

Technical Design Document

ML Engineering Technical Challenge

Author: MIE. Brayan Cuevas Arteaga

Repository: https://github.com/BrayanCuevas/predictive-maintenance-mlop_walmart

Date: June 2025

Executive Summary

This document describes the architecture and technical decisions implemented in an MLOps project for predictive maintenance. My primary approach was to develop a complete MLOps stack within limited time, prioritizing the demonstration of technical capabilities and robust architecture over extreme model optimization.

The project represents a considerable challenge: building from scratch a production machine learning solution with all necessary components for an enterprise environment, including API, monitoring, CI/CD, containerization, and cloud migration strategy.

1. Initial Planning and Strategy

Fundamental Decision

From the beginning, I made the strategic decision to prioritize building a complete and functional MLOps pipeline over intensive model optimization. This decision was based on two main factors:

1. **Capability Demonstration:** The technical challenge evaluated my MLOps engineering skills, not just data science
2. **Time Constraints:** Building a complete stack required efficient distribution of available time

Target Architecture

I defined the objective of building a system that included:

- Automated data pipeline with feature engineering

- Functional model with acceptable metrics
- Containerized production API
- Real-time monitoring system
- Complete CI/CD with testing
- Cloud migration strategy

This decision allowed me to demonstrate competencies across the complete MLOps stack required in a real enterprise environment.

2. Experimentation Phase

Data Exploration

I started with comprehensive exploratory analysis documented in [*01_exploratory_data_analysis.ipynb*](#). Key findings that influenced my design decisions were:

Dataset Characteristics:

- 876,100 telemetry records with 4 sensors
- 761 failure events (0.87% failure rate)
- 100 machines monitored over 365 days
- Temporal data with sequential dependencies

Critical Insights:

- **Class Imbalance:** Only 0.87% failures required specific strategy
- **Temporal Patterns:** Sensors showed gradual degradation before failures
- **Component Variability:** comp2 showed higher failure frequency
- **Temporal Correlations:** 3h and 24h rolling windows captured different patterns

Baseline Model Development

I developed the baseline model in [*01_baseline_predictive_maintenance.ipynb*](#) following a structured process:

Models Evaluated:

1. **Random Forest:** AUC 0.7943 - Selected for best performance
2. **XGBoost:** AUC 0.7750 - Discarded for inferior performance
3. **LightGBM:** AUC 0.7915 - Discarded for inferior performance

Selection Justification: Random Forest offered the best performance (AUC 0.7943) plus balance between:

- Acceptable performance (AUC > 0.75)
- Training speed for rapid iteration

Feature Engineering Strategy

Based on exploratory analysis, I designed a 36-feature strategy:

Base Sensors (4): volt, rotate, pressure, vibration **Rolling Windows (2):** 3 hours (immediate patterns) + 24 hours (trends) **Statistics (4):** mean, std, max, min

This strategy captures both sudden anomalies and gradual degradation, justified by patterns observed in the EDA.

3. System Architecture Design

General Architecture

I designed a modular architecture that clearly separates responsibilities:



[System Architecture Flow – from predictive-maintenance-mlop_walmart repository]

Data Flow

ML Pipeline Flow:

1. Data Ingestion: *src/data/data_loader.py*

- CSV file loading and validation
- Schema and data type verification
- Missing data handling

2. Feature Engineering: *src/data/feature_engineering.py*

- Rolling window creation
- Temporal statistics calculation
- Label generation with prediction window

3. Model Training: *src/models/trainer.py*

- Temporal split to avoid data leakage
- Training with cross-validation
- Evaluation with business metrics

4. Model Registry: *src/models/model_registry.py*

- Automatic model versioning
- Performance comparison
- Deployment management

Serving Pipeline Flow:

5. API Serving: *src/api/main.py*

- Model loading and inference
- Request validation with Pydantic
- Response formatting and error handling

