

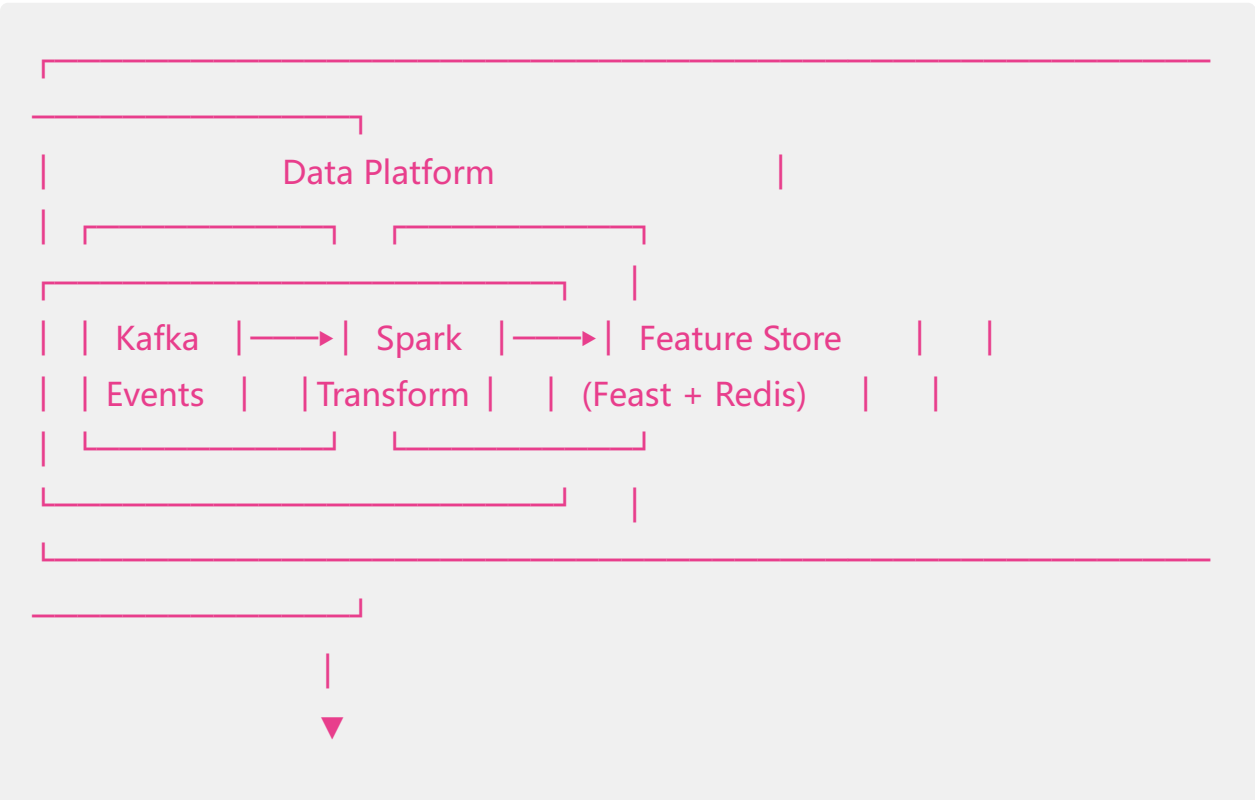
Uber Unified ML Platform - Complete Documentation

