



Master Big Data & Internet of Things

Année Universitaire 2025-2026

Mini MLOps Report

CI/CD Automation and Monitoring – MLOpsFull

Réalisé par :

Amine Hamouchi
Ghoufrane Bendehmane

Professeur responsable :

Badr Hirchoua

Mai 2026

Table des matières

Abstract	1
1 Introduction And Objective	2
2 CI/CD Implementation With Jenkins	3
2.1 Pipeline Architecture	3
2.2 Stage Model	3
2.3 Event Policy	4
2.4 CI/CD Pipeline Schematic	4
2.5 Model Quality Gate	4
2.6 CD Additions: Smoke Test And Documentation	5
2.7 Technical Challenges And Resolutions	5
3 Monitoring Implementation	6
3.1 Monitoring Scope	6
3.2 Core Components And Scripts	6
3.3 Monitoring Logic In Serving	7
3.4 Alert And Act Flow	7
3.5 Monitoring Pipeline Schematic	7
3.6 Jenkinsfile Monitoring Stage	7
4 Prometheus And Grafana Observability Stack	9
4.1 Purpose And Architecture	9
4.2 Docker Compose Services	9
4.3 Prometheus Configuration	9
4.4 Grafana Provisioning	10
4.5 Screenshot Evidence	10
5 Integrated CI/CD + Monitoring Governance	13
6 Pipeline Overview Figure	14
7 Quality Gate Failure Case Study	15
8 Deliverables And Evidence	16
8.1 Produced Deliverables	16
8.2 Validation Checklist (Concise)	16
9 Conclusion	17

Abstract

This report presents the academic implementation of CI/CD and monitoring for the `MLOpsFull` machine learning project. The work moves beyond basic automation by combining software validation, model-quality governance, deployment smoke checks, documentation quality enforcement, post-deployment monitoring scripts, and a Prometheus/Grafana observability stack via Docker Compose.

The resulting lifecycle is: **validate before merge, gate model quality, deliver on main, monitor, alert, and act**, then **visualize service health in Grafana**. This aligns with course MLOps requirements and provides reproducible artifacts for assessment.

Chapitre 1

Introduction And Objective

The project objective is to operationalize an ML text-classification system with controlled release and continuous observability. The implementation focuses on:

1. Jenkins-based CI/CD automation using event-driven stages.
2. Model-aware quality gates before deployment.
3. Monitoring layers for system health, data quality, drift, and performance.
4. Actionable decision outputs (retrain vs monitor-only) after alerts.
5. Prometheus and Grafana dashboards for operational visualization.

Main Principle

CI/CD is treated as a full MLOps control loop: code and model checks before release, followed by post-release monitoring and governance.

Chapitre 2

CI/CD Implementation With Jenkins

2.1 Pipeline Architecture

The pipeline is implemented in a declarative Jenkinsfile with:

- `agent any`,
- `timestamps()`,
- `disableConcurrentBuilds()`.
Build environment setup:
- create virtual environment,
- upgrade `pip`, `setuptools`, `wheel`,
- install dependencies from `requirements.txt`,
- use Windows-compatible activation (`.\venv\Scripts\Activate.ps1`).

2.2 Stage Model

Stage	Trigger	Purpose
Checkout	Always	Retrieve repository source code.
Setup Python	Always	Build clean environment and install dependencies.
CI Workloads	PR to <code>main</code>	Validate training/evaluation entry points.
MLOps Train + Evaluate Gate	PR + <code>main</code>	Run train/evaluate gate and enforce F1 threshold.
CD Serve + Docs	<code>main</code> push	Run deployment smoke tests and <code>mkdocs build --strict</code> .
Monitoring Rules + Act	<code>main</code> push	Execute expectations, drift/performance rules, alerts, and act decision.

Table 2.1: Event-driven Jenkins stage model used in the project.