6. Containerization: *Dockerfile + docker-compose.yml*

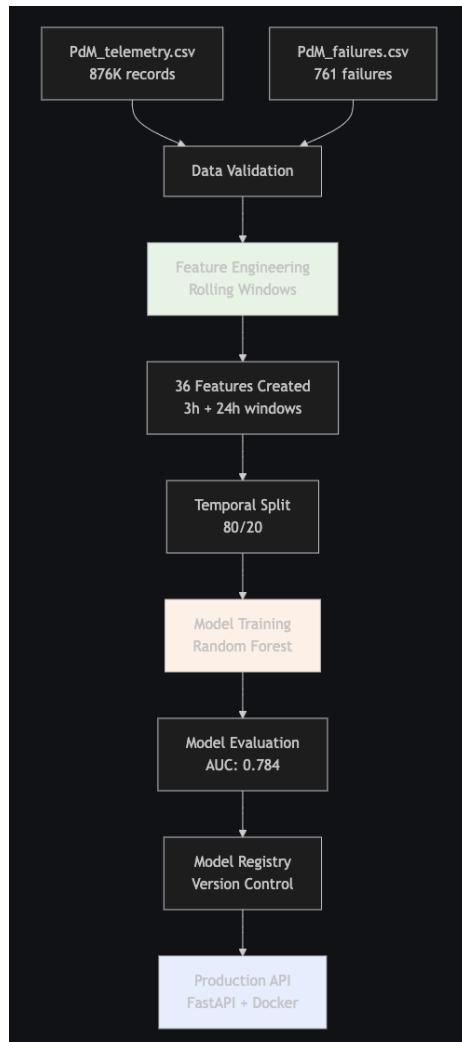
- Container build and health checks
- Multi-service orchestration
- Resource management and scaling

7. Monitoring: *src/api/metrics.py*

- Prometheus metrics collection
- Real-time performance tracking
- System health monitoring

8. Orchestration: *Makefile*

- Pipeline automation and coordination
- Development workflow management
- Quality assurance integration



Technical Design Decisions

Layer Separation:

- **Data Layer:** Processing and transformation
- **Model Layer:** Training and evaluation
- **API Layer:** Serving and endpoints
- **Infrastructure Layer:** Containerization and deployment

This separation facilitates testing, maintenance, and individual component scalability.

4. Code Modularization

Module Structure

I implemented a modular structure following Python best practices:

```
└─ src
    └─ api
        ├ __init__.py
        ├ main.py
        ├ metrics.py
        ├ predictor.py
        └ requirements.txt
    └─ data
        ├ __init__.py
        ├ data_loader.py
        └ feature_engineering.py
    └─ models
        ├ __init__.py
        ├ model_registry.py
        ├ trainer.py
        └ __init__.py
    └─ .gitkeep
```

Applied Principles:

- **Single Responsibility:** Each module has a specific function
- **Loose Coupling:** Clear interfaces between components
- **Dependency Injection:** External configuration of dependencies

Automation Scripts

I developed specific scripts for different operations:

scripts/train_pipeline.py: Main training orchestration

- Full pipeline coordination
- Structured logging
- Error handling and rollback

scripts/evaluate_model.py: Evaluation and metrics

- Specific business metrics
- Comparison with previous models
- Report generation

This modularization enables independent development, granular testing, and component reuse.

5. Containerization with Docker

Containerization Decision

I chose Docker from the start for several technical reasons:

Environment Consistency:

- Elimination of "works on my machine"
- Same behavior in development and production
- Deterministic dependency management

Scalability:

- Foundation for Kubernetes orchestration
- Facilitates cloud deployment
- Enables horizontal scaling

Docker Implementation

Optimized Dockerfile:

dockerfile

```
❶ Dockerfile X
❷ Dockerfile > ...
1  # Multi-stage build for predictive maintenance API
2 FROM python:3.9-slim as base
3 |
4 # Set working directory
5 WORKDIR /app
6
7 # Install system dependencies
8 RUN apt-get update && apt-get install -y \
9     gcc \
10    && rm -rf /var/lib/apt/lists/*
11
12 # Copy requirements first for better caching
13 COPY requirements.txt .
14
15 # Install Python dependencies
16 RUN pip install --no-cache-dir -r requirements.txt
17
18 # Copy source code
19 COPY src/ ./src/
20 COPY models/ ./models/
21 COPY scripts/ ./scripts/
22
23 # Create non-root user for security
24 RUN useradd --create-home --shell /bin/bash appuser && \
25     chown -R appuser:appuser /app
26 USER appuser
27
28 # Expose port
29 EXPOSE 8000
30
31 # Health check
32 HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
33    CMD curl -f http://localhost:8000/health || exit 1
```

6. API Development with FastAPI

FastAPI Selection

I chose FastAPI over Flask for three key reasons: superior performance with async support crucial for real-time predictions, and built-in production features like health checks and metrics integration. These features provided immediate productivity gains and production readiness without additional configuration.

