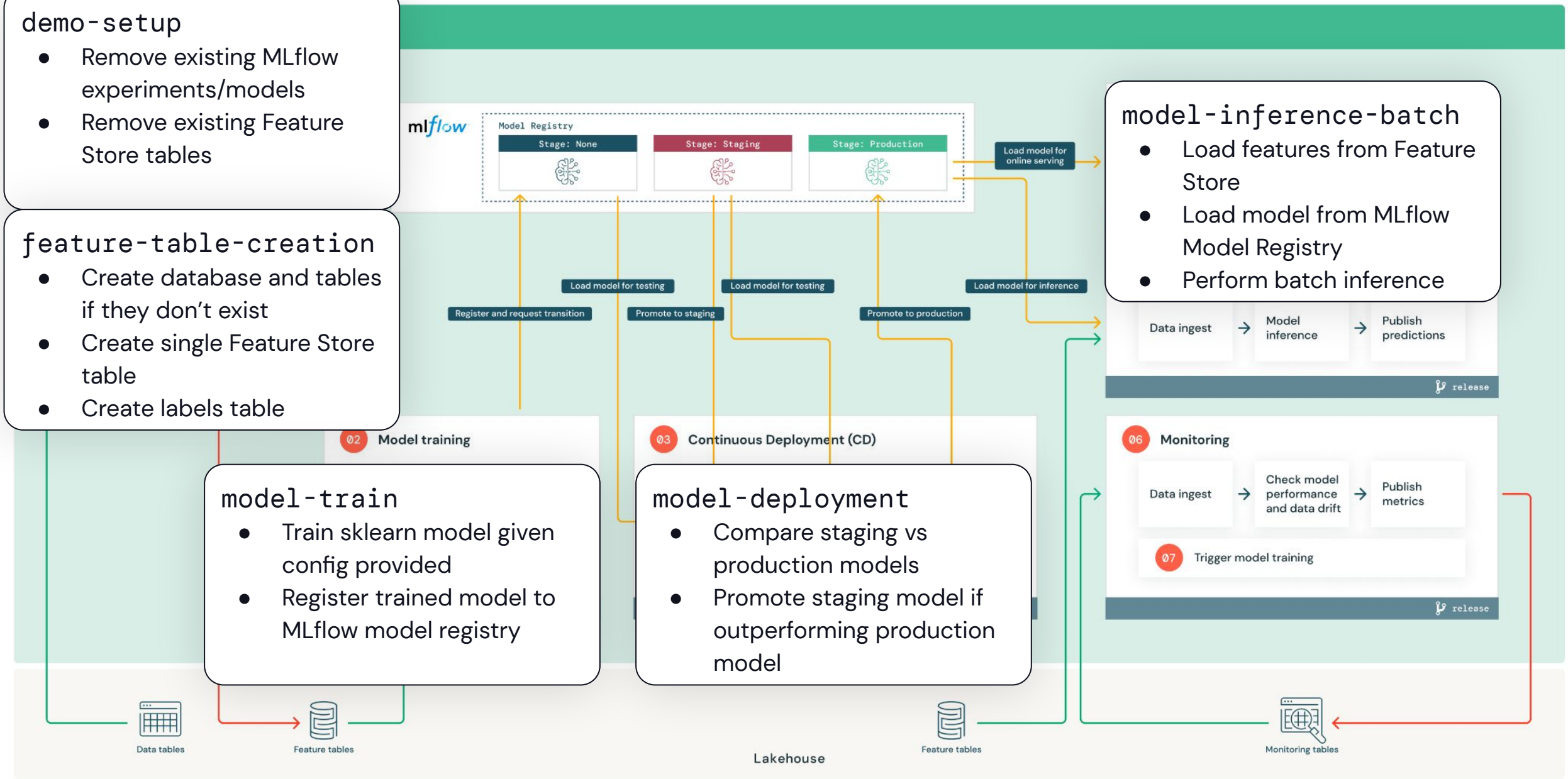
- Train sklearn model given config provided
- Register trained model to MLflow model registry

model-deployment

- Compare staging vs production models
- Promote staging model if outperforming production model

model-inference-batch

- Load features from Feature Store
- Load model from MLflow Model Registry
- Perform batch inference



Project Structure

Project Structure

```
├── README.md
├── .dbx                    <- Directory containing environment configuration.
│   ├── project.json        <- Used to configure the project artifact locations and Databricks CLI profiles.
```

Project Structure

```
├── README.md
├── .dbx                    <- Directory containing environment configuration.
│   ├── project.json        <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows           <- Directory containing GitHub Actions Workflows for CI testing.
```

Project Structure

```
├── README.md
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
```

Project Structure

```
├── README.md
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
│
├── conf                        <- Directory containing configuration files and environment variables files.
│   ├── job_configs             <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml          <- File to configure job deployment properties (cluster configs, job specs).
```

Project Structure

```
├── README.md
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
│
├── conf                        <- Directory containing configuration files and environment variables files.
│   ├── job_configs             <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml          <- File to configure job deployment properties (cluster configs, job specs).
│
├── telco_churn                 <- Python source code for the project pipelines.
```

Project Structure

```
├── README.md
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
│
├── conf                        <- Directory containing configuration files and environment variables files.
│   ├── job_configs             <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml          <- File to configure job deployment properties (cluster configs, job specs).
│
├── telco_churn                 <- Python source code for the project pipelines.
│
├── tests                       <- Unit and integration test code for the various pipelines.
│   ├── unit                    <- Directory containing unit tests.
│   └── integration             <- Directory containing integration tests.
```

Project Structure

Who owns what?

```
├── README.md
├── .dbx                          <- Directory containing environment configuration.
│   ├── project.json              <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows                 <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt              <- Requirements file for reproducing the package.
├── unit-requirements.txt         <- Requirements file for unit tests.
├── setup.py                      <- Used to build the package .whl
│
├── conf                          <- Directory containing configuration files and environment variables files.
│   ├── job_configs               <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml            <- File to configure job deployment properties (cluster configs, job specs).
│
├── telco_churn                   <- Python source code for the project pipelines.
│
├── tests                         <- Unit and integration test code for the various pipelines.
│   ├── unit                      <- Directory containing unit tests.
│   └── integration               <- Directory containing integration tests.
```

Project Structure

Who owns what?

