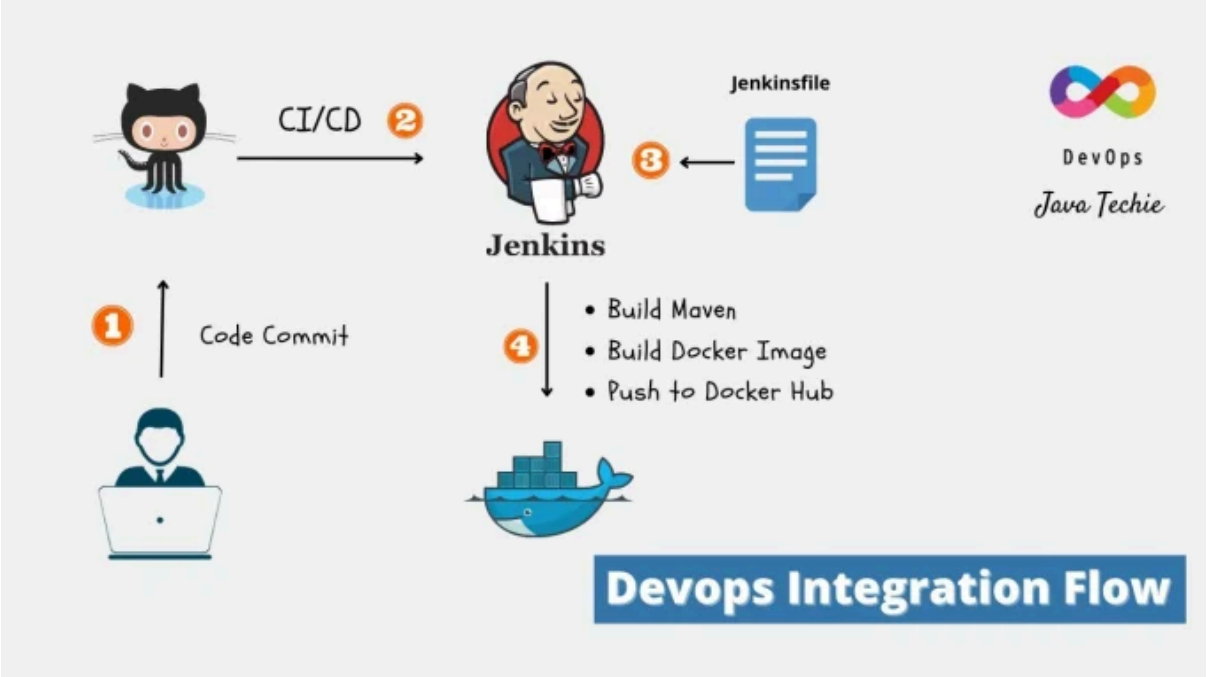
📚 TP 3 - Jobs et Déploiements avec Jenkins et Docker



Objectifs pédagogiques

- Maîtriser Jenkins pour l'orchestration ML
- Containeriser les applications ML
- Gérer les dépendances entre tâches complexes

Contexte

Déploiement d'un modèle de vision par ordinateur avec scalabilité automatique et monitoring.

🛠️ Mise en pratique

Étape 1 : Configuration Jenkins avec pipelines ML

```
// Jenkinsfile
pipeline {
    agent any

    environment {
        DOCKER_REGISTRY = 'your-registry.com'
        MODEL_NAME = 'vision-classifier'
        KUBECONFIG = credentials('k8s-config')
    }

    stages {
        stage('Data Processing') {
            parallel {
                stage('Data Validation') {
                    steps {
                        script {
                            sh '''
                                python scripts/validate_input_data.py
                                python scripts/check_data_quality.py
                            '''
                        }
                    }
                }
                stage('Feature Engineering') {
                    steps {
                        sh 'python src/feature_engineering.py'
                    }
                }
            }
        }

        stage('Model Training') {
            when {
                expression {
```

```
                return params.RETRAIN_MODEL == true ||
                       currentBuild.previousBuild?.result != 'SUCCESS'
            }
        }
        steps {
            script {
                def trainingJob = build job: 'model-training-job',
                    parameters: [
                        string(name: 'DATA_VERSION', value: env.DATA_VERSION),
                        string(name: 'MODEL_CONFIG', value: 'production')
                    ]

                env.MODEL_VERSION = trainingJob.getBuildVariables().MODEL_VERSION
            }
        }
    }

    stage('Model Evaluation') {
        steps {
            script {
                sh '''
                    python src/evaluate_model.py \
                        --model-version ${MODEL_VERSION} \
                        --threshold 0.9
                '''
                def evaluation = readJSON file: 'evaluation_results.json'

                if (evaluation.accuracy < 0.9) {
                    error("Model accuracy below threshold: ${evaluation.accuracy}")
                }
            }
        }
    }

    stage('Docker Build') {
        steps {
            script {
                def image =
docker.build("${DOCKER_REGISTRY}/${MODEL_NAME}:${MODEL_VERSION}")
                docker.withRegistry("https://${DOCKER_REGISTRY}", 'registry-credentials') {
                    image.push()
                    image.push('latest')
                }
            }
        }
    }

    stage('Deploy to Staging') {
```

```
        steps {
            sh '''
                helm upgrade --install ${MODEL_NAME}-staging helm/ml-service \
                    --set image.tag=${MODEL_VERSION} \
                    --set environment=staging \
                    --namespace ml-staging
            '''
        }
    }

    stage('Integration Tests') {
        steps {
            sh '''
                python tests/integration_tests.py \
                    --endpoint http://ml-staging.internal/predict
            '''
        }
    }

    stage('Deploy to Production') {
        when {
            branch 'main'
        }
        input {
            message "Deploy to production?"
            ok "Deploy"
        }
        steps {
            sh '''
                helm upgrade --install ${MODEL_NAME} helm/ml-service \
                    --set image.tag=${MODEL_VERSION} \
                    --set environment=production \
                    --set replicas=3 \
                    --namespace ml-production
            '''
        }
    }
}

post {
    always {
        archiveArtifacts artifacts: 'logs/**', allowEmptyArchive: true
        publishHTML([
            allowMissing: false,
            alwaysLinkToLastBuild: true,
            keepAll: true,
            reportDir: 'reports',
            reportFiles: 'model_report.html',
```

```
                reportName: 'Model Report'
            ])
        }
    }
}
```