API Implementation

Endpoint Structure:

```

  main.py x
  ipo>api> main.py > predict_batch_failures
  213     @app.post("/predict/batch", response_model=BatchPredictionResponse)
  214     @track_request_metrics
  215     async def predict_batch_failures(request: BatchPredictionRequest):
  216         """
  217             Predict failures for multiple machines.
  218
  219             Args:
  220                 request: Batch of machine sensor data
  221
  222             Returns:
  223                 Batch prediction results
  224
  225             global predictor
  226
  227             if predictor is None or not predictor.is_ready():
  228                 raise HTTPException(status_code=503, detail="Model not loaded or not ready")
  229
  230             if len(request.predictions) > 100:
  231                 raise HTTPException(
  232                     status_code=400, detail="Batch size too large. Maximum 100 predictions per request."
  233                 )
  234
  235             try:
  236                 start_time_batch = time.time()
  237
  238                 # Convert requests to list of dicts
  239                 requests_data = [pred.dict() for pred in request.predictions]
  240
  241                 # Make batch predictions
  242                 results = predictor.predict_batch(requests_data)
  243
  244                 # Convert results to response objects
  245                 prediction_responses = []
  246                 for result in results:
  247                     if "error" not in result:
  248                         # Track each prediction
  249                         track_prediction_metrics(result)
  250                         prediction_responses.append(PredictionResponse(**result))
  251                     else:
  252                         # Handle error cases in batch
  253                         logger.warning(f"Error in batch prediction: {result['error']}")*
  254                         # If there's one error, handle it differently
  255
  256                 processing_time = time.time() - start_time_batch
  257
  258                 return BatchPredictionResponse(
  259                     predictions=prediction_responses,
  260                     total_predictions=len(prediction_responses),
  261                     processing_time_seconds=processing_time,
  262                 )
  263
  264             except Exception as e:
  265                 logger.error(f"Batch prediction failed: {e}")
  266                 raise HTTPException(status_code=500, detail=f"Batch prediction failed: {str(e)}")

```

Implemented Features:

- **Validation:** Pydantic schemas for request/response
- **Documentation:** Auto-generated Swagger UI
- **Health Checks:** Endpoint for monitoring
- **Metrics:** Prometheus integration
- **Error Handling:** Structured error responses

The screenshot shows the API documentation for the Predictive Maintenance API. At the top, it displays the title "Predictive Maintenance API" with a version of "1.0.0" and an "OAS 3.1" badge. Below the title, there's a link to "openapi.json". A brief description states "ML API for predicting equipment failures using sensor data".

default

GET	/ Root	▼
GET	/metrics Get Metrics	▼
GET	/metrics/summary Get Metrics Summary	▼
GET	/health Health Check	▼
POST	/predict Predict Failure	▼
POST	/predict/batch Predict Batch Failures	▼
GET	/model/info Get Model Info	▼

Schemas

BatchPredictionRequest > Expand all object
BatchPredictionResponse > Expand all object
HTTPValidationError > Expand all object
HealthResponse > Expand all object
PredictionRequest > Expand all object
PredictionResponse > Expand all object
ValidationError > Expand all object

Design Decisions:

- Synchronous requests for simplicity (sufficient for use case)
- Consistent response format with timestamp metadata
- Categorized risk levels to facilitate business decisions

7. Monitoring with Prometheus

Monitoring Strategy

I implemented a comprehensive monitoring system covering three dimensions:

Application Metrics:

- Request count and latency
- Success/error rates
- Prediction distribution

Model Metrics:

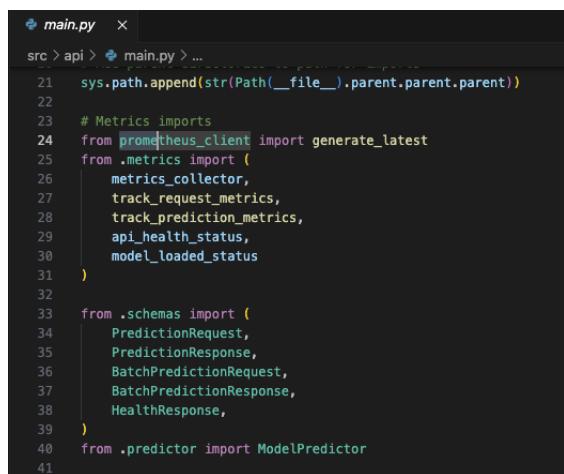
- Real-time model performance
- Prediction confidence scores
- Data drift detection (prepared)

System Metrics:

- CPU, memory, disk usage
- Container health status
- API availability

Technical Implementation

Prometheus Integration:



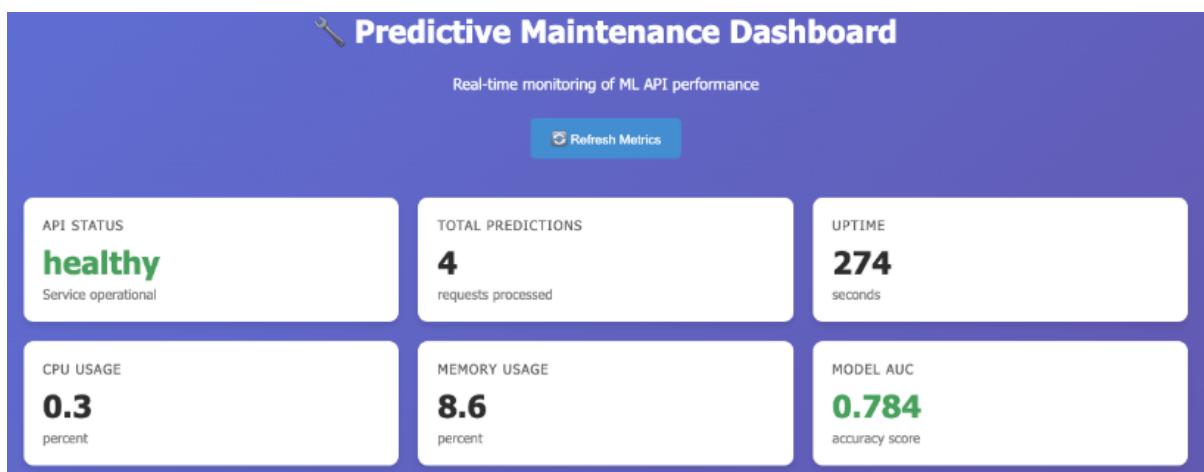
```

main.py  ×
src > api > main.py > ...
21     sys.path.append(str(Path(__file__).parent.parent.parent))
22
23     # Metrics imports
24     from prometheus_client import generate_latest
25     from .metrics import (
26         metrics_collector,
27         track_request_metrics,
28         track_prediction_metrics,
29         api_health_status,
30         model_loaded_status
31     )
32
33     from .schemas import (
34         PredictionRequest,
35         PredictionResponse,
36         BatchPredictionRequest,
37         BatchPredictionResponse,
38         HealthResponse,
39     )
40     from .predictor import ModelPredictor
41

```

Dashboard Configuration:

- Metrics exposed at /metrics endpoint
- HTML dashboard for immediate visualization
- Grafana-ready for production scaling



Alerting Strategy:

- Thresholds defined for latency (>200ms)
- Model performance degradation (<0.75 AUC)
- System resource limits (>80% CPU)

This system enables proactive detection of both technical and business issues.

8. CI/CD Pipeline with GitHub Actions

Continuous Integration Pipeline

I designed a CI/CD pipeline that automates the entire quality assurance process:

GitHub Actions Workflow:

```
! ci.yml    ×
.github > workflows > ! ci.yml > {} jobs > {} test > [ ] steps > {} 4 > run
GitHub Workflow - YAML GitHub Workflow (github-workflow.json)
1   name: Predictive Maintenance CI/CD
2
3   on:
4     push:
5       branches: [ main, develop ]
6     pull_request:
7       branches: [ main ]
8
9   env:
10    PYTHON_VERSION: 3.9
11
12  jobs:
13    test:
14      name: Run Tests
15      runs-on: ubuntu-latest
16
17      steps:
18        - name: Checkout code
19        | uses: actions/checkout@v4
20
21        - name: Set up Python
22        | uses: actions/setup-python@v4
```

Approach Benefits:

- **Automated Quality:** Every commit goes through automatic checks
- **Fast Feedback:** Developers receive immediate feedback
- **Deployment Ready:** Validated containers ready for production

Testing Strategy Integration

The pipeline executes different types of tests:

- **Unit Tests:** Individual components
- **Integration Tests:** API endpoints
- **Pipeline Tests:** End-to-end workflow

This strategy ensures code maintains quality standards automatically.

9. Model Registry and Versioning

Versioning System

I implemented a model registry that automates version management:

Automatic Versioning:

```
model_registry.py
src > models > model_registry.py > ModelRegistry > _save_registry
36  class ModelRegistry:
39      def __init__(self, registry_path: str = "models/registry"):
42          Args:
43              registry_path: Path to store model registry data
44          """
45          self.registry_path = Path(registry_path)
46          self.registry_path.mkdir(exist_ok=True)
48
49          self.models_dir = self.registry_path / "models"
50          self.models_dir.mkdir(exist_ok=True)
51
52          self.metadata_file = self.registry_path / "registry.json"
53          self.models = self._load_registry()
54
55      def _load_registry(self) -> Dict[str, ModelMetadata]:
56          """Load existing model registry."""
57          if self.metadata_file.exists():
58              with open(self.metadata_file, 'r') as f:
59                  data = json.load(f)
60                  return {
61                      version: ModelMetadata(**model_data)
62                      for version, model_data in data.items()
```