ML Engineer

```
├── README.md
├── .dbx                          <- Directory containing environment configuration.
│   ├── project.json              <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows                 <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt              <- Requirements file for reproducing the package.
├── unit-requirements.txt         <- Requirements file for unit tests.
├── setup.py                      <- Used to build the package .whl
│
├── conf                          <- Directory containing configuration files and environment variables files.
│   ├── job_configs               <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml            <- File to configure job deployment properties (cluster configs, job specs).
│
├── telco_churn                   <- Python source code for the project pipelines.
│
├── tests                         <- Unit and integration test code for the various pipelines.
│   ├── unit                      <- Directory containing unit tests.
│   └── integration               <- Directory containing integration tests.
```

Project Structure

Who owns what?

```
├── README.md                                                    ML Engineer
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
│
├── conf                        <- Directory containing configuration files and environment variables files.
│   ├── job_configs             <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml          <- File to configure job deployment properties (cluster configs, job specs).
│
├── telco_churn                 <- Python source code for the project pipelines.
│
├── tests                         <- Unit and integration test code for the various pipelines.
│   ├── unit                    <- Directory containing unit tests.
│   └── integration             <- Directory containing integration tests.                Data Scientist
```

Project Structure

```
├── README.md
├── .dbx                        <- Directory containing environment configuration.
│   ├── project.json            <- Used to configure the project artifact locations and Databricks CLI profiles.
├── .github
│   ├── workflows               <- Directory containing GitHub Actions Workflows for CI testing.
├── requirements.txt            <- Requirements file for reproducing the package.
├── unit-requirements.txt       <- Requirements file for unit tests.
├── setup.py                    <- Used to build the package .whl
│
├── conf                        <- Directory containing configuration files and environment variables files.
│   ├── job_configs             <- Directory containing .yml configuration files for pipeline definitions.
│   ├── deployment.yml          <- File to configure job deployment properties (cluster configs, job specs).
│                                                                                    Data Scientist
├── telco_churn                 <- Python source code for the project pipelines.
│
├── tests                       <- Unit and integration test code for the various pipelines.
│   ├── unit                    <- Directory containing unit tests.
│   └── integration             <- Directory containing integration tests.
```

Project Structure

```
├── telco_churn
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py
```

<- Python file containing a generic class called Job to run jobs via dbx.

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                  <- Python file containing a generic class called Job to run jobs via dbx.
│
│   ├── featurize.py               <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py   <- Create Databricks Feature Store table.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│   │                                                                    feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│                                                                                    feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│
│                                                                                    feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│
│                                                                                    model-train
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│                                                                                  feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│                                                                                  model-train
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
│
│   ├── model_deployment.py          <- Model comparison prior to deployment.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│                                                                                    feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│                                                                                    model-train
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
│                                                                                    model-deployment
│   ├── model_deployment.py          <- Model comparison prior to deployment.
```

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│   ├── model_train_pipeline.py      <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
│   ├── model_deployment.py          <- Model comparison prior to deployment.
│   ├── model_inference.py           <- Load model and perform inference.
```

feature-table-creation

model-train

model-deployment

Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│   ├── model_train_pipeline.py      <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow. Tracking.
│   ├── model_deployment.py          <- Model comparison prior to deployment.
│   ├── model_inference.py           <- Load model and perform inference.
```

feature-table-creation

model-train

model-deployment

model-inference-batch

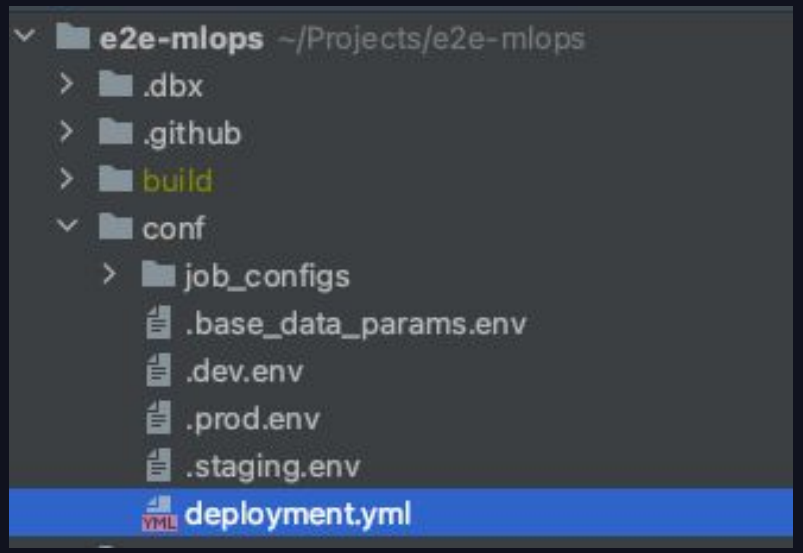
Project Structure