2.3 Event Policy

The CI/CD behavior follows branch/event context:

- Pull Request targeting main: execute validation and quality gate.
 - Push on main (non-PR): execute delivery workflow and monitoring integration.
- Pull-request stages are enabled only when `CHANGE_ID` is present:

```
stage('CI Workloads (PR validation)') {  
  when {  
    expression { env.CHANGE_ID } // PR only  
  }  
  steps {  
    bat """  
    call %ENV_DIR%\Scripts\activate  
    python -m madewithml.train --help  
    python -m madewithml.evaluate --help  
    """  
  }  
}
```

This enforces a controlled promotion path:

```
code change -> PR validation -> model gate -> merge  
-> main delivery -> monitoring checks
```

2.4 CI/CD Pipeline Schematic

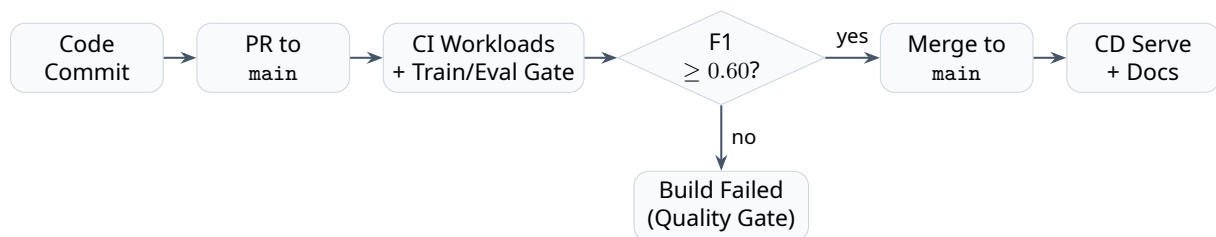


Figure 2.1: Schematized CI/CD flow used in Jenkins.

2.5 Model Quality Gate

The script `scripts/ci_train_eval_gate.py` writes:

- `artifacts/train_results.json`,
- `artifacts/eval_results.json`,
- `artifacts/quality_gate_summary.json`.

The Jenkins build fails when weighted F1 is below `F1_THRESHOLD` (default 0.30). This introduces model-aware release governance, not only software checks.

Quality-gate parameters are declared in the Jenkins environment block:

```
environment {  
  RESULTS_DIR = 'artifacts'  
  F1_THRESHOLD = '0.60'  
  CI_NUM_SAMPLES = '256'  
  CI_NUM_EPOCHS = '3'  
  CI_BATCH_SIZE = '16'  
}
```

The gate stage executes training and evaluation, then fails the build if F1 is below threshold:

```
stage('MLOps Train + Evaluate Gate (main)') {
    when {
        allOf {
            expression { (env.GIT_BRANCH ?: '').contains('origin/main') }
            expression { !env.CHANGE_ID }
        }
    }
    steps {
        bat """
        call %VENV_DIR%\Scripts\activate
        set PYTHONPATH=%CD%
        python scripts\ci_train_eval_gate.py ... --f1-threshold %F1_THRESHOLD%
        """
    }
}
```

2.6 CD Additions: Smoke Test And Documentation

On main pushes, the CD stage runs:

- `scripts/ci_deploy_smoke.py` to verify serving endpoints (`/`, `/run_id/`, `/metrics`, `/predict/`),
- `mkdocs build --strict` for documentation quality.

Recent documentation updates include `mkdocs.yml`, `docs/` pages, compatibility pin fixes for MkDocs/mkdocstrings tooling, and strict-mode docstring correction in `madewithml/predict.py`.

The CD stage in Jenkinsfile runs deployment smoke checks and strict documentation build:

```
stage('CD Serve + Docs (main push)') {
    when {
        allOf {
            expression { (env.GIT_BRANCH ?: '').contains('origin/main') }
            expression { !env.CHANGE_ID }
        }
    }
    steps {
        bat """
        call %VENV_DIR%\Scripts\activate
        set PYTHONPATH=%CD%
        python scripts\ci_deploy_smoke.py ...
        if errorlevel 1 exit /b 1
        mkdocs build --strict
        if errorlevel 1 exit /b 1
        """
    }
}
```

2.7 Technical Challenges And Resolutions

- **Shell mismatch:** initial use of Bash source failed on Windows; resolved via PowerShell activation.
- **Build backend error:** `setuptools.build_meta` issue resolved by upgrading packaging tools.
- **Interpreter compatibility:** pinned dependency stack showed instability on Python 3.12/3.11; Python 3.10 identified as stable target for CI agents.

