A diagram of a system

AI-generated content may be incorrect.

ML Pipeline Architecture with LightGBM

This architecture implements a complete **MLOps pipeline** for training, deploying, and maintaining a LightGBM model in production. It covers data handling, model training with MLflow tracking, model evaluation and promotion, containerization with BentoML & Docker, deployment on Kubernetes, monitoring with Prometheus/Grafana, and automation of maintenance tasks with Airflow. The design ensures reproducibility and continuous improvement through automated retraining triggers.

**1. Data Splitting (Fixed Test Set)**

At the start, **historical data** is split into training and a fixed test set. The test set is held out and reused for all model evaluations to ensure a consistent baseline. This prevents data leakage and allows fair comparison between the current production model and any new candidate model. Key considerations are to shuffle and stratify during the split, and to **persist the test set** so that all pipeline runs use the exact same test data slice. This consistent test set strategy helps detect genuine performance improvements and prevents overfitting to new data in evaluation.

**2. Preprocessing (Data Cleaning & Feature Engineering)**

Data preprocessing is encapsulated so it can be applied **uniformly to both training data and incoming production data**. This stage includes data cleaning and feature engineering. The same transformations must be applied at inference time to incoming feature data – to achieve this, the pipeline code is shared between training and serving environments.

**3. Training (LightGBM Model & MLflow Tracking)**

In the training stage, the pipeline trains a LightGBM model on the pre-processed training data. During training, MLflow Tracking is used to log key information: parameters, training metrics, and the trained model artifact​. After training, the new model is registered as a “candidate” model in the MLflow Model Registry. The registry will assign a version to this model and label it as a candidate model, separate from the current production model. This stage also logs evaluation metrics on the validation set (but not on the fixed test set, which is reserved for the next stage). By tracking experiments with MLflow, the pipeline enables comparing different runs and rolling back if needed. *Key considerations:* **resource management** (CPU/GPU, memory) is important for this stage – it could be executed on a dedicated training server (AWS SageMaker/E2).

**4. Evaluation (Champion/Challenger Testing)**

Once a candidate model is trained and registered, the **evaluation stage** runs it against the fixed test dataset and compares its performance to the current production model’s performance on that same test set. The pipeline loads the current prod model and scores the test data with both the prod model and the new candidate. It then computes relevant metrics (e.g., accuracy, AUC) for both. If the **candidate model outperforms the current model** on the test set and **meets predefined metric thresholds**, it is approved for promotion. If the candidate fails to beat the incumbent, the DVC pipeline can abort. All evaluation results are logged (stored back to MLflow) for audit. If the candidate is successful, the pipeline can transition the model in the registry to the “Production”. The next steps then prepare the model for serving.

**5. Packaging & Deployment (BentoML, Docker, Kubernetes)**

In this stage, the approved model is packaged into a **self-contained prediction service**. The pipeline uses **BentoML** to create a Bento bundle containing the model and its serving. BentoML streamlines containerizing models by capturing the environment dependencies and providing an inference server. Once the Bento bundle is created, it’s containerized as a Docker image. The Docker image is then deployed to a **Kubernetes** cluster for serving. BentoML’s model serving framework makes it straightforward to create a **portable REST API** for the model, and Kubernetes provides the infrastructure to serve it at scale.

**6. Monitoring (Prometheus & Grafana in Production)**

Once the model is live, the pipeline includes monitoring components to track its behaviour and performance. Prometheus is set up to scrape metrics from the model service. Grafana is used to visualize these metrics on dashboards. Alerts can be configured, if the model’s prediction rate drops or latency spikes, on-call engineers are notified. The pipeline can log summary statistics of incoming features to Prometheus. If data starts drifting significantly from training data (or if values fall outside expected ranges), an alert can be raised. This could indicate the model is facing unseen patterns and might need retraining soon.

**7. Automation with Airflow (Batch Jobs for Predictions & Metrics)**

An Airflow instance (running in Docker) automates various pipeline tasks on a schedule. Two important DAGs (Directed Acyclic Graphs) are set up:

* **Daily DAG:** Each day, Airflow triggers a batch job that stored recent predictions (SQLite DB) and **uploads the latest predictions to AWS S3**. In this design, after the model serves predictions, those prediction records are collected in a local SQLite database. The daily Airflow job extracts these records and backs them up to an S3 bucket. This provides durable storage of predictions and inputs, which is useful for later performance analysis.
* **Weekly DAG:** On a weekly schedule, Airflow triggers a pipeline to **evaluate model performance on recent ground truth data**. This job will download the latest **ground truth labels** from S3 (or wherever they are stored). The Airflow task joins the ground truth with the corresponding predictions (by LoanID) and computes **evaluation metrics on real production data**. This provides an up-to-date view of how the model is performing in the real world. It then compares these metrics against predefined thresholds. If performance has dropped below the threshold or new data row reach the target, the DAG can trigger a model retraining process (DVC start from preprocessing stage). If neither condition is met, the pipeline will **defer retraining**. In this case, any new data that has arrived is stored in a **temporary data repository.**