```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│   │                                                                                  feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│   │                                                                                  model-train
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
│   │                                                                                  model-deployment
│   ├── model_deployment.py          <- Model comparison prior to deployment.
│   │                                                                                  model-inference-batch
│   ├── model_inference.py             <- Load model and perform inference.
│   │
│   ├── jobs                         <- Package containing job entrypoint modules.
```

Project Structure

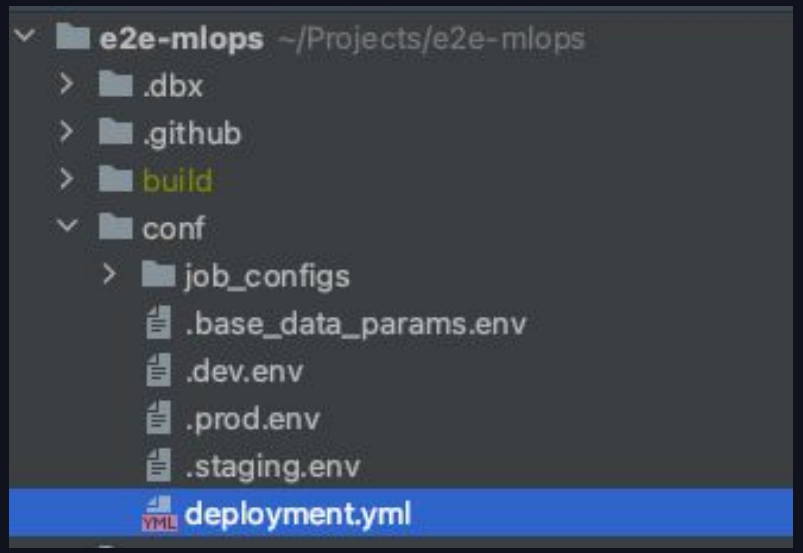
```
├── telco_churn
│   ├── __init__.py
│   ├── common.py                    <- Python file containing a generic class called Job to run jobs via dbx.
│   │                                                                                  feature-table-creation
│   ├── featurize.py                 <- Perform data preprocessing steps on features and label.
│   ├── feature_table_creator.py     <- Create Databricks Feature Store table.
│   │                                                                                  model-train
│   ├── model_train_pipeline.py        <- Create sklearn pipeline.
│   ├── model_train.py               <- Train model and track params, metrics and model to MLflow Tracking.
│   │                                                                                  model-deployment
│   ├── model_deployment.py          <- Model comparison prior to deployment.
│   │                                                                                  model-inference-batch
│   ├── model_inference.py             <- Load model and perform inference.
│   │
│   ├── jobs                         <- Package containing job entrypoint modules.
│   └── utils                          <- Package containing various utility modules.
```

Mechanics of pipeline



deployment.yml

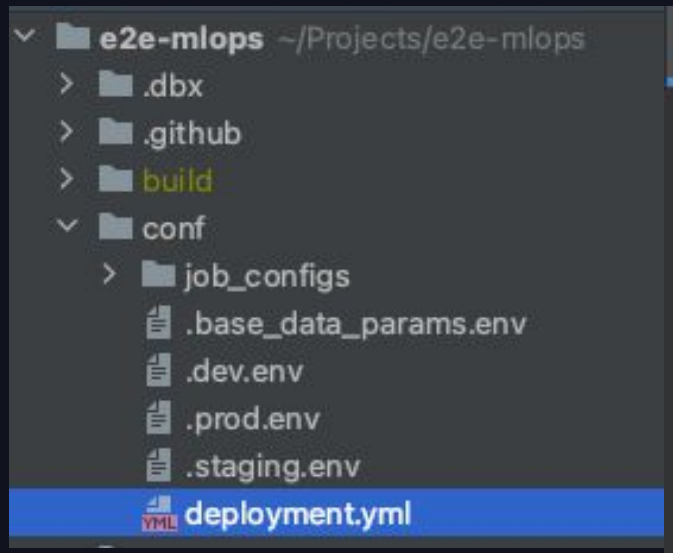
Mechanics of pipeline



`deployment.yml`

1. Cluster specifications
2. Job definitions

Mechanics of pipeline



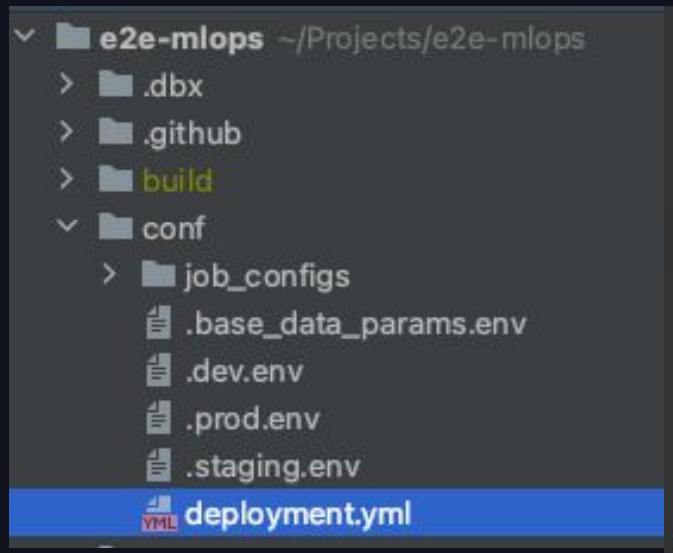
`deployment.yml`

1. Cluster specifications
2. Job definitions

deployment.yml

```yaml
default-cluster-spec: &default-cluster-spec
  spark_version: '11.0.x-cpu-ml-scala2.12'
#   node_type_id: 'i3.xlarge' # node_type_id and driver_node_ty
#   driver_node_type_id: 'i3.xlarge'
  num_workers: 1
  driver_instance_pool_id: '0617-151415-bells2-pool-hh7h6tjm'
  instance_pool_id: '0617-151415-bells2-pool-hh7h6tjm'

dev-cluster-config: <1 key>

staging-c…ter-config: <1 key>

prod-cluster-config: &prod-cluster-config
  new_cluster:
    <<: *default-cluster-spec
```

Mechanics of pipeline

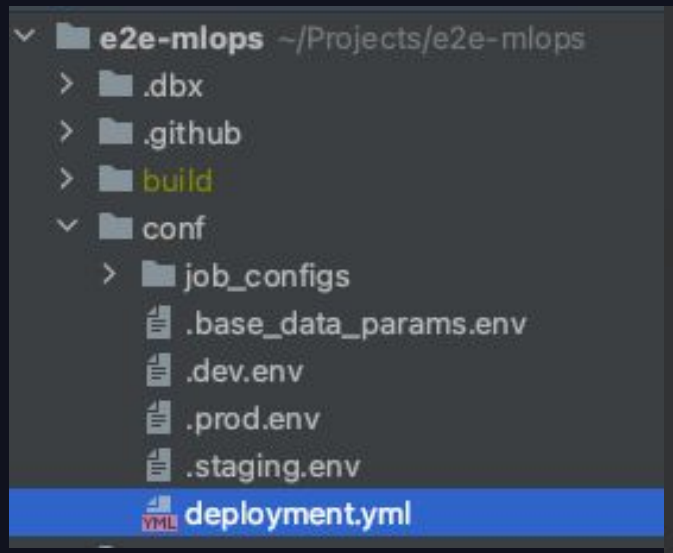


`deployment.yml`

1. Cluster specifications
2. Job definitions

```yaml
environments:

  dev: <2 keys>

  staging: <2 keys>

  prod:
    strict_path_adjustment_policy: true
    jobs:
      - <3 keys>
      - <2 keys>
      - name: 'model-train'
        <<:
          - *prod-cluster-config
        spark_python_task:
          python_file: 'file://telco_churn/jobs/model_train_job.py'
          parameters: ['--base-data-params', 'file:fuse://conf/.base_data_params.env',
                       '--env', 'file:fuse://conf/.prod.env',
                       '--conf-file', 'file:fuse://conf/job_configs/model_train.yml']
```

Mechanics of pipeline



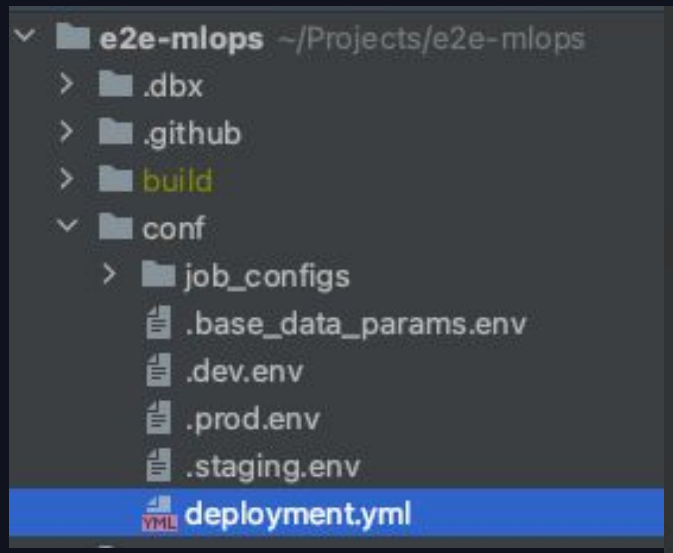
`deployment.yml`

1. Cluster specifications
2. Job definitions
   a. Cluster config

