MLOps Pipeline with Kubeflow and MLflow:
A Comprehensive Guide to Production-Ready ML Infrastructure

Vladimir Ovcharov
MLOps Engineer & ML Systems Architect
vladimir.ovcharov@highfunk.uk

July 10, 2025

Abstract

This article presents a comprehensive guide to building production-ready machine learning pipelines using Kubeflow for orchestration and MLflow for experiment tracking. We explore the architectural patterns, implementation strategies, and best practices for creating scalable, maintainable, and robust MLOps infrastructure. The guide includes practical code examples, deployment configurations, and monitoring strategies that can be directly applied in enterprise environments.

Keywords: MLOps, Kubeflow, MLflow, Kubernetes, Machine Learning Pipeline, Experiment Tracking, Model Deployment

1 Introduction

The evolution of machine learning from research prototypes to production-ready systems represents one of the most significant challenges facing modern data science organizations. While the academic community has made tremendous strides in developing sophisticated algorithms and achieving state-of-the-art results on benchmark datasets, the transition to real-world applications introduces a complex web of operational, technical, and organizational challenges that traditional software engineering practices struggle to address effectively.

1.1 The Production Challenge

The journey from a Jupyter notebook experiment to a production machine learning system involves far more than simply deploying a trained model. Organizations typically encounter what has been termed the "ML production gap" – the substantial difference between the controlled environment of model development and the chaotic reality of production systems where models must operate reliably, scale dynamically, and maintain performance over time.

Consider the typical lifecycle of a machine learning project: data scientists begin with exploratory data analysis, experiment with various algorithms and feature engineering approaches, tune hyperparameters, and eventually arrive at a model that demonstrates promising performance on held-out test data. However, this process often occurs in isolation, using static datasets, controlled environments, and manual workflows that bear little resemblance to the dynamic, distributed, and automated systems required for production deployment.

The challenges multiply when we consider the operational requirements of production ML systems:

- **Data Pipeline Reliability:** Production systems must handle streaming data, data quality issues, schema evolution, and upstream system failures gracefully

- **Model Performance Monitoring:** Unlike traditional software where bugs are typically deterministic, ML models can degrade silently due to data drift, concept drift, or changing business conditions
- **Scalability Requirements:** Models must serve predictions at scale, often handling thousands of requests per second with strict latency requirements
- **Regulatory Compliance:** Many industries require explainability, auditability, and bias detection capabilities that are difficult to retrofit into existing systems
- **Continuous Learning:** Models must be retrained and updated regularly to maintain performance, requiring sophisticated automation and validation pipelines

1.2 The Fragmentation Problem

Traditional approaches to ML deployment have led to significant fragmentation across the ML lifecycle. Different teams often use disparate tools for data preparation, model training, validation, deployment, and monitoring. This fragmentation creates several critical issues:

Tool Proliferation: Organizations frequently find themselves managing dozens of different tools and platforms, each optimized for a specific phase of the ML lifecycle. Data engineers might use Apache Airflow for data pipelines, data scientists prefer Jupyter notebooks for experimentation, ML engineers deploy models using custom Docker containers, and operations teams monitor systems using traditional APM tools that lack ML-specific metrics.

Knowledge Silos: The fragmented toolchain often leads to knowledge silos where each team becomes expert in their specific tools but lacks understanding of the broader system. This creates bottlenecks, communication barriers, and makes it difficult to optimize the end-to-end workflow.

Reproducibility Crisis: Without standardized environments and workflows, reproducing experimental results becomes extremely challenging. Models that perform well in development may fail in production due to subtle differences in data preprocessing, library versions, or infrastructure configurations.

Deployment Bottlenecks: The handoff between data science teams and production systems often becomes a significant bottleneck. Models developed in Python on local machines must be translated into production-ready services, often requiring significant engineering effort and introducing opportunities for errors.

1.3 The MLOps Revolution

Machine Learning Operations (MLOps) has emerged as a discipline specifically designed to address these challenges by applying DevOps principles and practices to machine learning workflows. MLOps recognizes that ML systems are fundamentally different from traditional software applications and require specialized approaches to testing, deployment, monitoring, and maintenance.

The core principles of MLOps include:

1. **Automation:** Minimizing manual interventions through automated pipelines for data processing, model training, validation, and deployment
2. **Reproducibility:** Ensuring that every experiment, training run, and deployment can be exactly reproduced through proper versioning of code, data, models, and infrastructure

3. **Monitoring:** Implementing comprehensive monitoring that goes beyond traditional system metrics to include model-specific indicators like prediction drift, data quality, and business impact

4. **Collaboration:** Breaking down silos between data science, engineering, and operations teams through shared tools, processes, and vocabulary

5. **Governance:** Establishing controls and processes for model approval, deployment, and compliance with regulatory requirements

1.4 Technology Landscape and Solution Selection

The MLOps ecosystem has rapidly evolved to include numerous platforms, frameworks, and tools, each addressing different aspects of the ML production challenge. The selection of appropriate technologies requires careful consideration of organizational needs, existing infrastructure, team capabilities, and long-term strategic goals.

Kubeflow has emerged as a leading platform for ML workflows on Kubernetes, providing a comprehensive suite of tools that span the entire ML lifecycle. Built on the foundation of Kubernetes, Kubeflow inherits the scalability, reliability, and ecosystem benefits of the container orchestration platform while adding ML-specific capabilities:

- **Kubeflow Pipelines:** A platform for building and deploying portable, scalable ML workflows
- **Katib:** Automated hyperparameter tuning and neural architecture search
- **KServe:** Model serving platform with advanced features like canary deployments and multi-framework support
- **Notebooks:** Managed Jupyter notebook environments with resource allocation and sharing capabilities
- **Training Operators:** Distributed training support for TensorFlow, PyTorch, and other frameworks

MLflow complements Kubeflow by providing robust experiment tracking and model management capabilities. Originally developed by Databricks, MLflow has become a de facto standard for ML lifecycle management, offering:

- **Experiment Tracking:** Comprehensive logging of parameters, metrics, and artifacts for every experiment
- **Model Registry:** Centralized model store with versioning, staging, and annotation capabilities
- **Model Packaging:** Standardized format for packaging models with their dependencies
- **Model Serving:** Simple deployment options for various serving platforms

1.5 Integration Strategy and Benefits

The combination of Kubeflow and MLflow creates a powerful MLOps platform that addresses the full spectrum of production ML challenges. This integration strategy leverages the strengths of both platforms while mitigating their individual limitations:

Kubeflow provides the infrastructure and orchestration layer, handling the complex task of managing distributed workloads, resource allocation, and pipeline execution across Kubernetes clusters. Its native integration with Kubernetes means that ML workloads can benefit from the same scalability, reliability, and operational practices used for other cloud-native applications.

MLflow serves as the metadata and lifecycle management layer, providing the tracking, versioning, and governance capabilities essential for maintaining reproducibility and compliance in production environments. Its framework-agnostic approach ensures that teams can continue using their preferred ML libraries while benefiting from standardized lifecycle management.

The synergy between these platforms enables several key capabilities:

- **End-to-End Traceability:** Every model deployed in production can be traced back to its training data, code version, hyperparameters, and experimental results
- **Automated Retraining:** Pipelines can automatically detect model performance degradation and trigger retraining workflows with minimal human intervention
- **A/B Testing and Gradual Rollouts:** New model versions can be safely deployed using canary deployments and traffic splitting capabilities
- **Resource Optimization:** Kubernetes-native resource management ensures efficient utilization of computational resources across training and serving workloads
- **Multi-Environment Consistency:** The same pipeline definitions can be deployed across development, staging, and production environments with environment-specific configurations

1.6 Article Scope and Objectives

This comprehensive guide provides practical, implementable solutions for organizations seeking to establish robust MLOps practices using Kubeflow and MLflow. Rather than focusing on theoretical concepts, we emphasize hands-on implementation with real-world code examples, configuration files, and architectural patterns that have been proven in production environments.

The article is structured to take readers through a complete implementation journey, from initial environment setup through advanced monitoring and optimization techniques. Each section builds upon previous concepts while providing sufficient detail for independent implementation. Code examples are production-ready and include error handling, logging, and best practices gleaned from real-world deployments.

Our target audience includes ML engineers, DevOps practitioners, data scientists, and technical leaders who are responsible for moving ML systems from experimentation to production. We assume familiarity with basic ML concepts, Kubernetes fundamentals, and Python programming, but provide sufficient context for readers to understand and adapt the solutions to their specific environments.

By the end of this guide, readers will have a complete understanding of how to:

- Design and implement scalable ML pipelines using Kubeflow
- Establish comprehensive experiment tracking and model management with MLflow
- Deploy models safely and efficiently using modern serving platforms
- Monitor ML systems for performance, drift, and operational issues

- Implement automated retraining and continuous deployment workflows
- Apply security, governance, and compliance best practices

2 Implementation Guide

This section provides detailed, step-by-step instructions for implementing a production-ready MLOps platform using Kubeflow and MLflow. The implementation follows a progressive approach, starting with foundational infrastructure and gradually building up to advanced features. Each step includes comprehensive code examples, configuration files, and troubleshooting guidance based on real-world deployment experiences.

2.1 Prerequisites and Environment Preparation

Before beginning the implementation, ensure that your environment meets the necessary requirements and that all prerequisite tools are properly configured.