Chapitre 3

Monitoring Implementation

3.1 Monitoring Scope

Monitoring implementation follows eight layers:

1. system health metrics,
2. expectations-based input validation,
3. notebook-aligned drift detection (KS + Chi-square),
4. sliding-window performance monitoring,
5. alert rule engine,
6. act workflow (decision automation),
7. Jenkins integration on `main`,
8. Prometheus scraping and Grafana dashboards (Docker Compose stack).

3.2 Core Components And Scripts

Centralized logging is configured in `madewithml/config.py`. Runtime outputs are written to:

- `logs/info.log`
- `logs/error.log`

Component	Function
<code>madewithml/serve.py</code>	Logs request latency/status, captures errors, and runs drift checks during inference.
<code>scripts/monitor_expectations.py</code>	Validates input schema/quality and writes expectations report artifact.
<code>scripts/monitor_sliding_metrics.py</code>	Computes cumulative and rolling F1 time series.
<code>scripts/monitor_alerts.py</code>	Applies threshold rules for drift and performance regression, outputs alert ticket.
<code>scripts/monitor_act.py</code>	Produces retrain vs monitor-only decision and optional retrain trigger file.

Table 3.1: Monitoring components integrated in the project.

Generated monitoring artifacts:

- artifacts/monitoring/expectations_report.json
- artifacts/monitoring/performance_timeseries.json
- artifacts/alerts/latest_alert.json
- artifacts/alerts/action_decision.json
- artifacts/alerts/retrain.trigger (when retraining is recommended)

3.3 Monitoring Logic In Serving

For /predict/ requests, drift is checked against reference distributions. The implementation evolved from simple text-length checks to notebook-aligned token-count based methods:

- KSDrift on numeric token-count features,
- ChiSquareDrift on token-size buckets (small, medium, large).

When drift is detected, structured warning events with p-values and distances are recorded in logs/error.log without blocking inference.

3.4 Alert And Act Flow

```
serve traffic
-> request metrics + drift signals
-> expectations and sliding-window metrics
-> alert rules (thresholds)
-> action decision (retrain / monitor only)
```

3.5 Monitoring Pipeline Schematic



Figure 3.1: Schematized monitoring, alerting, and act workflow.

3.6 Jenkinsfile Monitoring Stage

On main pushes, monitoring scripts are executed in sequence with fail-fast exit codes:

```
stage('Monitoring Rules + Act (main push)') {
  when {
    allOf {
      expression { (env.GIT_BRANCH ?: '').contains('origin/main') }
      expression { !env.CHANGE_ID }
    }
  }
}
```

```

    }
  }
  steps {
    bat """
    python scripts\monitor_expectations.py ...
    if not "!RC!"=="0" exit /b !RC!
    python scripts\monitor_sliding_metrics.py ...
    if not "!RC!"=="0" exit /b !RC!
    python scripts\monitor_alerts.py ...
    if not "!RC!"=="0" exit /b !RC!
    python scripts\monitor_act.py ...
    if not "!RC!"=="0" exit /b !RC!
    """
  }
}

```

Artifacts are archived in the post block:

```

post {
  always {
    archiveArtifacts artifacts: 'artifacts/*.json, artifacts/monitoring/*.json,
    artifacts/alerts/*.json, site/**', allowEmptyArchive: true
  }
}

```

Chapitre 4

Prometheus And Grafana Observability Stack

4.1 Purpose And Architecture

Beyond JSON monitoring artifacts and log files, the project exposes Prometheus-compatible metrics from the FastAPI service and visualizes them in Grafana. The stack is defined in `docker-compose.yml` and uses a shared Docker network (`mlops_net`).

- **mlopsfull-serve** (profile serve): model API on port 8000, exposes GET `/metrics`.
- **prometheus**: scrapes API metrics every 15 seconds on port 9090.
- **grafana**: reads Prometheus as default datasource and loads provisioned dashboards on port 3000.