```
27  environments:
28
29    dev: <2 keys>
78
79    staging: <2 keys>
89
90    prod:
91      strict_path_adjustment_policy: true
92      jobs:
93        - <3 keys>
100       - <2 keys>
129       - name: 'model-train'
130         <<:
131           - *prod-cluster-config
132         spark_python_task:
133           python_file: 'file://telco_churn/jobs/model_train_job.py'
134           parameters: ['--base-data-params', 'file:fuse://conf/.base_data_params.env',
135                        '--env', 'file:fuse://conf/.prod.env',
136                        '--conf-file', 'file:fuse://conf/job_configs/model_train.yml']
```

Cluster config

Mechanics of pipeline



`deployment.yml`

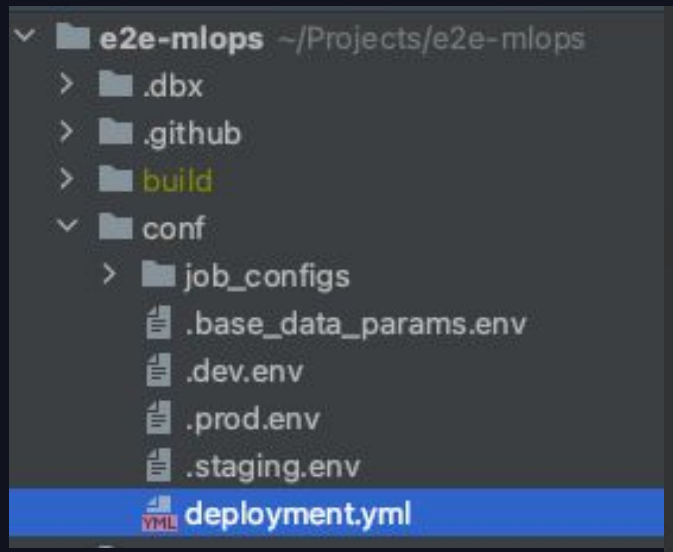
1. Cluster specifications
2. Job definitions
   a. Cluster config
   b. Job entry file

```
environments:

  dev: <2 keys>

  staging: <2 keys>

  prod:
    strict_path_adjustment_policy: true
    jobs:
      - <3 keys>
      - <2 keys>
      - name: 'model-train'
        <<:
          - *prod-cluster-config
        spark_python_task:
          python_file: 'file://telco_churn/jobs/model_train_job.py'
          parameters: ['--base-data-params', 'file:fuse://conf/.base_data_params.env',
                       '--env', 'file:fuse://conf/.prod.env',
                       '--conf-file', 'file:fuse://conf/job_configs/model_train.yml']
```

Cluster config

Job entry file

Mechanics of pipeline



`deployment.yml`

1. Cluster specifications
2. Job definitions
   a. Cluster config
   b. Job entry file
   c. Environment vars

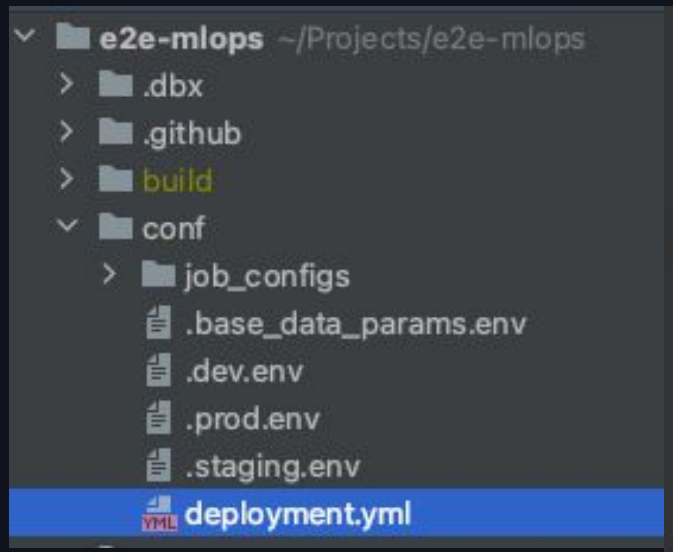
```
environments:

  dev: <2 keys>

  staging: <2 keys>

  prod:
    strict_path_adjustment_policy: true
    jobs:
      - <3 keys>
      - <2 keys>
      - name: 'model-train'
        <<:
          - *prod-cluster-config
        spark_python_task:
          python_file: 'file://telco_churn/jobs/model_train_job.py'
          parameters: ['--base-data-params', 'file:fuse://conf/.base_data_params.env',
                       '--env', 'file:fuse://conf/.prod.env',
                       '--conf-file', 'file:fuse://conf/job_configs/model_train.yml']
```

Cluster config

Job entry file

Environment specific variables

Mechanics of pipeline



`deployment.yml`

1. Cluster specifications
2. Job definitions
   a. Cluster config
   b. Job entry file
   c. Environment vars
   d. Config file

```
environments:

  dev: <2 keys>

  staging: <2 keys>

  prod:
    strict_path_adjustment_policy: true
    jobs:
      - <3 keys>
      - <2 keys>
      - name: 'model-train'
        <<:
          - *prod-cluster-config
        spark_python_task:
          python_file: 'file://telco_churn/jobs/model_train_job.py'
          parameters: ['--base-data-params', 'file:fuse://conf/.base_data_params.env',
                       '--env', 'file:fuse://conf/.prod.env',
                       '--conf-file', 'file:fuse://conf/job_configs/model_train.yml']
```

Cluster config

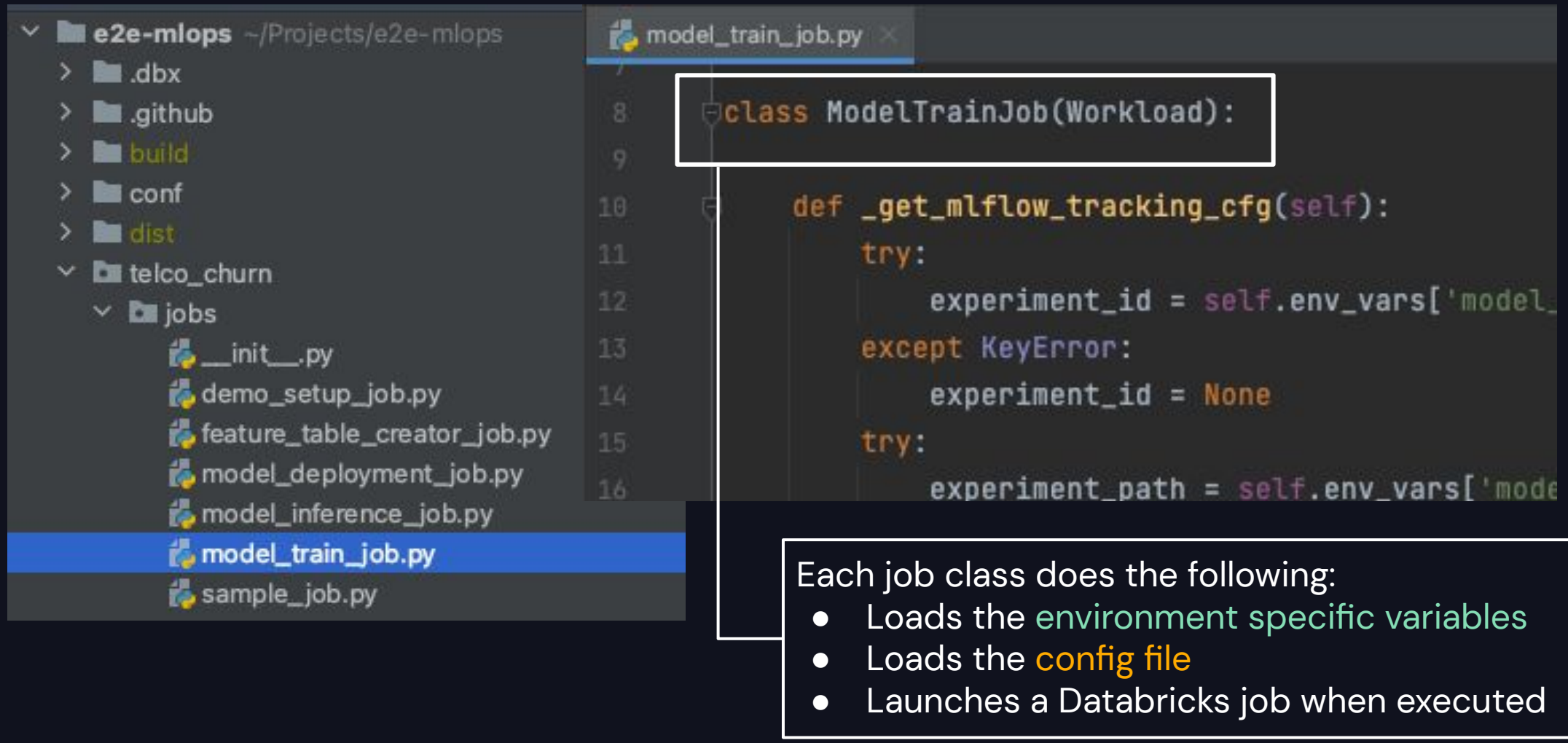
Job entry file

Environment specific variables

Config file

Example: model-train

Job entry file: `telco_churn/jobs/model_train_job.py`



Each job class does the following:

- Loads the environment specific variables
- Loads the config file
- Launches a Databricks job when executed

Demo Workflow

Demo Workflow

1. Start at steady state
    - Model in production
    - Feature tables
    - Existing pipelines in prod environment
    - Monitoring dashboard

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

3. Create pull request
   - Unit tests
   - Integration tests
   - If tests pass, change is merged into main (staging) branch

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

3. Create pull request
   - Unit tests
   - Integration tests
   - If tests pass, change is merged into main (staging) branch

ML Engineer

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

3. Create pull request
   - Unit tests
   - Integration tests
   - If tests pass, change is merged into main (staging) branch

4. Cut release branch — ML Engineer
   - Tag a new release version
   - On pushing tag, release pipeline triggered