2.1.1 Infrastructure Requirements

Kubernetes Cluster Specifications:

- **Minimum cluster size:** 3 nodes with 4 CPU cores and 16GB RAM each
- **Recommended cluster size:** 5+ nodes with 8 CPU cores and 32GB RAM each
- **Kubernetes version:** 1.24 or later (tested up to 1.28)
- **Storage:** Dynamic volume provisioning with SSD-backed storage classes
- **Network:** CNI-compatible networking (Calico, Flannel, or cloud provider CNI)
- **Load Balancer:** Cloud provider load balancer or MetalLB for on-premises

Additional Infrastructure Components:

- Object storage (AWS S3, Google Cloud Storage, Azure Blob, or MinIO)
- Container registry (Docker Hub, ECR, GCR, or Harbor)
- DNS management for custom domains and SSL certificates
- Monitoring infrastructure (Prometheus operator recommended)

2.1.2 Required Tools and Dependencies

Install and configure the following tools on your management workstation:

```bash
#!/bin/bash

# Install kubectl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

# Install Helm
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

```

```
# Install kustomize
curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh" | bash
sudo mv kustomize /usr/local/bin/

# Install yq for YAML processing
sudo wget -qO /usr/local/bin/yq https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64
sudo chmod +x /usr/local/bin/yq

# Install kubens and kubectx for easier cluster management
sudo git clone https://github.com/ahmetb/kubectx /opt/kubectx
sudo ln -s /opt/kubectx/kubectx /usr/local/bin/kubectx
sudo ln -s /opt/kubectx/kubens /usr/local/bin/kubens

# Verify installations
echo "Verifying tool installations..."
kubectl version --client
helm version
kustomize version
yq --version
```

Listing 1: Tool Installation Script

2.2 Kubeflow Installation and Configuration

Kubeflow installation requires careful attention to component selection and configuration to ensure optimal performance and security.

2.2.1 Namespace and Security Setup

Begin by creating the necessary namespaces and security configurations:

```yaml
# kubeflow-namespaces.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: kubeflow
  labels:
    control-plane: kubeflow
    istio-injection: enabled
---
apiVersion: v1
kind: Namespace
metadata:
  name: kubeflow-user-example-com
  labels:
    control-plane: kubeflow
    istio-injection: enabled
    user: example@example.com
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kubeflow-admin
  namespace: kubeflow
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kubeflow-admin
roleRef:
```

```
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kubeflow-admin
  namespace: kubeflow
```

Listing 2: Namespace and RBAC Configuration

2.2.2 Kubeflow Manifests Installation

Install Kubeflow using the official manifests with customizations for production environments:

```
#!/bin/bash

set -e

# Configuration variables
export KF_VERSION="v1.7.0"
export KF_NAME="mlops-platform"
export BASE_DIR="${HOME}/kubeflow"
export KF_DIR="${BASE_DIR}/${KF_NAME}"
export CONFIG_URI="https://raw.githubusercontent.com/kubeflow/manifests/${
    KF_VERSION}/kfdef/kfctl_k8s_istio.v1.7.0.yaml"

# Create directory structure
mkdir -p ${KF_DIR}
cd ${KF_DIR}

# Download Kubeflow manifests
echo "Downloading Kubeflow manifests..."
wget -O kubeflow-manifests.tar.gz \
  "https://github.com/kubeflow/manifests/archive/${KF_VERSION}.tar.gz"
tar -xzf kubeflow-manifests.tar.gz
cd manifests-${KF_VERSION#v}

# Apply custom configurations
echo "Applying custom configurations..."
cat > custom-config.yaml << EOF
apiVersion: kfdef.apps.kubeflow.org/v1
kind: KfDef
metadata:
  name: ${KF_NAME}
  namespace: kubeflow
spec:
  applications:
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: stacks/kubernetes/application/istio-1-16
    name: istio-1-16
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: stacks/kubernetes/application/cluster-local-gateway-1-16
    name: cluster-local-gateway-1-16
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: apps/pipeline/upstream/env/cert-manager/platform-agnostic-multi-
    user
    name: kubeflow-pipelines
```

```
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: apps/jupyter/jupyter-web-app/upstream/overlays/istio
    name: jupyter-web-app
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: apps/katib/upstream/installs/katib-with-kubeflow
    name: katib
  - kustomizeConfig:
      repoRef:
        name: manifests
        path: apps/training-operator/upstream/overlays/kubeflow
    name: training-operator
  repos:
  - name: manifests
    uri: https://github.com/kubeflow/manifests/archive/${KF_VERSION}.tar.gz
EOF

# Install cert-manager first (required for webhooks)
echo "Installing cert-manager..."
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download
    /v1.12.0/cert-manager.yaml
kubectl wait --for=condition=ready pod -l app=cert-manager -n cert-manager --
    timeout=300s

# Install Kubeflow components
echo "Installing Kubeflow components..."
while ! kustomize build example | kubectl apply -f -; do
  echo "Retrying to apply resources"
  sleep 10
done

# Wait for deployments to be ready
echo "Waiting for Kubeflow components to be ready..."
kubectl wait --for=condition=ready pod -l app=istiod -n istio-system --timeout
    =300s
kubectl wait --for=condition=ready pod -l app=istio-proxy -n kubeflow --timeout
    =300s

echo "Kubeflow installation completed successfully!"
```

Listing 3: Kubeflow Installation Process

2.2.3 Kubeflow Post-Installation Configuration

Configure Kubeflow for production use with proper security and resource management:

```
# kubeflow-production-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: kubeflow-config
  namespace: kubeflow
data:
  # Pipeline configuration
  pipeline-config.yaml: |
    executorImage: gcr.io/ml-pipeline/api-server:2.0.0
    cacheEnabled: true
    cacheDatabase:
      host: mysql.kubeflow.svc.cluster.local
      port: "3306"
```

```
      database: mlpipeline
    objectStore:
      endpoint: minio-service.kubeflow.svc.cluster.local:9000
      bucket: mlpipeline
      accessKey: minio
      secretKey: minio123

  # Resource limits
  resource-limits.yaml: |
    defaultRequests:
      cpu: "100m"
      memory: "128Mi"
    defaultLimits:
      cpu: "1000m"
      memory: "1Gi"
    maxRequests:
      cpu: "8000m"
      memory: "16Gi"
---
apiVersion: v1
kind: Secret
metadata:
  name: kubeflow-secrets
  namespace: kubeflow
type: Opaque
data:
  # Base64 encoded values
  mysql-password: bWxwaXBlbGluZQ==  # mlpipeline
  minio-access-key: bWluaW8=        # minio
  minio-secret-key: bWluaW8xMjM=    # minio123
```

Listing 4: Production Kubeflow Configuration

2.3 MLflow Setup and Integration

MLflow serves as the experiment tracking and model registry backbone of our MLOps platform. This section covers both standalone MLflow deployment and integration with Kubeflow.

2.3.1 MLflow Server Deployment

Deploy MLflow server with persistent storage and database backend:

```
# mlflow-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mlflow-server
  namespace: kubeflow
  labels:
    app: mlflow-server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mlflow-server
  template:
    metadata:
      labels:
        app: mlflow-server
    spec:
      containers:
      - name: mlflow-server
```

```yaml
        image: python:3.9-slim
        ports:
        - containerPort: 5000
        env:
        - name: MLFLOW_S3_ENDPOINT_URL
          value: "http://minio-service.kubeflow.svc.cluster.local:9000"
        - name: AWS_ACCESS_KEY_ID
          valueFrom:
            secretKeyRef:
              name: mlflow-secrets
              key: aws-access-key-id
        - name: AWS_SECRET_ACCESS_KEY
          valueFrom:
            secretKeyRef:
              name: mlflow-secrets
              key: aws-secret-access-key
        - name: MLFLOW_DATABASE_URI
          valueFrom:
            secretKeyRef:
              name: mlflow-secrets
              key: database-uri
        command:
        - /bin/bash
        - -c
        - |
          pip install mlflow[extras]==2.7.1 psycopg2-binary boto3
          mlflow server \
            --host 0.0.0.0 \
            --port 5000 \
            --backend-store-uri ${MLFLOW_DATABASE_URI} \
            --default-artifact-root s3://mlflow-artifacts/ \
            --serve-artifacts
        resources:
          requests:
            cpu: 100m
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 1Gi
        livenessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: mlflow-server
  namespace: kubeflow
spec:
  selector:
    app: mlflow-server
  ports:
  - port: 5000
    targetPort: 5000
```

```
    protocol: TCP
  type: ClusterIP
---
apiVersion: v1
kind: Secret
metadata:
  name: mlflow-secrets
  namespace: kubeflow
type: Opaque
data:
  aws-access-key-id: bWluaW8=  # minio
  aws-secret-access-key: bWluaW8xMjM=  # minio123
  database-uri:
    cG9zdGdyZXNxbDovL21sZmxvdzptbGZsb3dAcG9zdGdyZXNxbC5rdWJlZmxvdy5zdmMuY2x1c3Rlci5sb2NhbDo1N
    ==
```

Listing 5: MLflow Server Deployment

2.3.2 Database and Storage Configuration

Set up PostgreSQL database and MinIO object storage for MLflow:

```
# postgresql-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql
  namespace: kubeflow
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgresql
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      containers:
      - name: postgresql
        image: postgres:13
        env:
        - name: POSTGRES_DB
          value: mlflow
        - name: POSTGRES_USER
          value: mlflow
        - name: POSTGRES_PASSWORD
          value: mlflow
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: postgresql-storage
          mountPath: /var/lib/postgresql/data
        resources:
          requests:
            cpu: 100m
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 1Gi
      volumes:
```

```yaml
      - name: postgresql-storage
        persistentVolumeClaim:
          claimName: postgresql-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: postgresql
  namespace: kubeflow
spec:
  selector:
    app: postgresql
  ports:
  - port: 5432
    targetPort: 5432
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgresql-pvc
  namespace: kubeflow
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: fast-ssd
---
# minio-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: minio
  namespace: kubeflow
spec:
  replicas: 1
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        image: minio/minio:RELEASE.2023-09-04T19-57-37Z
        args:
        - server
        - /data
        - --console-address=:9001
        env:
        - name: MINIO_ROOT_USER
          value: minio
        - name: MINIO_ROOT_PASSWORD
          value: minio123
        ports:
        - containerPort: 9000
        - containerPort: 9001
        volumeMounts:
        - name: minio-storage
          mountPath: /data
```

```
        resources:
          requests:
            cpu: 100m
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 1Gi
      volumes:
      - name: minio-storage
        persistentVolumeClaim:
          claimName: minio-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: minio-service
  namespace: kubeflow
spec:
  selector:
    app: minio
  ports:
  - name: api
    port: 9000
    targetPort: 9000
  - name: console
    port: 9001
    targetPort: 9001
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: minio-pvc
  namespace: kubeflow
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
  storageClassName: fast-ssd
```

Listing 6: MLflow Storage Infrastructure

2.4 Network Configuration and Security

Proper network configuration ensures secure communication between components while maintaining performance and accessibility.