4.2 Docker Compose Services

```
prometheus:
  image: prom/prometheus:v2.52.0
  ports: ["9090:9090"]
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro

grafana:
  image: grafana/grafana:10.4.2
  ports: ["3000:3000"]
  volumes:
    - ./monitoring/grafana/provisioning:/etc/grafana/provisioning:ro
    - ./monitoring/grafana/dashboards:/var/lib/grafana/dashboards:ro
  depends_on: [prometheus]
```

Local startup commands:

```
docker compose up -d prometheus grafana
docker compose --profile serve up -d mlopsfull-serve
```

Set `RUN_ID` to a trained MLflow run before starting the serve profile so predictions can generate live metrics.

4.3 Prometheus Configuration

Prometheus scrapes both itself and the ML API:

```
scrape_configs:
  - job_name: mlopsfull-api
    metrics_path: /metrics
    static_configs:
      - targets: [mlopsfull-serve:8000]
```

Exported metric families include request counters, latency histograms, prediction class counters, input-length histograms, validation-failure counters, and an other prediction-rate gauge (defined in `madewithml/monitoring_metrics.py`).

4.4 Grafana Provisioning

Grafana auto-loads:

- `datasource: monitoring/grafana/provisioning/datasources/prometheus.yml` (`http://prometheus:9090`)
- `dashboard provider: monitoring/grafana/provisioning/dashboards/dashboards.yml`,
- `dashboard JSON: monitoring/grafana/dashboards/mlopsfull-monitoring.json`.

Default dashboard panels include API request rate, latency p95, predictions by class, input text length, prediction confidence, and input expectation failures.

4.5 Screenshot Evidence

Figures 4.1–4.4 show runtime evidence from Jenkins, Prometheus, and Grafana after running the CI/CD and observability stack locally.

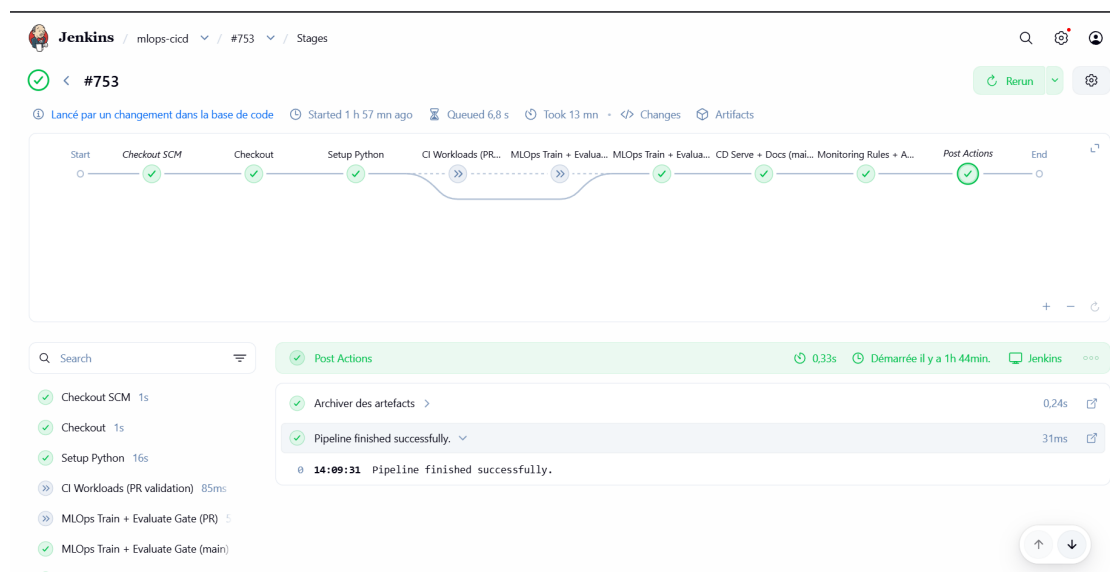


Figure 4.1: Jenkins pipeline execution on `main`: validation, quality gate, deployment smoke test, documentation build, and monitoring stages.