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

3. Create pull request
   - Unit tests
   - Integration tests
   - If tests pass, change is merged into main (staging) branch

4. Cut release branch — ML Engineer
   - Tag a new release version
   - On pushing tag, release pipeline triggered

5. Release pipeline deploys workflows to prod
   - model-train
   - model-deployment
   - model-inference-batch

Demo Workflow

1. Start at steady state
   - Model in production
   - Feature tables
   - Existing pipelines in prod environment
   - Monitoring dashboard

2. Model update — Data Scientist
   - Model performance has degraded
   - Update model train config file
   - Push code changes

3. Create pull request
   - Unit tests
   - Integration tests
   - If tests pass, change is merged into main (staging) branch

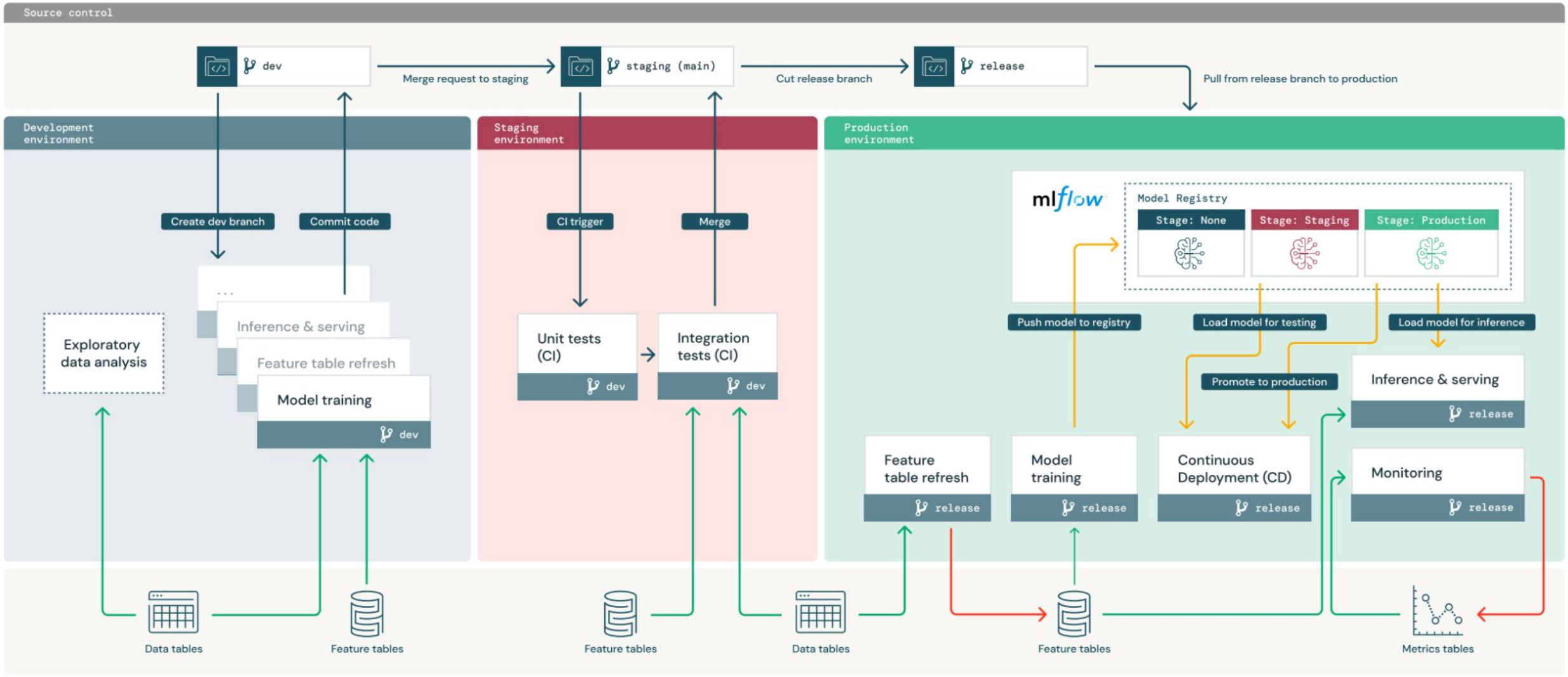
4. Cut release branch — ML Engineer
   - Tag a new release version
   - On pushing tag, release pipeline triggered

5. Release pipeline deploys workflows to prod
   - model-train
   - model-deployment
   - model-inference-batch

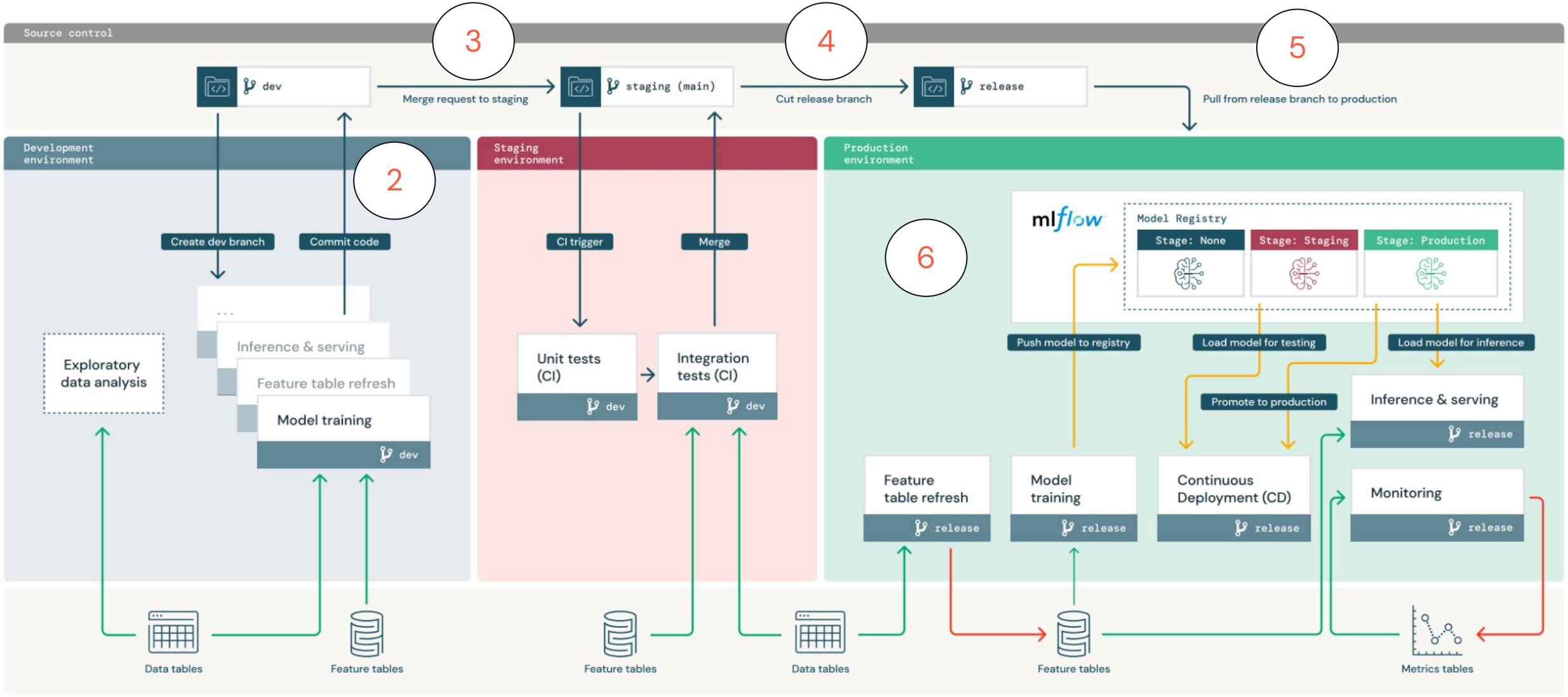
6. Manually trigger or automate workflows

Demo

Demo workflow



Demo workflow



Model Management

End to End Model Management

Setup MLflow Model webhook
Slack notifications, Trigger Testing Jobs

Schedule Monthly Retrain Job
Databricks Job

Data Prep & Featurization
ETL + EDA, Feature Engineering

Build baseline Model with AutoML
MLflow autologging, Hyperopt +Spark

Promote Best Run to Registry
Annotate model. Request transition to staging

Automated Model testing
Schema, Demographic accuracy, Docs & artifacts...
Approve/reject request

Run inferences
Load model