2.4.1 Istio Service Mesh Configuration

Configure Istio for secure service-to-service communication:

```
# istio-security-policies.yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: kubeflow
spec:
  mtls:
    mode: STRICT
---
```

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: mlflow-access
  namespace: kubeflow
spec:
  selector:
    matchLabels:
      app: mlflow-server
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/kubeflow/sa/default"]
    - source:
        namespaces: ["kubeflow-user-example-com"]
  - to:
    - operation:
        methods: ["GET", "POST", "PUT", "DELETE"]
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: mlflow-vs
  namespace: kubeflow
spec:
  hosts:
  - mlflow.example.com
  gateways:
  - kubeflow-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        host: mlflow-server.kubeflow.svc.cluster.local
        port:
          number: 5000
    timeout: 300s
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: kubeflow-gateway
  namespace: kubeflow
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*.example.com"
    tls:
      httpsRedirect: true
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - "*.example.com"
```

```
    tls:
      mode: SIMPLE
      credentialName: kubeflow-tls-secret
```

Listing 7: Istio Security Configuration

2.5 Monitoring and Observability Setup

Comprehensive monitoring is essential for maintaining platform health and performance.

2.5.1 Prometheus and Grafana Configuration

Deploy monitoring stack with custom dashboards for MLOps metrics:

```
# monitoring-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
  labels:
    name: monitoring
---
# prometheus-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s

    rule_files:
      - "mlops_rules.yml"

    scrape_configs:
    - job_name: 'kubeflow-pipelines'
      kubernetes_sd_configs:
      - role: pod
        namespaces:
          names:
          - kubeflow
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_label_app]
        action: keep
        regex: ml-pipeline.*

    - job_name: 'mlflow-server'
      kubernetes_sd_configs:
      - role: pod
        namespaces:
          names:
          - kubeflow
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_label_app]
        action: keep
        regex: mlflow-server

    - job_name: 'model-servers'
      kubernetes_sd_configs:
      - role: pod
```

```
        namespaces:
          names:
          - kubeflow-user-example-com
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_annotation_serving_kserve_io_inferenceservice]
        action: keep
        regex: .+

  mlops_rules.yml: |
    groups:
    - name: mlops.rules
      rules:
      - alert: ModelServerDown
        expr: up{job="model-servers"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Model server {{ $labels.instance }} is down"
          description: "Model server has been down for more than 1 minute"

      - alert: HighModelLatency
        expr: histogram_quantile(0.95, rate(model_request_duration_seconds_bucket[5m])) > 1
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High model inference latency detected"
          description: "95th percentile latency is {{ $value }}s"

      - alert: MLflowServerDown
        expr: up{job="mlflow-server"} == 0
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "MLflow server is down"
          description: "MLflow server has been unreachable for more than 2 minutes"
```

Listing 8: Monitoring Stack Deployment

2.6 Initial Platform Validation

After completing the installation, validate that all components are functioning correctly:

```bash
#!/bin/bash

set -e

echo "=== MLOps Platform Validation ==="

# Check namespace status
echo "Checking namespace status..."
kubectl get namespaces kubeflow kubeflow-user-example-com monitoring

# Check Kubeflow components
echo "Checking Kubeflow components..."
kubectl get pods -n kubeflow | grep -E "(Running|Completed)" || exit 1

# Check MLflow server
```

```
echo "Checking MLflow server..."
kubectl get pods -n kubeflow -l app=mlflow-server
kubectl wait --for=condition=ready pod -l app=mlflow-server -n kubeflow --
    timeout=300s

# Test MLflow API
echo "Testing MLflow API..."
MLF_HOST=$(kubectl get svc mlflow-server -n kubeflow -o jsonpath='{.spec.
    clusterIP}')
kubectl run test-pod --rm -i --tty --image=curlimages/curl -- \
  curl -f http://${MLF_HOST}:5000/health || exit 1

# Check storage components
echo "Checking storage components..."
kubectl get pods -n kubeflow -l app=postgresql
kubectl get pods -n kubeflow -l app=minio

# Check Istio configuration
echo "Checking Istio configuration..."
kubectl get virtualservices,gateways -n kubeflow

# Test pipeline functionality
echo "Testing pipeline functionality..."
kubectl get workflows -n kubeflow-user-example-com || echo "No workflows found
    (expected for new installation)"

echo "=== Platform validation completed successfully! ==="
echo ""
echo "Access URLs (configure DNS or port-forward):"
echo "- Kubeflow Central Dashboard: https://kubeflow.example.com"
echo "- MLflow UI: https://mlflow.example.com"
echo "- MinIO Console: https://minio.example.com"
echo ""
echo "Next steps:"
echo "1. Configure DNS entries for your domain"
echo "2. Set up SSL certificates"
echo "3. Create user profiles and RBAC policies"
echo "4. Run your first ML pipeline"
```

Listing 9: Platform Validation Script

This implementation guide provides a solid foundation for deploying a production-ready MLOps platform. The next sections will cover advanced pipeline development, model serving configurations, and operational best practices.

3 Model Deployment and Serving

Model deployment and serving represent critical phases in the MLOps lifecycle where trained models transition from experimental artifacts to production systems that deliver real business value. This section provides comprehensive guidance for implementing robust, scalable, and secure model serving infrastructure using KServe, along with advanced deployment patterns that ensure safe model releases and optimal performance.

3.1 KServe Architecture and Integration

KServe (formerly KFServing) serves as the cornerstone of our model serving strategy, providing a Kubernetes-native platform for deploying and managing machine learning models at scale. Built on the foundation of Knative Serving, KServe inherits powerful capabilities for autoscaling, traffic management, and serverless deployment patterns while adding ML-specific functionality.

3.1.1 KServe Core Components

Understanding KServe's architecture is essential for effective model deployment and troubleshooting:

InferenceService: The primary custom resource that defines how models should be served, including predictor configurations, transformers, and explainers.

Model Server: Framework-specific serving runtimes that handle model loading, inference request processing, and response formatting. KServe supports multiple built-in servers including:

- **Scikit-learn Server:** Optimized for traditional ML models with pickle format support
- **TensorFlow Serving:** High-performance serving for TensorFlow models with batching and GPU support
- **PyTorch Server (TorchServe):** Native PyTorch model serving with custom handler support
- **XGBoost Server:** Specialized serving for gradient boosting models
- **Custom Predictors:** User-defined serving logic for complex inference pipelines

Data Plane: Handles actual inference requests and responses, implementing the KServe v1 and v2 inference protocols for standardized communication.

Control Plane: Manages the lifecycle of inference services, including deployment, scaling, and traffic routing decisions.

3.1.2 KServe Installation and Configuration

Deploy KServe with production-ready configurations:

```bash
#!/bin/bash

set -e

echo "Installing KServe with dependencies..."

# Install Knative Serving (required for KServe)
echo "Installing Knative Serving..."
kubectl apply -f https://github.com/knative/serving/releases/download/knative-v1.11.0/serving-crds.yaml
kubectl apply -f https://github.com/knative/serving/releases/download/knative-v1.11.0/serving-core.yaml

# Install Knative Istio controller
kubectl apply -f https://github.com/knative/net-istio/releases/download/knative-v1.11.0/net-istio.yaml

# Configure Knative Serving
kubectl patch configmap/config-network \
  --namespace knative-serving \
  --type merge \
  --patch '{"data":{"ingress-class":"istio.ingress.networking.knative.dev"}}'

# Set domain configuration
kubectl patch configmap/config-domain \
  --namespace knative-serving \
  --type merge \
  --patch '{"data":{"example.com":""}}'

# Install KServe CRDs and controllers
```

```
echo "Installing KServe..."
kubectl apply -f https://github.com/kserve/kserve/releases/download/v0.11.0/
    kserve.yaml

# Install KServe built-in ClusterServingRuntimes
kubectl apply -f https://github.com/kserve/kserve/releases/download/v0.11.0/
    kserve-runtimes.yaml

# Wait for KServe controller to be ready
echo "Waiting for KServe controller..."
kubectl wait --for=condition=ready pod -l control-plane=kserve-controller-
    manager -n kserve --timeout=300s

echo "KServe installation completed successfully!"
```

Listing 10: KServe Installation Script

3.1.3 Production KServe Configuration

Configure KServe for production environments with proper resource management and security:

```
# kserve-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: inferenceservice-config
  namespace: kserve
data:
  predictors: |
    {
      "tensorflow": {
        "image": "tensorflow/serving:2.13.0",
        "defaultImageVersion": "2.13.0",
        "defaultGpuImageVersion": "2.13.0-gpu",
        "supportedFrameworks": ["tensorflow"],
        "multiModelServer": false
      },
      "pytorch": {
        "image": "pytorch/torchserve:0.8.2-cpu",
        "defaultImageVersion": "0.8.2-cpu",
        "defaultGpuImageVersion": "0.8.2-gpu",
        "supportedFrameworks": ["pytorch"],
        "multiModelServer": false
      },
      "sklearn": {
        "image": "kserve/sklearnserver:v0.11.0",
        "defaultImageVersion": "v0.11.0",
        "supportedFrameworks": ["sklearn"],
        "multiModelServer": true
      },
      "xgboost": {
        "image": "kserve/xgbserver:v0.11.0",
        "defaultImageVersion": "v0.11.0",
        "supportedFrameworks": ["xgboost"],
        "multiModelServer": true
      }
    }

  transformer: |
    {
      "feast": {
        "image": "kserve/feast-transformer:v0.11.0",
        "defaultImageVersion": "v0.11.0"
```

```
      }
    }

  explainer: |
    {
      "alibi": {
        "image": "kserve/alibi-explainer:v0.11.0",
        "defaultImageVersion": "v0.11.0"
      }
    }

  storageInitializer: |
    {
      "image": "kserve/storage-initializer:v0.11.0",
      "memoryRequest": "100Mi",
      "memoryLimit": "1Gi",
      "cpuRequest": "100m",
      "cpuLimit": "1000m"
    }

  credentials: |
    {
      "gcs": {
        "gcsCredentialFileName": "gcloud-application-credentials.json"
      },
      "s3": {
        "s3AccessKeyIDName": "AWS_ACCESS_KEY_ID",
        "s3SecretAccessKeyName": "AWS_SECRET_ACCESS_KEY",
        "s3Endpoint": "",
        "s3UseHttps": true,
        "s3Region": "us-west-1",
        "s3VerifySSL": true,
        "s3UseVirtualBucket": false,
        "s3UseAnonymousCredential": false,
        "s3CABundle": ""
      }
    }

  ingress: |
    {
      "ingressGateway": "kubeflow/kubeflow-gateway",
      "ingressService": "istio-ingressgateway.istio-system.svc.cluster.local",
      "localGateway": "knative-serving/knative-local-gateway",
      "localGatewayService": "knative-local-gateway.istio-system.svc.cluster.local",
      "ingressDomain": "example.com",
      "ingressClassName": "istio",
      "domainTemplate": "{{.Name}}-{{.Namespace}}.{{.IngressDomain}}",
      "urlScheme": "https",
      "disableIstioVirtualHost": false
    }

  deploy: |
    {
      "defaultDeploymentMode": "Serverless",
      "progressDeadlineSeconds": 600,
      "defaultCpuRequest": "100m",
      "defaultMemoryRequest": "128Mi",
      "defaultCpuLimit": "1000m",
      "defaultMemoryLimit": "2Gi"
    }
---
apiVersion: v1
```

```yaml
kind: ConfigMap
metadata:
  name: kserve-logger-config
  namespace: kserve
data:
  logger.properties: |
    # Root logger option
    log4j.rootLogger=INFO, stdout

    # Direct log messages to stdout
    log4j.appender.stdout=org.apache.log4j.ConsoleAppender
    log4j.appender.stdout.Target=System.out
    log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
    log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
     %c{1}:%L - %m%n

    # Suppress unnecessary logs
    log4j.logger.org.apache.hadoop=WARN
    log4j.logger.org.apache.spark=WARN
    log4j.logger.org.eclipse.jetty=WARN
    log4j.logger.org.apache.kafka=WARN
```

Listing 11: KServe Production Configuration

3.2 Model Packaging and Registry Integration

Effective model deployment requires standardized packaging and integration with model registries to ensure consistency, traceability, and reproducibility.

3.2.1 MLflow Model Integration

Create a comprehensive MLflow integration that automatically packages models for KServe deployment:

```python
import mlflow
import mlflow.sklearn
import mlflow.pytorch
import mlflow.tensorflow
import yaml
import json
import os
from typing import Dict, Any, Optional
from kubernetes import client, config
from datetime import datetime
import logging

class MLflowKServeIntegration:
    """
    Handles integration between MLflow model registry and KServe deployments
    """

    def __init__(self,
                 mlflow_uri: str,
                 namespace: str = "kubeflow-user-example-com",
                 kserve_domain: str = "example.com"):
        self.mlflow_uri = mlflow_uri
        self.namespace = namespace
        self.kserve_domain = kserve_domain

        # Initialize MLflow client
        mlflow.set_tracking_uri(mlflow_uri)
        self.mlflow_client = mlflow.MlflowClient()
```

```

        # Initialize Kubernetes client
        try:
            config.load_incluster_config()
        except:
            config.load_kube_config()

        self.k8s_client = client.ApiClient()
        self.custom_client = client.CustomObjectsApi()

        # Configure logging
        logging.basicConfig(level=logging.INFO)
        self.logger = logging.getLogger(__name__)

    def get_model_info(self, model_name: str, version: Optional[str] = None) ->
     Dict[str, Any]:
        """
        Retrieve comprehensive model information from MLflow registry
        """
        try:
            if version is None:
                # Get latest version
                latest_versions = self.mlflow_client.get_latest_versions(
                    model_name, stages=["Production", "Staging"]
                )
                if not latest_versions:
                    raise ValueError(f"No versions found for model {model_name}
    ")
                model_version = latest_versions[0]
            else:
                model_version = self.mlflow_client.get_model_version(model_name
    , version)

            # Get run information
            run = self.mlflow_client.get_run(model_version.run_id)

            # Extract model artifacts
            model_uri = f"models:/{model_name}/{model_version.version}"

            return {
                "name": model_name,
                "version": model_version.version,
                "stage": model_version.current_stage,
                "model_uri": model_uri,
                "run_id": model_version.run_id,
                "framework": self._detect_framework(run),
                "artifacts": run.data.tags.get("mlflow.log-model.history", "[]"
    ),
                "metrics": run.data.metrics,
                "params": run.data.params,
                "tags": run.data.tags,
                "creation_timestamp": model_version.creation_timestamp,
                "last_updated_timestamp": model_version.last_updated_timestamp
            }

        except Exception as e:
            self.logger.error(f"Error retrieving model info: {str(e)}")
            raise

    def _detect_framework(self, run) -> str:
        """
        Detect ML framework from run artifacts and tags
        """
```

```python
        artifacts = json.loads(run.data.tags.get("mlflow.log-model.history", "[]"))

        for artifact in artifacts:
            if "sklearn" in artifact.get("flavors", {}):
                return "sklearn"
            elif "pytorch" in artifact.get("flavors", {}):
                return "pytorch"
            elif "tensorflow" in artifact.get("flavors", {}):
                return "tensorflow"
            elif "xgboost" in artifact.get("flavors", {}):
                return "xgboost"

        # Fallback to run tags
        if "mlflow.source.type" in run.data.tags:
            source_type = run.data.tags["mlflow.source.type"].lower()
            if "sklearn" in source_type:
                return "sklearn"
            elif "pytorch" in source_type:
                return "pytorch"
            elif "tensorflow" in source_type:
                return "tensorflow"

        return "sklearn"  # Default fallback

    def create_inference_service(self,
                                model_name: str,
                                model_version: Optional[str] = None,
                                service_name: Optional[str] = None,
                                resources: Optional[Dict[str, Any]] = None,
                                autoscaling: Optional[Dict[str, Any]] = None,
                                canary_percent: int = 0) -> Dict[str, Any]:
        """
        Create KServe InferenceService from MLflow model
        """

        # Get model information
        model_info = self.get_model_info(model_name, model_version)

        if service_name is None:
            service_name = f"{model_name.lower().replace('_', '-')}-v{model_info['version']}"

        # Default resource configuration
        default_resources = {
            "requests": {"cpu": "100m", "memory": "256Mi"},
            "limits": {"cpu": "1000m", "memory": "2Gi"}
        }
        if resources:
            default_resources.update(resources)

        # Default autoscaling configuration
        default_autoscaling = {
            "minReplicas": 1,
            "maxReplicas": 10,
            "targetUtilizationPercentage": 70,
            "scaleToZeroGracePeriod": "30s",
            "scaleDownDelay": "0s",
            "stableWindow": "60s"
        }
        if autoscaling:
            default_autoscaling.update(autoscaling)

```

```
        # Create InferenceService specification
        inference_service = {
            "apiVersion": "serving.kserve.io/v1beta1",
            "kind": "InferenceService",
            "metadata": {
                "name": service_name,
                "namespace": self.namespace,
                "labels": {
                    "model-name": model_name,
                    "model-version": str(model_info['version']),
                    "model-stage": model_info['stage'],
                    "framework": model_info['framework'],
                    "managed-by": "mlflow-kserve-integration"
                },
                "annotations": {
                    "mlflow.model.uri": model_info['model_uri'],
                    "mlflow.run.id": model_info['run_id'],
                    "deployment.timestamp": datetime.utcnow().isoformat(),
                    "serving.kserve.io/deploymentMode": "Serverless"
                }
            },
            "spec": {
                "predictor": {
                    model_info['framework']: {
                        "storageUri": model_info['model_uri'],
                        "resources": default_resources,
                        "env": [
                            {
                                "name": "STORAGE_URI",
                                "value": model_info['model_uri']
                            },
                            {
                                "name": "MODEL_NAME",
                                "value": model_name
                            }
                        ]
                    }
                }
            }
        }