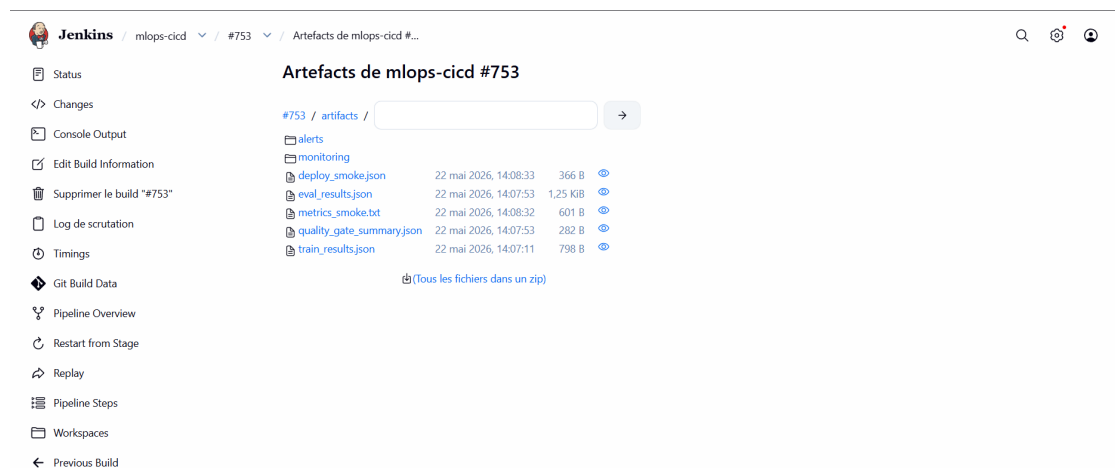


Figure 4.2: Archived Jenkins artifacts: quality-gate outputs, monitoring reports, and build traceability files retained after the run.

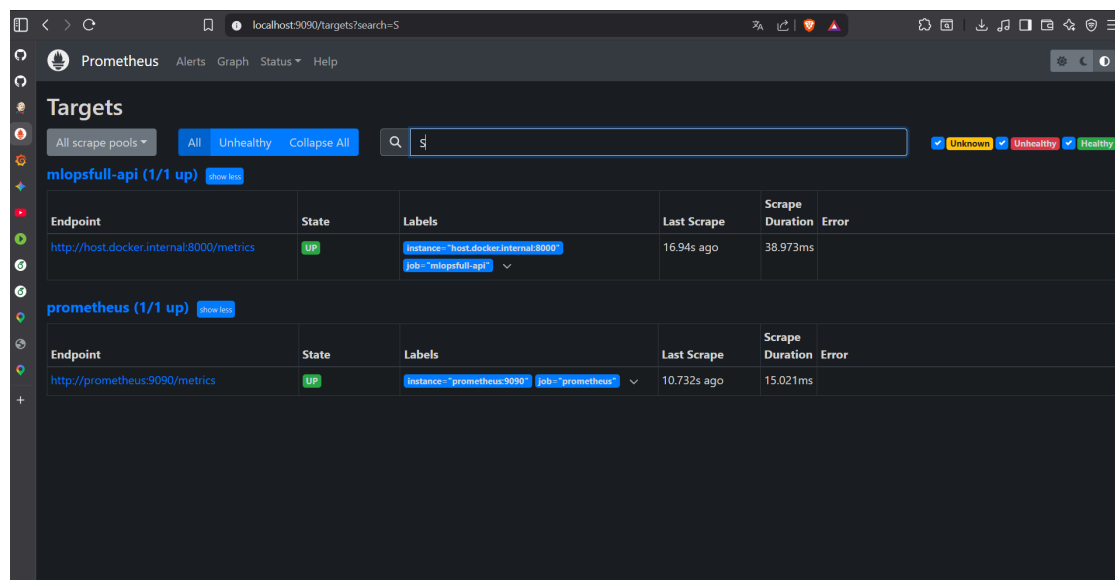


Figure 4.3: Prometheus targets page (localhost:9090/targets): the mlopsfull-api scrape job is UP when the Ray Serve metrics endpoint is reachable.