Étape 2 : Containerisation avec optimisation ML

Dockerfile.ml-service

```
FROM python:3.9-slim

# Installation des dépendances système pour ML
RUN apt-get update && apt-get install -y \
    libgomp1 \
    libgl1-mesa-glx \
    libglib2.0-0 \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Installation des dépendances Python
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copie du code source
COPY src/ ./src/
COPY models/ ./models/
COPY config/ ./config/

# Configuration pour l'optimisation ML
ENV PYTHONUNBUFFERED=1
ENV OMP_NUM_THREADS=1
ENV MKL_NUM_THREADS=1

# Healthcheck pour Kubernetes
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD python src/health_check.py

EXPOSE 8080

CMD ["python", "src/api_server.py"]
```

Étape 3 : Service API optimisé

src/api_server.py

```
from flask import Flask, request, jsonify
```

```
import joblib
import numpy as np
from prometheus_client import Counter, Histogram, generate_latest
import logging
import time

app = Flask(__name__)

 Métriques Prometheus
PREDICTION_COUNT = Counter('ml_predictions_total', 'Total predictions made')
PREDICTION_DURATION = Histogram('ml_prediction_duration_seconds', 'Prediction
duration')
ERROR_COUNT = Counter('ml_errors_total', 'Total errors', ['error_type'])

class ModelServer:
    def __init__(self):
        self.model = None
        self.model_version = None
        self.load_model()

    def load_model(self):
        """Charge le modèle avec gestion d'erreur"""
        try:
            self.model = joblib.load('models/production_model.pkl')
            with open('models/model_version.txt', 'r') as f:
                self.model_version = f.read().strip()
            logging.info(f"Model {self.model_version} loaded successfully")
        except Exception as e:
            logging.error(f"Failed to load model: {e}")
            ERROR_COUNT.labels(error_type='model_loading').inc()

model_server = ModelServer()

@app.route('/predict', methods=['POST'])
def predict():
    start_time = time.time()

    try:
        data = request.get_json()

         Validation des données d'entrée
        if not data or 'features' not in data:
            ERROR_COUNT.labels(error_type='invalid_input').inc()
            return jsonify({'error': 'Invalid input format'}), 400

         Préparation des features
        features = np.array(data['features']).reshape(1, -1)
```

```
        Prédiction
        prediction = model_server.model.predict(features)[0]
        probability = model_server.model.predict_proba(features)[0].max()

        Métriques
        PREDICTION_COUNT.inc()
        PREDICTION_DURATION.observe(time.time() - start_time)

        return jsonify({
            'prediction': int(prediction),
            'probability': float(probability),
            'model_version': model_server.model_version,
            'timestamp': time.time()
        })

    except Exception as e:
        ERROR_COUNT.labels(error_type='prediction_error').inc()
        logging.error(f"Prediction error: {e}")
        return jsonify({'error': 'Prediction failed'}), 500

@app.route('/metrics')
def metrics():
    return generate_latest()

@app.route('/health')
def health():
    if model_server.model is None:
        return jsonify({'status': 'unhealthy', 'reason': 'model_not_loaded'}), 503
    return jsonify({'status': 'healthy', 'model_version': model_server.model_version})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Livrables

1. Pipeline Jenkins complexe avec parallélisation
2. Images Docker optimisées pour ML
3. API de service avec métriques
4. Configuration Kubernetes/Helm

Critères d'évaluation

- Orchestration : Gestion des dépendances complexes
- Performance : Optimisation des containers ML
- Monitoring : Métriques et observabilité