        # Add canary configuration if specified
        if canary_percent > 0:
            inference_service["spec"]["predictor"]["canaryTrafficPercent"] = canary_percent

        # Add autoscaling annotations
        inference_service["metadata"]["annotations"].update({
            "autoscaling.knative.dev/minScale": str(default_autoscaling["minReplicas"]),
            "autoscaling.knative.dev/maxScale": str(default_autoscaling["maxReplicas"]),
            "autoscaling.knative.dev/target": str(default_autoscaling["targetUtilizationPercentage"]),
            "autoscaling.knative.dev/scaleToZeroGracePeriod": default_autoscaling["scaleToZeroGracePeriod"],
            "autoscaling.knative.dev/scaleDownDelay": default_autoscaling["scaleDownDelay"],
            "autoscaling.knative.dev/window": default_autoscaling["stableWindow"]
        })

        # Deploy to Kubernetes
```

```
        try:
            response = self.custom_client.create_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.namespace,
                plural="inferenceservices",
                body=inference_service
            )

            self.logger.info(f"Successfully created InferenceService: {
    service_name}")

            # Update MLflow model stage if deploying to production
            if model_info['stage'] != "Production":
                self.mlflow_client.transition_model_version_stage(
                    name=model_name,
                    version=model_info['version'],
                    stage="Production",
                    archive_existing_versions=False
                )

            return {
                "service_name": service_name,
                "namespace": self.namespace,
                "model_info": model_info,
                "inference_service": response,
                "endpoint_url": f"https://{service_name}-{self.namespace}.{self
    .kserve_domain}"
            }

        except Exception as e:
            self.logger.error(f"Failed to create InferenceService: {str(e)}")
            raise

    def update_traffic_split(self,
                             service_name: str,
                             traffic_config: Dict[str, int]) -> Dict[str, Any]:
        """
        Update traffic splitting between model versions
        """
        try:
            # Get current InferenceService
            current_service = self.custom_client.get_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.namespace,
                plural="inferenceservices",
                name=service_name
            )

            # Update traffic configuration
            if "canary" in traffic_config:
                patch_body = {
                    "spec": {
                        "predictor": {
                            "canaryTrafficPercent": traffic_config["canary"]
                        }
                    }
                }

                response = self.custom_client.patch_namespaced_custom_object(
                    group="serving.kserve.io",
                    version="v1beta1",
```

```
                    namespace=self.namespace,
                    plural="inferenceservices",
                    name=service_name,
                    body=patch_body
                )

                self.logger.info(f"Updated traffic split for {service_name}: {
    traffic_config}")
                return response

        except Exception as e:
            self.logger.error(f"Failed to update traffic split: {str(e)}")
            raise

    def rollback_deployment(self, service_name: str, target_version: str) ->
    Dict[str, Any]:
        """
        Rollback deployment to a previous model version
        """
        try:
            # Get service metadata to extract model info
            current_service = self.custom_client.get_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.namespace,
                plural="inferenceservices",
                name=service_name
            )

            model_name = current_service["metadata"]["labels"]["model-name"]

            # Get target version info
            target_model_info = self.get_model_info(model_name, target_version)

            # Update the service to use target version
            patch_body = {
                "spec": {
                    "predictor": {
                        target_model_info['framework']: {
                            "storageUri": target_model_info['model_uri']
                        }
                    }
                },
                "metadata": {
                    "labels": {
                        "model-version": str(target_model_info['version'])
                    },
                    "annotations": {
                        "mlflow.model.uri": target_model_info['model_uri'],
                        "mlflow.run.id": target_model_info['run_id'],
                        "rollback.timestamp": datetime.utcnow().isoformat(),
                        "rollback.target.version": target_version
                    }
                }
            }

            response = self.custom_client.patch_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.namespace,
                plural="inferenceservices",
                name=service_name,
                body=patch_body
```

```
            )

            self.logger.info(f"Successfully rolled back {service_name} to
    version {target_version}")
            return response

        except Exception as e:
            self.logger.error(f"Failed to rollback deployment: {str(e)}")
            raise

# Usage example
if __name__ == "__main__":
    # Initialize integration
    integration = MLflowKServeIntegration(
        mlflow_uri="http://mlflow-server.kubeflow.svc.cluster.local:5000",
        namespace="kubeflow-user-example-com"
    )

    # Deploy a model
    deployment_result = integration.create_inference_service(
        model_name="fraud-detection-model",
        resources={
            "requests": {"cpu": "200m", "memory": "512Mi"},
            "limits": {"cpu": "2000m", "memory": "4Gi"}
        },
        autoscaling={
            "minReplicas": 2,
            "maxReplicas": 20,
            "targetUtilizationPercentage": 80
        }
    )

    print(f"Model deployed successfully: {deployment_result['endpoint_url']}")
```

Listing 12: MLflow KServe Integration

3.3 Advanced Deployment Patterns

Production model deployment requires sophisticated strategies to minimize risk while ensuring continuous service availability. This section covers advanced deployment patterns that enable safe model releases.

3.3.1 Canary Deployments

Implement gradual rollout of new model versions with automated monitoring and rollback capabilities:

```
import asyncio
import logging
from typing import Dict, List, Optional
from dataclasses import dataclass
from datetime import datetime, timedelta
import numpy as np
from prometheus_client.parser import text_string_to_metric_families
import aiohttp
import yaml

@dataclass
class CanaryConfig:
    """Configuration for canary deployment"""
    initial_traffic_percent: int = 5
    increment_percent: int = 10
```

```
    max_traffic_percent: int = 50
    evaluation_duration_minutes: int = 10
    success_rate_threshold: float = 0.99
    latency_threshold_ms: float = 1000
    error_rate_threshold: float = 0.01
    auto_promote: bool = True
    auto_rollback: bool = True

@dataclass
class DeploymentMetrics:
    """Metrics for deployment evaluation"""
    success_rate: float
    avg_latency_ms: float
    error_rate: float
    request_count: int
    timestamp: datetime

class CanaryDeploymentController:
    """
    Automated canary deployment controller with monitoring and decision making
    """

    def __init__(self,
                 kserve_integration: MLflowKServeIntegration,
                 prometheus_url: str,
                 config: CanaryConfig):
        self.kserve_integration = kserve_integration
        self.prometheus_url = prometheus_url
        self.config = config
        self.logger = logging.getLogger(__name__)

    async def deploy_canary(self,
                          model_name: str,
                          new_version: str,
                          service_name: str) -> Dict[str, Any]:
        """
        Execute automated canary deployment process
        """

        deployment_log = {
            "model_name": model_name,
            "new_version": new_version,
            "service_name": service_name,
            "start_time": datetime.utcnow(),
            "stages": [],
            "final_status": "in_progress"
        }

        try:
            # Stage 1: Deploy new version with minimal traffic
            self.logger.info(f"Starting canary deployment for {model_name} v{new_version}")

            # Create new InferenceService for canary version
            canary_service_name = f"{service_name}-canary"
            canary_deployment = self.kserve_integration.create_inference_service(
                model_name=model_name,
                model_version=new_version,
                service_name=canary_service_name,
                canary_percent=self.config.initial_traffic_percent
            )

```

```
            deployment_log["stages"].append({
                "stage": "initial_deployment",
                "traffic_percent": self.config.initial_traffic_percent,
                "timestamp": datetime.utcnow(),
                "status": "success"
            })

            # Wait for deployment to stabilize
            await asyncio.sleep(60)

            # Stage 2: Gradual traffic increase with monitoring
            current_traffic = self.config.initial_traffic_percent

            while current_traffic < self.config.max_traffic_percent:
                # Evaluate current performance
                baseline_metrics = await self.get_deployment_metrics(service_name)
                canary_metrics = await self.get_deployment_metrics(canary_service_name)

                # Make deployment decision
                decision = self._evaluate_canary_performance(baseline_metrics, canary_metrics)

                stage_log = {
                    "stage": "traffic_increase",
                    "traffic_percent": current_traffic,
                    "timestamp": datetime.utcnow(),
                    "baseline_metrics": baseline_metrics.__dict__ if baseline_metrics else None,
                    "canary_metrics": canary_metrics.__dict__ if canary_metrics else None,
                    "decision": decision
                }

                if decision["action"] == "continue":
                    # Increase traffic
                    current_traffic = min(
                        current_traffic + self.config.increment_percent,
                        self.config.max_traffic_percent
                    )

                    await self._update_traffic_split(canary_service_name, current_traffic)
                    stage_log["new_traffic_percent"] = current_traffic
                    stage_log["status"] = "success"

                    self.logger.info(f"Increased canary traffic to {current_traffic}%")

                elif decision["action"] == "rollback":
                    # Automatic rollback
                    if self.config.auto_rollback:
                        await self._rollback_canary(service_name, canary_service_name)
                        stage_log["status"] = "rollback"
                        deployment_log["final_status"] = "failed"
                        break
                    else:
                        # Manual intervention required
                        stage_log["status"] = "requires_manual_intervention"
                        deployment_log["final_status"] = "requires_intervention"
```

```python
                    break

            elif decision["action"] == "hold":
                # Hold current traffic level for extended evaluation
                self.logger.info(f"Holding canary traffic at {current_traffic}% for extended evaluation")
                await asyncio.sleep(self.config.evaluation_duration_minutes * 60 * 2)  # Extended wait
                stage_log["status"] = "hold"

            deployment_log["stages"].append(stage_log)