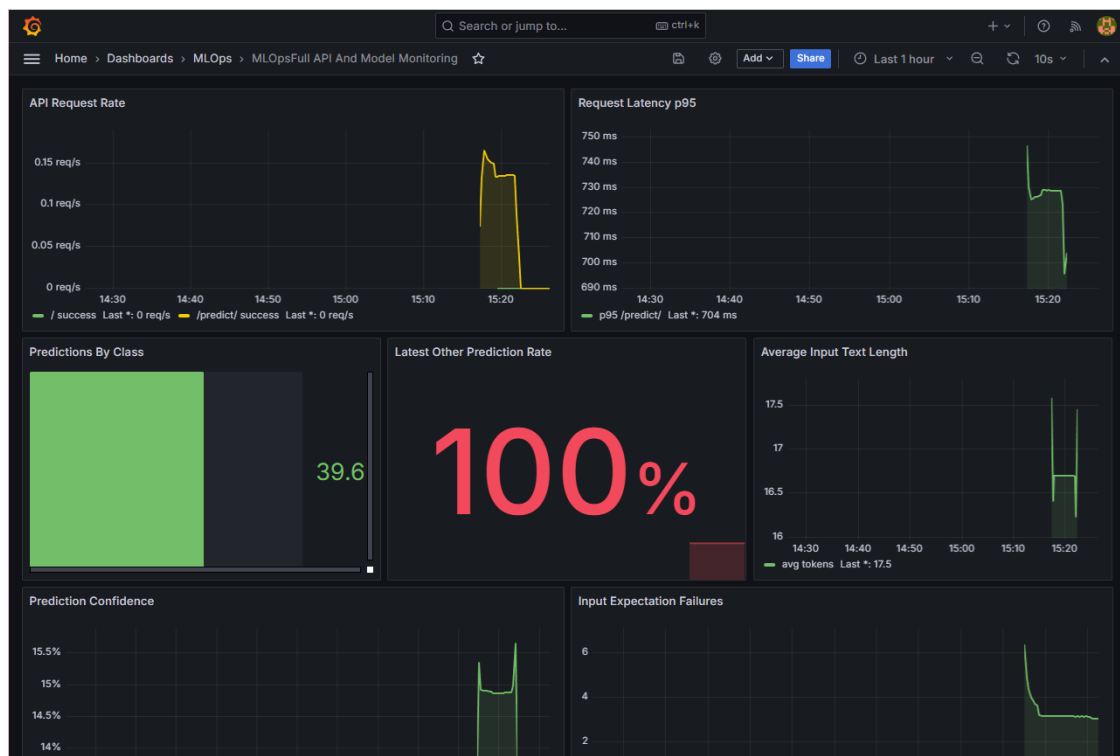


Figure 4.4: Grafana dashboard (localhost:3000): provisioned *MLOpsFull API And Model Monitoring* panels fed by the Prometheus datasource.

Chapitre 5

Integrated CI/CD + Monitoring Governance

On main pushes, Jenkins executes:

```
1) monitor_expectations.py  
2) monitor_sliding_metrics.py  
3) monitor_alerts.py  
4) monitor_act.py
```

Commands are fail-fast (`if errorlevel 1 exit /b 1`), and outputs are archived for traceability. This operationalizes monitoring as a release governance mechanism instead of a manual activity.



Figure 5.1: Integrated governance pipeline: CI/CD connected with monitoring checks.

Chapitre 6

Pipeline Overview Figure

After detailing the CI/CD and monitoring pipelines separately, Figure 6.1 provides a consolidated end-to-end view of the full Jenkins workflow.