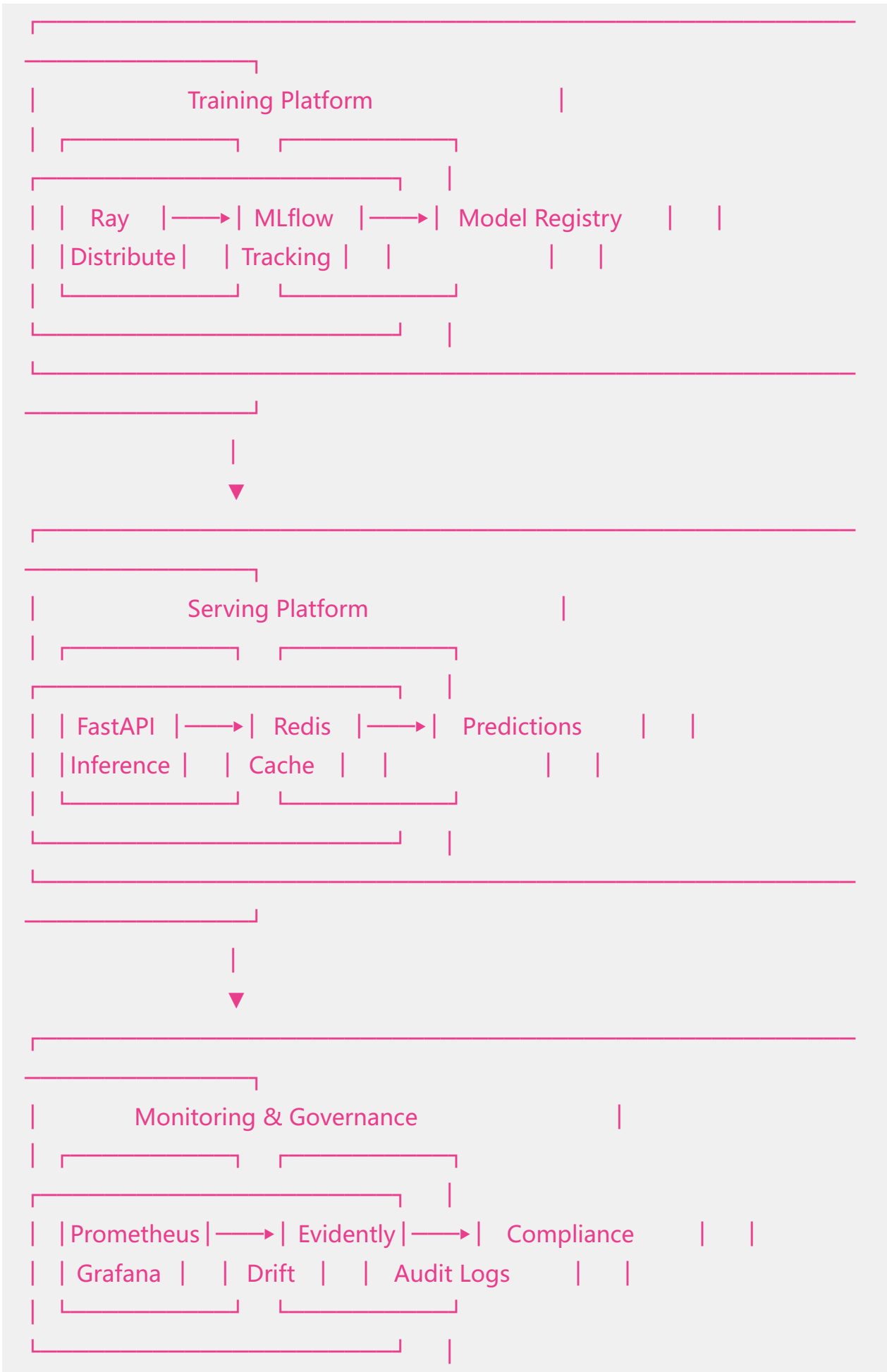
Table of Contents

- 1. [Architecture Overview](#)
- 2. [Components](#)
- 3. [Deployment Guide](#)
- 4. [Development Guide](#)
- 5. [Operations Guide](#)
- 6. [API Reference](#)

Architecture Overview

The Uber Unified ML Platform is a production-grade system for the complete ML lifecycle:







Components

1. Data Platform

Purpose: Real-time and batch data ingestion and feature engineering

Components:

- **Kafka:** Event streaming for rider, driver, and trip events
- **Airflow:** Orchestration of batch pipelines
- **Spark:** Distributed data transformation
- **Great Expectations:** Data quality validation

Key Files:

- `data_platform/kafka_producer.py`: Event production
- `data_platform/kafka_consumer.py`: Real-time feature engineering
- `data_platform/airflow/dags/batch_feature_pipeline.py`: Batch orchestration

2. Feature Store

Purpose: Centralized feature management with online/offline consistency

Components:

- **Feast:** Feature registry and serving
- **Redis:** Low-latency online features
- **PostgreSQL:** Offline feature storage

Key Files:

- `feature_store/feature_definitions.py`: Feature schemas
- `feature_store/feature_client.py`: Feature retrieval API

Usage Example:

```
from feature_store.feature_client import get_feature_store

fs = get_feature_store()
features = fs.get_eta_prediction_features(
    rider_id="rider_123",
    driver_id="driver_456",
    trip_id="trip_789"
)
```

3. Training Platform

Purpose: Distributed model training with experiment tracking

Components:

- **Ray:** Multi-node distributed training
- **MLflow:** Experiment tracking and model registry
- **XGBoost/PyTorch:** ML frameworks

Key Files:

- `training/ray_orchestrator.py`: Distributed training orchestration
- `training/mlflow_tracker.py`: MLflow wrapper
- `training/train_eta_model.py`: ETA prediction training

Training Example:

```
from training.train_eta_model import train_eta_model

run_id = train_eta_model(
    experiment_name="eta-prediction",
    run_name="production_v2"
)
```

4. Serving Platform

Purpose: Low-latency model inference API

Components:

- **FastAPI:** REST API for predictions
- **Redis:** Feature and prediction caching
- **Kubernetes:** Container orchestration with autoscaling

Key Files:

- `serving/main.py`: Inference API
- `serving/deployment.py`: Blue-green deployment

API Example:

```
curl -X POST "http://api.uber-ml.com/predict/eta" \  
-H "Content-Type: application/json" \  
-d '{  
  "rider_id": "rider_123",  
  "driver_id": "driver_456",  
  "trip_id": "trip_789",  
  "pickup_lat": 37.7749,  
  "pickup_lng": -122.4194,  
  "dropoff_lat": 37.8044,  
  "dropoff_lng": -122.2712  
'
```

5. Monitoring & Governance

Purpose: Model performance monitoring and regulatory compliance

Components:

- **Prometheus + Grafana:** Metrics and dashboards
- **Evidently AI:** Drift detection
- **Custom:** Compliance and audit system

Key Files:

- `monitoring/drift_detection.py`: Statistical drift detection
 - `governance/compliance.py`: Approval workflows and audit trails
-

Deployment Guide

Prerequisites

- Docker and Docker Compose
- Kubernetes cluster (EKS/GKE) or Minikube
- Terraform (for infrastructure)
- Python 3.9+
- AWS/GCP credentials

Local Development

1. Clone and Setup:

```
git clone https://github.com/uber/ml-platform.git
cd ml-platform
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

1. Start Services:

```
docker-compose up -d
```

1. Initialize Feature Store:

```
cd feature_store  
feast apply
```

1. Run Training:

```
python training/train_eta_model.py
```

1. Start Inference Service:

```
python serving/main.py
```

Production Deployment

1. Provision Infrastructure:

```
cd infrastructure/terraform  
terraform init  
terraform plan  
terraform apply
```

1. Configure kubectl:

```
aws eks update-kubeconfig --name uber-ml-platform --region us-west-2
```

1. Deploy to Kubernetes:

```
kubectl apply -f infrastructure/k8s/deployments.yaml
```

1. Verify Deployment:

```
kubectl get pods -n ml-platform
kubectl get svc -n ml-platform
```

1. Access Services:

```
# Inference API
kubectl port-forward svc/inference-service 8000:80 -n ml-platform

# Grafana Dashboard
kubectl port-forward svc/grafana-service 3000:3000 -n ml-platform

# MLflow UI
kubectl port-forward svc/mlflow-service 5000:5000 -n ml-platform
```

Development Guide

Adding a New Model

1. Define Features in `feature_store/feature_definitions.py`:

```
new_model_features = FeatureView(
    name="new_model_features",
    entities=[rider_entity, driver_entity],
    schema=[...],
    online=True,
    source=feature_source
)
```

1. Create Training Script in `training/`:

```
def train_new_model():
    # Initialize tracker
```

```
tracker = MLflowTracker(experiment_name="new-model")
run_id = tracker.start_run()

# Get features
fs = get_feature_store()
features = fs.get_historical_features(...)

# Train model
model = train(features)

# Log to MLflow
tracker.log_model(model, "model")
tracker.end_run()
```

1. **Add Inference Endpoint** in `serving/main.py`:

```
@app.post("/predict/new-model")
async def predict_new_model(request: NewModelRequest):
    features = get_features(request)
    model = load_model("new-model")
    prediction = model.predict(features)
    return NewModelResponse(prediction=prediction)
```

Running Tests

```
# Unit tests
pytest tests/test_platform.py -v

# Integration tests
pytest tests/test_platform.py -v -m integration

# Coverage report
pytest tests/ --cov=. --cov-report=html
```

Code Quality

```
# Format code
black .

# Lint
flake8 .

# Type checking
mypy .
```

Operations Guide

Monitoring

Grafana Dashboards:

- Model Performance: Latency, throughput, error rates
- Feature Store: Cache hit rates, retrieval latency
- Infrastructure: CPU, memory, GPU utilization

Alerts:

- High error rate ($> 5\%$)
- High latency ($p99 > 1s$)
- Feature drift detected
- Model performance degradation

Drift Detection

Monitor and retrain models automatically:

```
from monitoring.drift_detection import DriftDetector
```

```
detector = DriftDetector(reference_data, drift_threshold=0.1)
results = detector.detect_drift(current_data)

if results['dataset_drift']:
    trigger_retraining(model_name, results)
```

Model Deployment

Blue-Green Deployment:

```
python serving/deployment.py eta-prediction-model 5 blue_green
```

Canary Deployment:

```
python serving/deployment.py eta-prediction-model 5 canary
```

Troubleshooting

Issue: High prediction latency

- Check Redis cache hit rate
- Verify feature retrieval time
- Check model loading time

Issue: Drift detected

- Review feature distributions
- Check for data quality issues
- Trigger model retraining

Issue: Deployment failed

- Check model validation logs
 - Verify resource availability
 - Review rollback procedures
-

API Reference

Endpoints

Health Check

```
GET /health
```

ETA Prediction

```
POST /predict/eta
{
  "rider_id": "string",
  "driver_id": "string",
  "trip_id": "string",
  "pickup_lat": 37.7749,
  "pickup_lng": -122.4194,
  "dropoff_lat": 37.8044,
  "dropoff_lng": -122.2712
}
```

Surge Pricing

```
POST /predict/surge
{
  "rider_id": "string",
  "pickup_lat": 37.7749,
  "pickup_lng": -122.4194
}
```

Metrics

GET /metrics

Performance Targets

- **Feature Retrieval:** < 50ms (p99)
 - **Model Prediction:** < 200ms (p99)
 - **End-to-End API:** < 300ms (p99)
 - **Throughput:** > 10,000 QPS
 - **Availability:** 99.99%
-

Security & Compliance

- **Encryption:** At-rest (S3, RDS) and in-transit (TLS)
 - **Access Control:** IAM roles and RBAC
 - **Audit Logging:** All model operations logged
 - **Data Retention:** Automated cleanup per GDPR
 - **Compliance:** SOC2, GDPR ready
-