            # Wait for evaluation period
            await asyncio.sleep(self.config.evaluation_duration_minutes * 60)

        # Stage 3: Final evaluation and promotion decision
        if deployment_log["final_status"] == "in_progress":
            final_baseline_metrics = await self.get_deployment_metrics(service_name)
            final_canary_metrics = await self.get_deployment_metrics(canary_service_name)

            final_decision = self._evaluate_canary_performance(
                final_baseline_metrics,
                final_canary_metrics,
                final_evaluation=True
            )

            if final_decision["action"] == "promote" and self.config.auto_promote:
                # Promote canary to production
                await self._promote_canary(service_name, canary_service_name, new_version)
                deployment_log["final_status"] = "promoted"
                self.logger.info(f"Successfully promoted {model_name} v{new_version} to production")
            else:
                deployment_log["final_status"] = "requires_manual_promotion"
                self.logger.info(f"Canary deployment ready for manual promotion")

            deployment_log["stages"].append({
                "stage": "final_evaluation",
                "timestamp": datetime.utcnow(),
                "final_decision": final_decision,
                "status": deployment_log["final_status"]
            })

        deployment_log["end_time"] = datetime.utcnow()
        deployment_log["duration_minutes"] = (
            deployment_log["end_time"] - deployment_log["start_time"]
        ).total_seconds() / 60

        return deployment_log

    except Exception as e:
        self.logger.error(f"Canary deployment failed: {str(e)}")
        deployment_log["final_status"] = "error"
        deployment_log["error"] = str(e)

        # Attempt cleanup
```

```
        try:
            await self._cleanup_failed_canary(canary_service_name)
        except:
            pass

        raise

    async def get_deployment_metrics(self, service_name: str) -> Optional[DeploymentMetrics]:
        """
        Retrieve performance metrics for a deployment from Prometheus
        """
        try:
            queries = {
                "success_rate": f'rate(kserve_request_total{{service_name="{service_name}",code!~"5.."}}[5m]) / rate(kserve_request_total{{service_name="{service_name}"}}[5m])',
                "avg_latency": f'histogram_quantile(0.50, rate(kserve_request_duration_seconds_bucket{{service_name="{service_name}"}}[5m])) * 1000',
                "error_rate": f'rate(kserve_request_total{{service_name="{service_name}",code=~"5.."}}[5m]) / rate(kserve_request_total{{service_name="{service_name}"}}[5m])',
                "request_count": f'rate(kserve_request_total{{service_name="{service_name}"}}[5m]) * 300'  # 5 minute rate * 300 seconds
            }

            metrics = {}
            async with aiohttp.ClientSession() as session:
                for metric_name, query in queries.items():
                    url = f"{self.prometheus_url}/api/v1/query"
                    params = {"query": query}

                    async with session.get(url, params=params) as response:
                        if response.status == 200:
                            data = await response.json()
                            result = data.get("data", {}).get("result", [])

                            if result:
                                value = float(result[0]["value"][1])
                                metrics[metric_name] = value
                            else:
                                metrics[metric_name] = 0.0
                        else:
                            self.logger.warning(f"Failed to query {metric_name}: {response.status}")
                            metrics[metric_name] = 0.0

            return DeploymentMetrics(
                success_rate=metrics.get("success_rate", 0.0),
                avg_latency_ms=metrics.get("avg_latency", 0.0),
                error_rate=metrics.get("error_rate", 0.0),
                request_count=int(metrics.get("request_count", 0)),
                timestamp=datetime.utcnow()
            )

        except Exception as e:
            self.logger.error(f"Failed to retrieve metrics for {service_name}: {str(e)}")
            return None

    def _evaluate_canary_performance(self,
```

```
                                  baseline_metrics: Optional[DeploymentMetrics],
                                  canary_metrics: Optional[DeploymentMetrics],
                                  final_evaluation: bool = False) -> Dict[str, any]:
        """
        Evaluate canary performance against baseline and thresholds
        """
        if not canary_metrics:
            return {
                "action": "rollback",
                "reason": "No canary metrics available",
                "confidence": 1.0
            }

        # Check absolute thresholds
        threshold_checks = {
            "success_rate": canary_metrics.success_rate >= self.config.success_rate_threshold,
            "latency": canary_metrics.avg_latency_ms <= self.config.latency_threshold_ms,
            "error_rate": canary_metrics.error_rate <= self.config.error_rate_threshold,
            "min_requests": canary_metrics.request_count >= 10  # Minimum sample size
        }

        failed_checks = [check for check, passed in threshold_checks.items() if not passed]

        if failed_checks:
            return {
                "action": "rollback",
                "reason": f"Failed threshold checks: {failed_checks}",
                "canary_metrics": canary_metrics.__dict__,
                "failed_checks": failed_checks,
                "confidence": 1.0
            }

        # Compare with baseline if available
        if baseline_metrics and baseline_metrics.request_count >= 10:
            relative_checks = {
                "success_rate_degradation": (canary_metrics.success_rate / baseline_metrics.success_rate) >= 0.99,
                "latency_increase": (canary_metrics.avg_latency_ms / baseline_metrics.avg_latency_ms) <= 1.2,
                "error_rate_increase": canary_metrics.error_rate <= (baseline_metrics.error_rate * 2.0 + 0.001)
            }

            failed_relative_checks = [check for check, passed in relative_checks.items() if not passed]

            if failed_relative_checks:
                # Calculate confidence based on sample size and magnitude of degradation
                confidence = min(1.0, canary_metrics.request_count / 100)

                return {
                    "action": "rollback" if confidence > 0.7 else "hold",
                    "reason": f"Performance degradation detected: {failed_relative_checks}",
                    "canary_metrics": canary_metrics.__dict__,
```

```
                    "baseline_metrics": baseline_metrics.__dict__,
                    "failed_checks": failed_relative_checks,
                    "confidence": confidence
                }

        # Determine action based on evaluation type
        if final_evaluation:
            return {
                "action": "promote",
                "reason": "All performance checks passed",
                "canary_metrics": canary_metrics.__dict__,
                "confidence": 1.0
            }
        else:
            return {
                "action": "continue",
                "reason": "Performance within acceptable range",
                "canary_metrics": canary_metrics.__dict__,
                "confidence": 0.8
            }

    async def _update_traffic_split(self, canary_service_name: str, traffic_percent: int):
        """Update traffic split for canary deployment"""
        await self.kserve_integration.update_traffic_split(
            canary_service_name,
            {"canary": traffic_percent}
        )

    async def _rollback_canary(self, service_name: str, canary_service_name: str):
        """Rollback canary deployment"""
        try:
            # Set canary traffic to 0
            await self._update_traffic_split(canary_service_name, 0)

            # Delete canary service after grace period
            await asyncio.sleep(30)

            self.kserve_integration.custom_client.delete_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.kserve_integration.namespace,
                plural="inferenceservices",
                name=canary_service_name
            )

            self.logger.info(f"Rolled back canary deployment: {canary_service_name}")

        except Exception as e:
            self.logger.error(f"Error during rollback: {str(e)}")

    async def _promote_canary(self, service_name: str, canary_service_name: str, new_version: str):
        """Promote canary to production"""
        try:
            # Get canary service configuration
            canary_service = self.kserve_integration.custom_client.get_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
```

```
                namespace=self.kserve_integration.namespace,
                plural="inferenceservices",
                name=canary_service_name
            )

            # Update production service with canary configuration
            patch_body = {
                "spec": canary_service["spec"],
                "metadata": {
                    "labels": canary_service["metadata"]["labels"],
                    "annotations": {
                        **canary_service["metadata"]["annotations"],
                        "promotion.timestamp": datetime.utcnow().isoformat(),
                        "promoted.from": canary_service_name
                    }
                }
            }

            # Remove canary-specific configurations
            if "canaryTrafficPercent" in patch_body["spec"]["predictor"]:
                del patch_body["spec"]["predictor"]["canaryTrafficPercent"]

            # Update production service
            self.kserve_integration.custom_client.patch_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.kserve_integration.namespace,
                plural="inferenceservices",
                name=service_name,
                body=patch_body
            )

            # Clean up canary service
            await asyncio.sleep(60)  # Wait for traffic to shift
            await self._cleanup_failed_canary(canary_service_name)

            self.logger.info(f"Successfully promoted {canary_service_name} to production")

        except Exception as e:
            self.logger.error(f"Error during promotion: {str(e)}")
            raise

    async def _cleanup_failed_canary(self, canary_service_name: str):
        """Clean up failed canary deployment"""
        try:
            self.kserve_integration.custom_client.delete_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.kserve_integration.namespace,
                plural="inferenceservices",
                name=canary_service_name
            )
            self.logger.info(f"Cleaned up canary service: {canary_service_name}")
\subsubsection{Blue-Green Deployments}

Implement zero-downtime deployments with instant rollback capabilities:

\begin{lstlisting}[language=python, caption=Blue-Green Deployment Implementation]
```

```python
class BlueGreenDeploymentManager:
    """
    Manages blue-green deployments for zero-downtime model updates
    """

    def __init__(self, kserve_integration: MLflowKServeIntegration):
        self.kserve_integration = kserve_integration
        self.logger = logging.getLogger(__name__)

    async def deploy_blue_green(self,
                              model_name: str,
                              new_version: str,
                              service_name: str,
                              validation_tests: List[callable] = None) -> Dict[str, Any]:
        """
        Execute blue-green deployment with automated validation
        """

        deployment_result = {
            "model_name": model_name,
            "new_version": new_version,
            "service_name": service_name,
            "start_time": datetime.utcnow(),
            "status": "in_progress"
        }

        try:
            # Step 1: Deploy green environment
            green_service_name = f"{service_name}-green"

            self.logger.info(f"Deploying green environment: {green_service_name}")

            green_deployment = self.kserve_integration.create_inference_service(
                model_name=model_name,
                model_version=new_version,
                service_name=green_service_name
            )