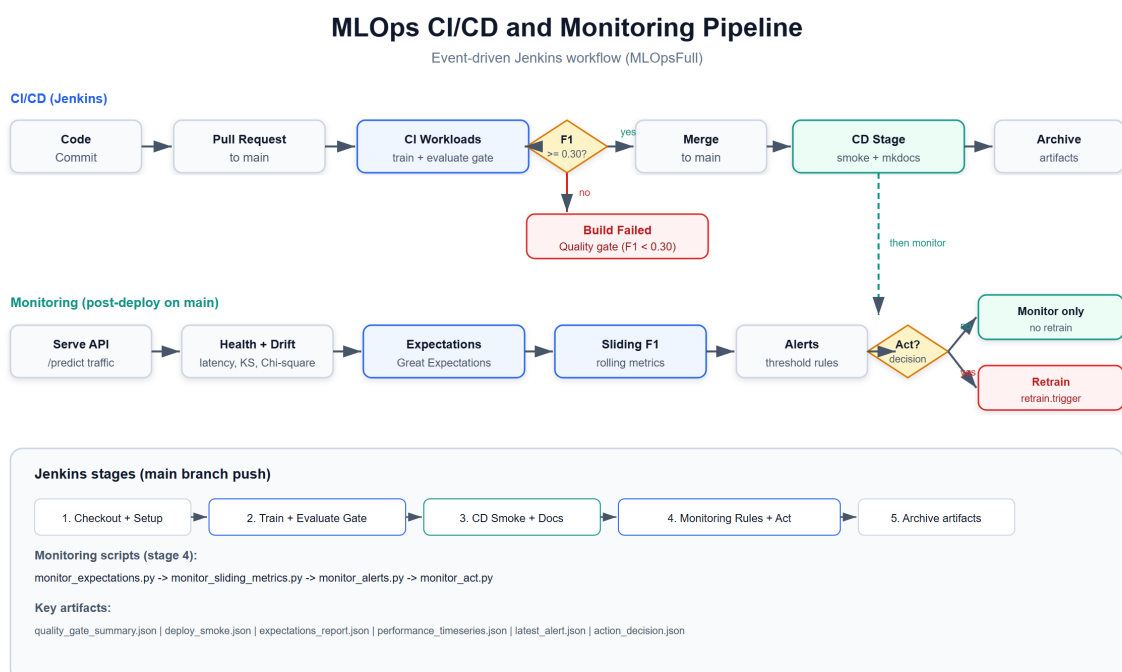


Figure 6.1: Complete schematized pipeline: CI/CD quality gate, CD delivery, monitoring layers, alert/act decisions, and Jenkins stage sequence.

Chapitre 7

Quality Gate Failure Case Study

An observed Jenkins run demonstrated correct gate behavior:

- training completed successfully,
- evaluation reported weighted $F1 = 0.20147$,
- threshold was 0.30,
- gate result was `passed=false`,
- pipeline stopped with non-zero exit code and skipped CD stage.

Academic interpretation: this is a **successful governance outcome**, because low-quality models are prevented from deployment.

Corrective action kept threshold strict and increased CI training effort:

- `CI_NUM_SAMPLES = 256`
- `CI_NUM_EPOCHS = 3`
- `CI_BATCH_SIZE = 16`

Chapitre 8

Deliverables And Evidence

8.1 Produced Deliverables

- Jenkins pipeline with CI/CD and monitoring stages.
- Quality-gate artifacts and deployment smoke artifacts.
- Monitoring artifacts (expectations, performance, alerts, action decision).
- MkDocs documentation sources and strict build output (`site/`).
- Prometheus/Grafana Docker stack with provisioned dashboard (`docker-compose.yml`, `monitoring/`).

8.2 Validation Checklist (Concise)

1. Run PR build and confirm validation + quality gate execution.
2. Run `main` build and confirm smoke test + strict docs build.
3. Confirm monitoring stage executes all four scripts in order.
4. Verify archived artifacts exist for quality gate and monitoring outputs.
5. Confirm non-zero exits block progression when gate/rules fail.
6. Start Prometheus and Grafana with Docker Compose; confirm targets are UP.
7. Open Grafana dashboard and verify live panels after sending `/predict/` traffic.
8. Confirm Jenkins, Prometheus, and Grafana screenshots (Figures 4.1–4.4) match the live stack.

Chapitre 9

Conclusion

The project now demonstrates an academic, end-to-end MLOps control loop centered on CI/CD and monitoring:

- **CI:** event-driven validation and model-quality gating,
- **CD:** controlled post-merge delivery with smoke and docs quality checks,
- **Monitoring:** system health + data/model monitoring + alert-to-action decisions,
- **Observability:** Prometheus metrics and Grafana dashboards for live operational visibility.

The implementation therefore satisfies key course expectations: reproducible automation, measurable release criteria, operational monitoring integrated into the deployment lifecycle, and visual observability through Prometheus/Grafana.