Batch or real-time

Feature Store

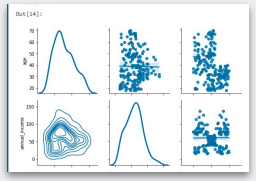
Tracking

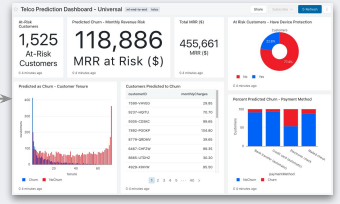
STAGING Request
Model Registry
Webhook triggers test

Approved. Move to STAGING

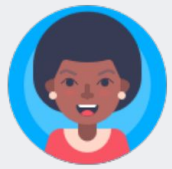
STAGING

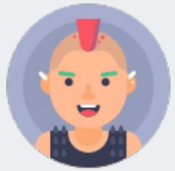
Rejected





{ api }
Realtime HTTP inference


Data Scientist


ML Engineer


Data Engineer

Data Transformation

Within the model or prior to model training? It depends!

	Pros	Cons
Prior to Model Training	- Compute once and reuse - Leverages full dataset	- Increases production footprint - Slower to iterate
Within the Model	- Easier to iterate - Smaller production footprint	- Model latency depends upon transformation overheads - Data visibility

mlflow Model Registry

VISION: Centralized and collaborative model lifecycle management

mlflow

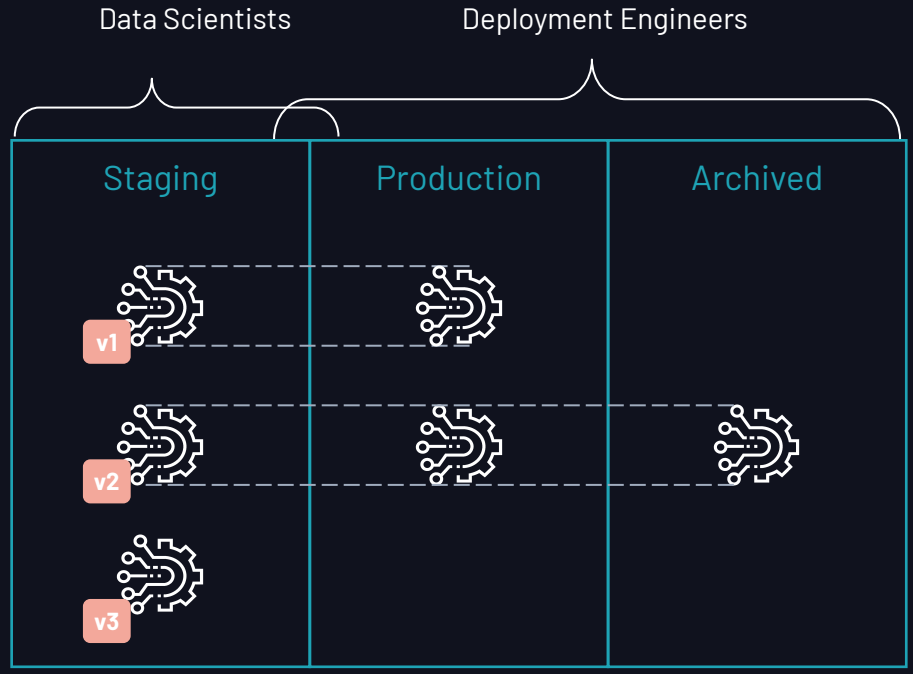
Tracking Server

Parameters Metrics Artifacts

Metadata Models

mlflow

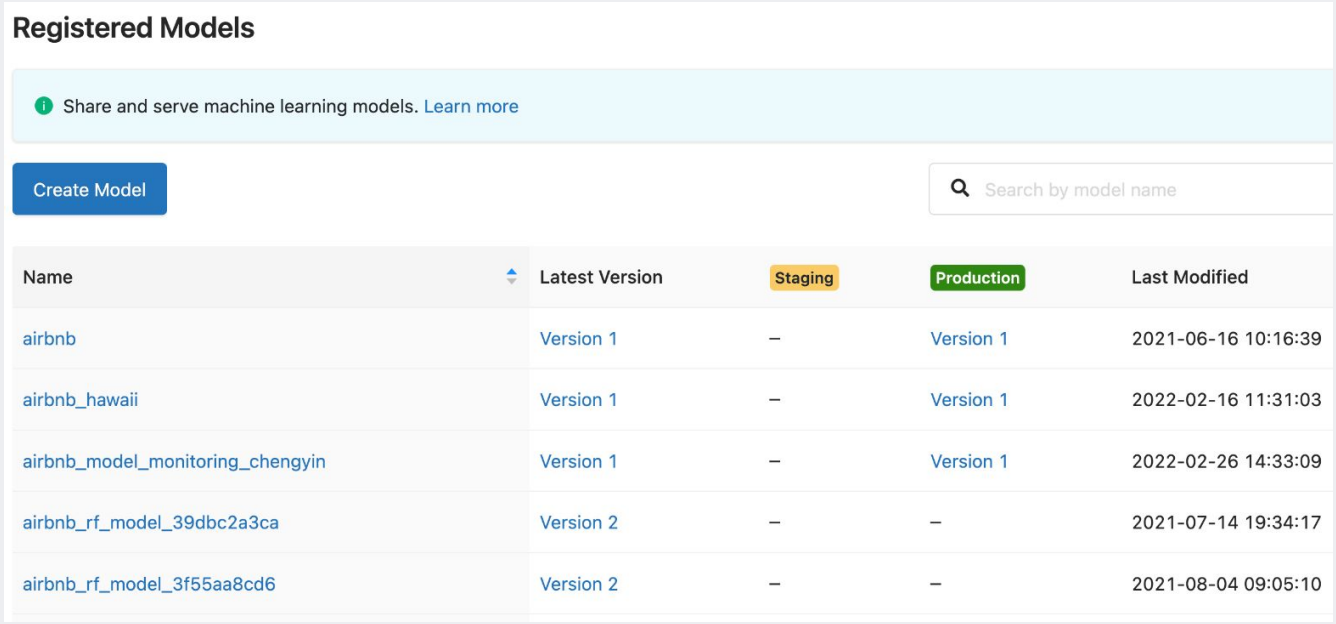
Model Registry



databricks

Model Management

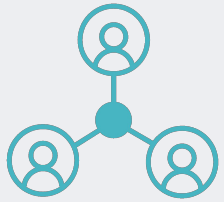
Use one central place to collaboratively manage ML models, from experimentation to online testing and production.



Provides:

- Automatic models versioning
- Integration with governance workflows
- Automatic/manual approval of stage transition
- Full visibility into model lifecycle from experimentation to production

Model Registry Benefits



One Collaborative Hub: The Model Registry provides a central hub for making models discoverable, improving **collaboration** and **knowledge sharing** across the organization.

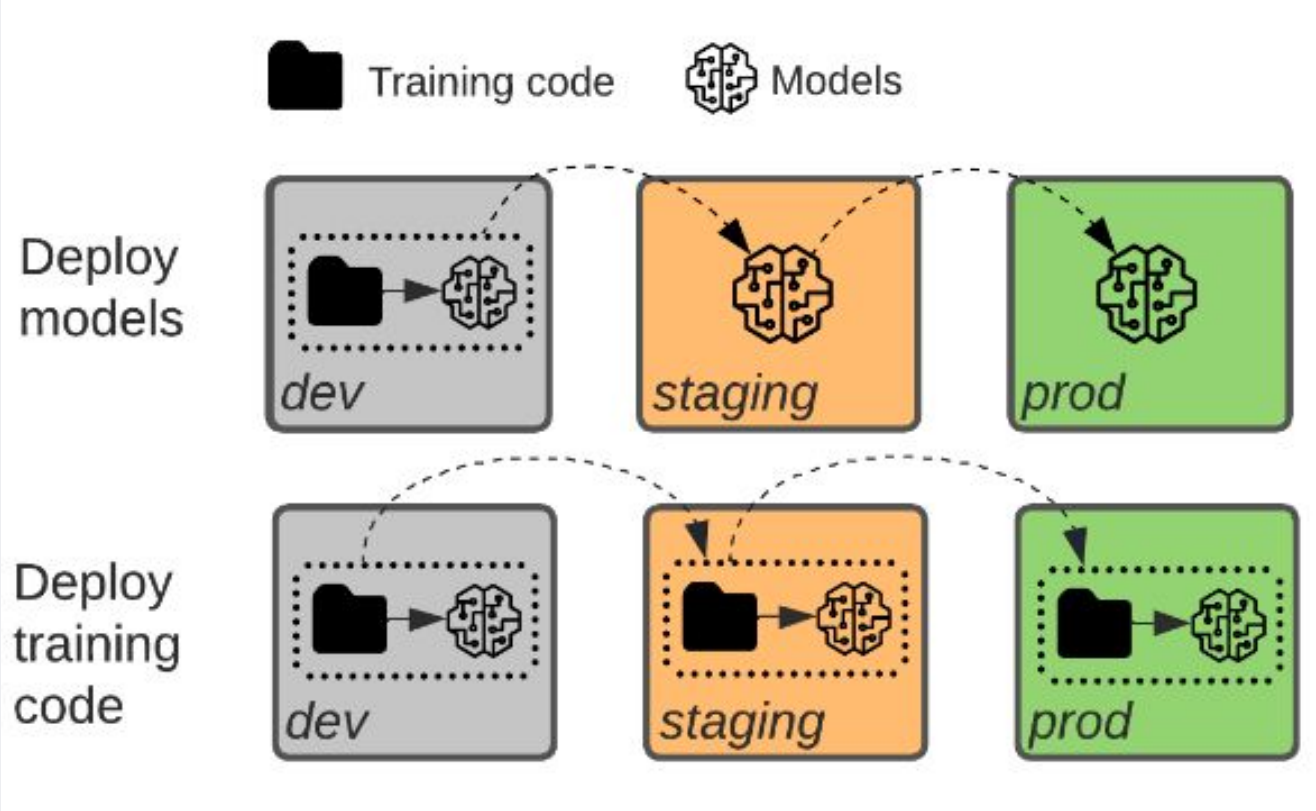