            # Step 2: Wait for green environment to be ready
            await self._wait_for_service_ready(green_service_name)

            # Step 3: Run validation tests against green environment
            if validation_tests:
                validation_results = await self._run_validation_tests(
                    green_service_name,
                    validation_tests
                )

                if not validation_results["all_passed"]:
                    # Cleanup and abort
                    await self._cleanup_service(green_service_name)
                    deployment_result["status"] = "validation_failed"
                    deployment_result["validation_results"] = validation_results
                    return deployment_result

            # Step 4: Switch traffic from blue to green
            self.logger.info("Switching traffic from blue to green")

            # Get current blue service configuration
```

```
            blue_service = self.kserve_integration.custom_client.get_namespaced_custom_object(
                group="serving.kserve.io",
                version="v1beta1",
                namespace=self.kserve_integration.namespace,
                plural="inferenceservices",
                name=service_name
            )

            # Backup blue configuration
            blue_backup_name = f"{service_name}-blue-backup-{int(datetime.utcnow().timestamp())}"
            deployment_result["blue_backup_name"] = blue_backup_name

            # Rename current service to backup
            await self._rename_service(service_name, blue_backup_name)

            # Rename green service to production
            await self._rename_service(green_service_name, service_name)

            # Step 5: Monitor post-deployment metrics
            post_deployment_metrics = await self._monitor_post_deployment(service_name)
            deployment_result["post_deployment_metrics"] = post_deployment_metrics

            # Step 6: Cleanup old blue environment after successful deployment
            await asyncio.sleep(300)  # Wait 5 minutes before cleanup
            await self._cleanup_service(blue_backup_name)

            deployment_result["status"] = "completed"
            deployment_result["end_time"] = datetime.utcnow()

            self.logger.info(f"Blue-green deployment completed successfully for {model_name}")

            return deployment_result

        except Exception as e:
            self.logger.error(f"Blue-green deployment failed: {str(e)}")
            deployment_result["status"] = "failed"
            deployment_result["error"] = str(e)

            # Attempt rollback if switch was attempted
            if "blue_backup_name" in deployment_result:
                try:
                    await self.rollback_blue_green(service_name, deployment_result["blue_backup_name"])
                    deployment_result["rollback_attempted"] = True
                except:
                    deployment_result["rollback_failed"] = True

            raise

    async def rollback_blue_green(self, service_name: str, blue_backup_name: str):
        """
        Rollback blue-green deployment to previous version
        """
        try:
            self.logger.info(f"Rolling back blue-green deployment for {service_name}")

```

```python
            # Delete current green service
            await self._cleanup_service(service_name)

            # Restore blue service
            await self._rename_service(blue_backup_name, service_name)

            self.logger.info("Blue-green rollback completed successfully")

        except Exception as e:
            self.logger.error(f"Blue-green rollback failed: {str(e)}")
            raise

    async def _wait_for_service_ready(self, service_name: str, timeout_seconds: int = 300):
        """Wait for service to be ready"""
        start_time = datetime.utcnow()

        while (datetime.utcnow() - start_time).total_seconds() < timeout_seconds:
            try:
                service = self.kserve_integration.custom_client.get_namespaced_custom_object(
                    group="serving.kserve.io",
                    version="v1beta1",
                    namespace=self.kserve_integration.namespace,
                    plural="inferenceservices",
                    name=service_name
                )

                # Check if service is ready
                conditions = service.get("status", {}).get("conditions", [])
                ready_condition = next((c for c in conditions if c["type"] == "Ready"), None)

                if ready_condition and ready_condition["status"] == "True":
                    self.logger.info(f"Service {service_name} is ready")
                    return

            except Exception as e:
                self.logger.debug(f"Waiting for service readiness: {str(e)}")

            await asyncio.sleep(10)

        raise TimeoutError(f"Service {service_name} did not become ready within {timeout_seconds} seconds")

    async def _run_validation_tests(self, service_name: str, validation_tests: List[callable]) -> Dict[str, Any]:
        """Run validation tests against deployed service"""
        results = {
            "all_passed": True,
            "test_results": [],
            "execution_time": datetime.utcnow()
        }

        for test_func in validation_tests:
            try:
                test_result = await test_func(service_name)
                results["test_results"].append({
                    "test_name": test_func.__name__,
                    "status": "passed" if test_result else "failed",
                    "result": test_result
                })
```

```

                if not test_result:
                    results["all_passed"] = False

            except Exception as e:
                results["test_results"].append({
                    "test_name": test_func.__name__,
                    "status": "error",
                    "error": str(e)
                })
                results["all_passed"] = False

        return results

\subsubsection{A/B Testing Framework}

Implement statistical A/B testing for model variants:

\begin{lstlisting}[language=python, caption=A/B Testing Implementation]
import scipy.stats as stats
from typing import Tuple
import pandas as pd

class ABTestingFramework:
    """
    Statistical A/B testing framework for model variants
    """

    def __init__(self,
                 kserve_integration: MLflowKServeIntegration,
                 prometheus_url: str,
                 significance_level: float = 0.05,
                 minimum_sample_size: int = 1000,
                 test_duration_hours: int = 72):

        self.kserve_integration = kserve_integration
        self.prometheus_url = prometheus_url
        self.significance_level = significance_level
        self.minimum_sample_size = minimum_sample_size
        self.test_duration_hours = test_duration_hours
        self.logger = logging.getLogger(__name__)

    async def setup_ab_test(self,
                            control_model: Dict[str, str],
                            variant_model: Dict[str, str],
                            traffic_split: int = 50,
                            success_metric: str = "conversion_rate") -> Dict[str,
     Any]:
        """
        Setup A/B test between two model variants
        """

        test_config = {
            "test_id": f"ab_test_{int(datetime.utcnow().timestamp())}",
            "control_model": control_model,
            "variant_model": variant_model,
            "traffic_split": traffic_split,
            "success_metric": success_metric,
            "start_time": datetime.utcnow(),
            "status": "active"
        }

        try:
```

```
            # Deploy control version (if not already deployed)
            control_service_name = f"{control_model['name']}-control-{test_config['test_id']}"

            if not await self._service_exists(control_service_name):
                await self.kserve_integration.create_inference_service(
                    model_name=control_model['name'],
                    model_version=control_model['version'],
                    service_name=control_service_name
                )

            # Deploy variant version
            variant_service_name = f"{variant_model['name']}-variant-{test_config['test_id']}"
            await self.kserve_integration.create_inference_service(
                model_name=variant_model['name'],
                model_version=variant_model['version'],
                service_name=variant_service_name
            )

            # Configure traffic splitting
            await self._configure_ab_traffic(
                control_service_name,
                variant_service_name,
                traffic_split
            )

            test_config["control_service"] = control_service_name
            test_config["variant_service"] = variant_service_name

            self.logger.info(f"A/B test setup completed: {test_config['test_id']}")

            return test_config

        except Exception as e:
            self.logger.error(f"Failed to setup A/B test: {str(e)}")
            test_config["status"] = "failed"
            test_config["error"] = str(e)
            raise

    async def analyze_ab_test(self, test_config: Dict[str, Any]) -> Dict[str, Any]:
        """
        Analyze A/B test results and make statistical conclusions
        """

        try:
            # Collect metrics for both variants
            control_metrics = await self._collect_ab_metrics(
                test_config["control_service"],
                test_config["start_time"]
            )

            variant_metrics = await self._collect_ab_metrics(
                test_config["variant_service"],
                test_config["start_time"]
            )

            # Perform statistical analysis
            analysis_result = self._perform_statistical_analysis(
                control_metrics,
                variant_metrics,
```

```
                test_config["success_metric"]
            )

            # Generate recommendations
            recommendation = self._generate_recommendation(analysis_result, test_config)

            result = {
                "test_id": test_config["test_id"],
                "analysis_timestamp": datetime.utcnow(),
                "control_metrics": control_metrics,
                "variant_metrics": variant_metrics,
                "statistical_analysis": analysis_result,
                "recommendation": recommendation,
                "test_duration_hours": (datetime.utcnow() - test_config["start_time"]).total_seconds() / 3600
            }

            return result

        except Exception as e:
            self.logger.error(f"Failed to analyze A/B test: {str(e)}")
            raise

    def _perform_statistical_analysis(self,
                                    control_metrics: Dict,
                                    variant_metrics: Dict,
                                    success_metric: str) -> Dict[str, Any]:
        """
        Perform statistical significance testing
        """

        # Extract success counts and total counts
        control_successes = control_metrics.get(f"{success_metric}_count", 0)
        control_total = control_metrics.get("total_requests", 0)
        variant_successes = variant_metrics.get(f"{success_metric}_count", 0)
        variant_total = variant_metrics.get("total_requests", 0)

        if control_total == 0 or variant_total == 0:
            return {
                "test_type": "insufficient_data",
                "statistical_significance": False,
                "p_value": None,
                "confidence_interval": None,
                "effect_size": None
            }

        # Calculate conversion rates
        control_rate = control_successes / control_total
        variant_rate = variant_successes / variant_total

        # Perform two-proportion z-test
        count = np.array([control_successes, variant_successes])
        nobs = np.array([control_total, variant_total])

        # Calculate z-statistic and p-value
        z_stat, p_value = stats.proportions_ztest(count, nobs)

        # Calculate confidence interval for difference
        pooled_rate = (control_successes + variant_successes) / (control_total + variant_total)
        se_diff = np.sqrt(pooled_rate * (1 - pooled_rate) * (1/control_total + 1/variant_total))
```

```
        rate_diff = variant_rate - control_rate

        margin_of_error = stats.norm.ppf(1 - self.significance_level/2) *
    se_diff
        ci_lower = rate_diff - margin_of_error
        ci_upper = rate_diff + margin_of_error

        # Calculate effect size (Cohen's h)
        effect_size = 2 * (np.arcsin(np.sqrt(variant_rate)) - np.arcsin(np.sqrt
    (control_rate)))

        return {
            "test_type": "two_proportion_z_test",
            "control_rate": control_rate,
            "variant_rate": variant_rate,
            "rate_difference": rate_diff,
            "relative_improvement": (rate_diff / control_rate) * 100 if
    control_rate > 0 else 0,
            "z_statistic": z_stat,
            "p_value": p_value,
            "statistical_significance": p_value < self.significance_level,
            "confidence_interval": (ci_lower, ci_upper),
            "effect_size": effect_size,
            "sample_sizes": {"control": control_total, "variant": variant_total
    }
        }

    def _generate_recommendation(self,
                                 analysis_result: Dict[str, Any],
                                 test_config: Dict[str, Any]) -> Dict[str, Any]:
        """
        Generate actionable recommendations based on test results
        """

        if analysis_result["test_type"] == "insufficient_data":
            return {
                "action": "continue_test",
                "reason": "Insufficient data for statistical analysis",
                "required_sample_size": self.minimum_sample_size
            }

        # Check minimum sample size requirement
        min_sample_met = all(
            size >= self.minimum_sample_size
            for size in analysis_result["sample_sizes"].values()
        )

        if not min_sample_met:
            return {
                "action": "continue_test",
                "reason": "Minimum sample size not reached",
                "current_samples": analysis_result["sample_sizes"],
                "required_sample_size": self.minimum_sample_size
            }

        # Check test duration
        test_duration = (datetime.utcnow() - test_config["start_time"]).
    total_seconds() / 3600
        if test_duration < self.test_duration_hours:
            return {
                "action": "continue_test",
                "reason": f"Test duration ({test_duration:.1f}h) below minimum
    ({self.test_duration_hours}h)",
```

```python
                "current_duration_hours": test_duration,
                "required_duration_hours": self.test_duration_hours
            }

        # Make recommendation based on statistical results
        if analysis_result["statistical_significance"]:
            if analysis_result["rate_difference"] > 0:
                return {
                    "action": "deploy_variant",
                    "reason": "Variant shows statistically significant improvement",
                    "improvement": f"{analysis_result['relative_improvement']:.2f}%",
                    "confidence": f"{(1 - self.significance_level) * 100:.0f}%"
                }
            else:
                return {
                    "action": "keep_control",
                    "reason": "Control performs significantly better than variant",
                    "degradation": f"{analysis_result['relative_improvement']:.2f}%",
                    "confidence": f"{(1 - self.significance_level) * 100:.0f}%"
                }
        else:
            return {
                "action": "no_significant_difference",
                "reason": "No statistically significant difference detected",
                "p_value": analysis_result["p_value"],
                "recommendation": "Consider other factors like cost, complexity, or business requirements"
            }

# Example usage for A/B testing
async def example_ab_test():
    """Example A/B test implementation"""

    # Initialize framework
    ab_framework = ABTestingFramework(
        kserve_integration=integration,
        prometheus_url="http://prometheus.monitoring.svc.cluster.local:9090"
    )

    # Setup A/B test
    test_config = await ab_framework.setup_ab_test(
        control_model={"name": "recommendation-model", "version": "1.2.0"},
        variant_model={"name": "recommendation-model", "version": "1.3.0"},
        traffic_split=50,
        success_metric="click_through_rate"
    )

    # Monitor test for specified duration
    while True:
        analysis = await ab_framework.analyze_ab_test(test_config)

        if analysis["recommendation"]["action"] != "continue_test":
            print(f"Test completed with recommendation: {analysis['recommendation']['action']}")
            break

        print(f"Test continuing... Current improvement: {analysis['statistical_analysis'].get('relative_improvement', 0):.2f}%")
```

```
        await asyncio.sleep(3600)  # Check every hour
```

Listing 13: Automated Canary Deployment Controller

This comprehensive model deployment and serving section provides production-ready implementations for advanced deployment patterns including canary deployments, blue-green deployments, and statistical A/B testing frameworks, all integrated with KServe and MLflow for complete MLOps workflow management.

4 Monitoring and Observability

4.1 Model Performance Monitoring

Implementing comprehensive monitoring is crucial for maintaining model performance in production:

```python
import prometheus_client
from prometheus_client import Counter, Histogram, Gauge
import numpy as np
from scipy import stats

class ModelMonitor:
    def __init__(self):
        # Prometheus metrics
        self.prediction_counter = Counter(
            'model_predictions_total',
            'Total number of predictions made',
            ['model_name', 'version']
        )

        self.prediction_latency = Histogram(
            'model_prediction_duration_seconds',
            'Model prediction latency',
            ['model_name', 'version']
        )

        self.model_accuracy = Gauge(
            'model_accuracy_score',
            'Current model accuracy',
            ['model_name', 'version']
        )

        self.drift_score = Gauge(
            'model_drift_score',
            'Data drift detection score',
            ['model_name', 'feature']
        )

    def log_prediction(self, model_name, version, latency):
        """Log prediction metrics"""
        self.prediction_counter.labels(
            model_name=model_name,
            version=version
        ).inc()

        self.prediction_latency.labels(
            model_name=model_name,
            version=version
        ).observe(latency)

    def detect_data_drift(self, reference_data, current_data, feature_name):
        """Detect data drift using Kolmogorov-Smirnov test"""
```

```
        try:
            # Perform KS test
            ks_statistic, p_value = stats.ks_2samp(reference_data, current_data
)

            # Update drift metric
            self.drift_score.labels(
                model_name="production_model",
                feature=feature_name
            ).set(ks_statistic)

            # Alert if significant drift detected
            if p_value < 0.05:
                self.send_drift_alert(feature_name, ks_statistic, p_value)

            return ks_statistic, p_value

        except Exception as e:
            print(f"Error detecting drift for {feature_name}: {str(e)}")
            return None, None

    def send_drift_alert(self, feature_name, ks_statistic, p_value):
        """Send alert when data drift is detected"""
        alert_message = f"""
        Data Drift Alert!
        Feature: {feature_name}
        KS Statistic: {ks_statistic:.4f}
        P-value: {p_value:.4f}
        Recommendation: Review model performance and consider retraining
        """
        # Integration with alerting system (Slack, PagerDuty, etc.)
        print(alert_message)
```

Listing 14: Model Monitoring Implementation

5 Best Practices and Recommendations

5.1 Pipeline Design Principles

1. **Modularity:** Design pipeline components as independent, reusable modules that can be easily tested and maintained.

2. **Reproducibility:** Ensure all experiments and deployments are fully reproducible through proper versioning of code, data, and dependencies.

3. **Scalability:** Design pipelines to handle varying workloads and data volumes without manual intervention.

4. **Monitoring:** Implement comprehensive monitoring at every stage of the pipeline to quickly identify and resolve issues.

5. **Security:** Apply security best practices including proper authentication, authorization, and data encryption.

5.2 Performance Optimization

- Use GPU acceleration for training intensive models

- Implement efficient data loading and preprocessing
- Optimize model serving with batching and caching strategies
- Monitor resource utilization and scale components as needed
- Implement model quantization and pruning for inference optimization

6 Troubleshooting Common Issues

6.1 Pipeline Failures

Common issues and their solutions:

Issue	Symptoms	Solution
Resource limitations	Pipeline steps timing out or failing	Increase resource requests/limits
Data access issues	Permission denied errors	Check RBAC and storage permissions
Model convergence	Poor model performance	Adjust hyperparameters and data quality
Deployment failures	Service unavailable errors	Verify KServe configuration

Table 1: Common Pipeline Issues and Solutions

7 Conclusion

This comprehensive guide demonstrates how to build production-ready MLOps pipelines using Kubeflow and MLflow. The combination of these technologies provides a robust foundation for managing the complete machine learning lifecycle, from experimentation to production deployment.

Key benefits of this approach include:

- **Standardization:** Consistent workflows across teams and projects
- **Scalability:** Kubernetes-native scaling capabilities
- **Reproducibility:** Complete experiment and deployment tracking
- **Monitoring:** Comprehensive observability and alerting
- **Collaboration:** Shared infrastructure and knowledge base

As MLOps practices continue to evolve, this foundation provides the flexibility to adapt and integrate new tools and methodologies while maintaining operational excellence.

8 References

1. Kubeflow Documentation. `https://www.kubeflow.org/docs/`
2. MLflow Documentation. `https://mlflow.org/docs/latest/index.html`
3. KServe Documentation. `https://kserve.github.io/website/`

4. Kubernetes Documentation. `https://kubernetes.io/docs/`

5. Sculley, D., et al. "Hidden Technical Debt in Machine Learning Systems." NIPS 2015.

9 About the Author

Vladimir Ovcharov is an MLOps Engineer and ML Systems Architect with 8+ years of experience in building production-ready machine learning infrastructure. He specializes in designing scalable ML pipelines, implementing robust monitoring systems, and optimizing model deployment workflows.

Contact: vladimir.ovcharov@example.com