Manage the entire Model Lifecycle (MLOps): The Model Registry provides lifecycle management for models from experimentation to deployment, improving **reliability** and robustness of the model deployment process.

Visibility and Governance: The Model Registry provides full visibility into the deployment stage of all models, who requested and approved changes, allowing for full governance and auditability.

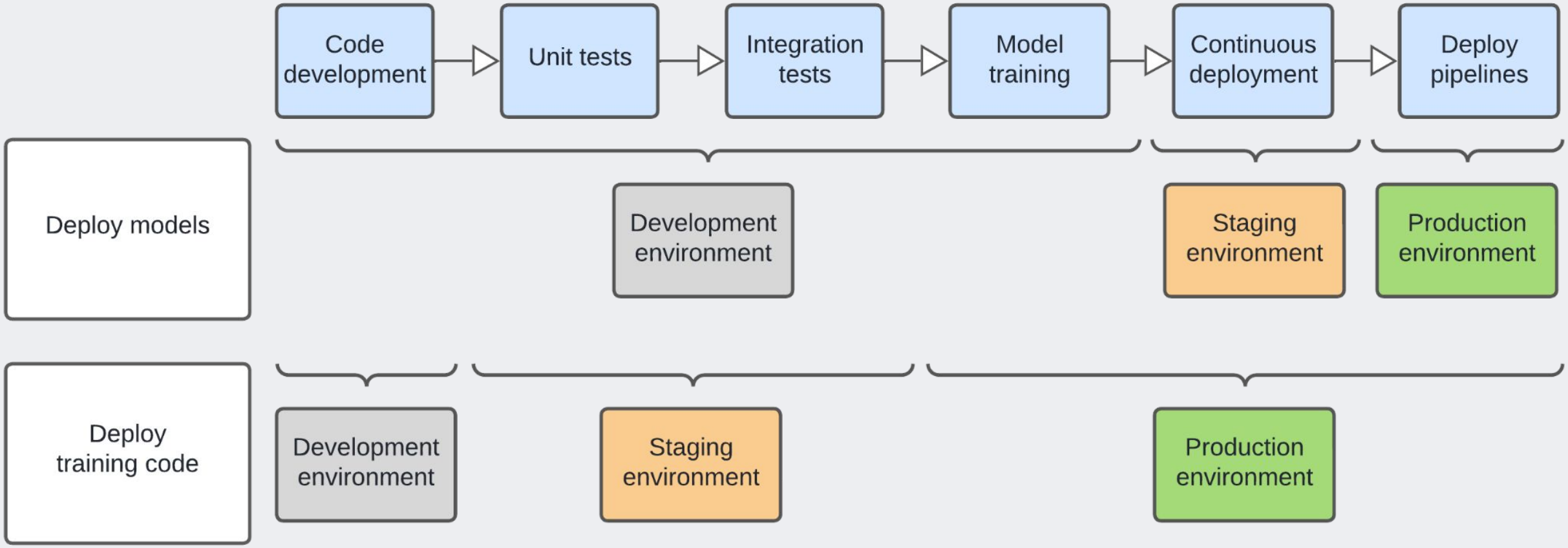
Model Deployment Patterns

Deploy models:
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.

Deploy training code:
the code to train models is developed in the dev environment, and this code is moved to staging and then production.



Contrast for Code Life Cycle



Comparisons (Process)

		Deploy models	Deploy training code
Process	**Dev**	Develop training code. Develop ancillary code.[1] Train model on prod data. → Promote model & ancillary code.	Develop training code. Develop ancillary code. → Promote code.
	Staging	Test model & ancillary code. → Promote model & 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.

		Deploy models	Deploy training code
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 & 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 & 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

Getting Started

Resources

Full whitepaper: “The Big Book of MLOps”

bit.ly/big-book-of-mlops

End-to-end MLOps demo repository

bit.ly/e2e-mlops-databricks

Catch up on other DAIS talks on MLOps.
Databricks product-related talks include:

- Enable Production ML with Databricks Feature Store (Wed @ 10:45am)
- MLflow Pipelines: Accelerating MLOps from Development to Production (Wed @ 11am)
- The Databricks Notebook: Front door of the Lakehouse (Wed @ 5:30pm)

Questions?

DATA+AI
SUMMIT 2022

Thank you