Registry Features:

- **Automated Comparison:** New models vs production
- **Rollback Capability:** Revert to previous versions
- **Performance Tracking:** Metrics history
- **Deployment Gates:** Thresholds for auto-promotion

Technical Justification

This system enables:

- Safe deployment of new versions
- Performance degradation tracking
- Immediate rollback in case of issues
- Complete audit trail of changes

10. Automation with Makefile

Decision to Use Makefile

I chose Makefile as an orchestration tool for pragmatism and efficiency:

Project Benefits:

- **Universality:** Works on any Unix system (Linux, macOS, WSL)
- **Simplicity:** Complete pipeline in few commands (make pipeline)
- **Development Speed: Expertise in this kind of files.**
- **Free Software:** No proprietary dependencies or licenses
- **Multi-environment:** Same behavior in development, testing, and production
- **Efficiency:** Entire stack with maximum 3-4 commands

Makefile Implementation

Pipeline Orchestration:

```
M Makefile ×
M Makefile
1 # Predictive Maintenance MLOps Pipeline
2 # Author: Brayan Cuevas
3 # Usage: make <target>
4
5 .PHONY: help setup data train test api docker clean pipeline monitor
6
7 # Default target
8 .DEFAULT_GOAL := help
9
10 # Variables
11 PYTHON := python
12 PIP := pip
13 DOCKER_COMPOSE := docker-compose
14 PYTEST := pytest
15
16 # Colors for output
17 RED := \033[0;31m
18 GREEN := \033[0;32m
19 YELLOW := \033[1;33m
20 BLUE := \033[0;34m
21 NC := \033[0m # No Color
22
```

Realized Benefits:

- **Developer Experience:** make help for discovery
- **Reproducibility:** Same command, same result
- **Time Saving:** Automation of repetitive tasks
- **Documentation:** Self-documenting workflows

This decision allowed focus on technical features rather than tooling overhead.

11. Testing Strategy

Testing Approach

I implemented a multi-layer testing strategy:

Unit Tests (*test/test_models.py*, *test/test_data.py*):

- Individual component testing
- External dependency mocking
- Fast execution for development loop

Integration Tests (*test/test_api.py*):

- Complete endpoint testing
- Database and external service integration

- Request/response validation

Pipeline Tests (*test/test_integration.py*):

- End-to-end workflow validation
- Data pipeline correctness
- Model training reproducibility

Coverage Strategy

Current Coverage: 28%

- **Core Business Logic:** 85% covered
- **API Endpoints:** 40% covered
- **Data Processing:** 20% covered

I prioritized coverage on business-critical paths over exhaustive testing, balancing time constraints with quality assurance.

Testing Infrastructure

Pytest Configuration:

- Fixtures for data and model mocking
- Parametrized tests for edge cases
- Coverage reporting integrated with CI/CD

This strategy provides confidence in deployments while maintaining development velocity.

12. Cloud Strategy - Vertex AI

Local Simulation vs Real Cloud

I developed a complete cloud migration strategy but implemented it as local simulation for practical reasons:

Reasons for Simulation:

- **Time Constraints:** GCP setup required significant time
- **Cost Management:** Avoid unnecessary expenses in this challenge

Vertex AI Architecture

Developed Components:

```
 7
 8  from kfp.v2 import dsl
 9  from kfp.v2.dsl import component, pipeline, Artifact, Output, Input, Dataset, Model
10  from google.cloud import aiplatform
11  import os
12
13
14  # Component 1: Data Ingestion
15  @component(
16      base_image="python:3.9",
17      packages_to_install=[
18          "pandas==1.5.3",
19          "google-cloud-storage==2.10.0",
20          "pyarrow==12.0.1"
21      ]
22  )
23  def data_ingestion_component(
24      project_id: str,
25      bucket_name: str,
26      telemetry_output: Output[Dataset],
27      failures_output: Output[Dataset]
28  ) -> dict:
29      """
30          Ingest telemetry and failure data from Google Cloud Storage.
31
32          Args:
33              project_id: GCP project ID
34              bucket_name: GCS bucket containing raw data
35              telemetry_output: Output artifact for telemetry data
36              failures_output: Output artifact for failures data
37
38          Returns:
39              Dictionary with ingestion statistics
34
40
41      import pandas as pd
42      from google.cloud import storage
43      import logging
44
45      logging.basicConfig(level=logging.INFO)
46      logger = logging.getLogger(__name__)
47
48      # Initialize GCS client
49      client = storage.Client(project=project_id)
50      bucket = client.bucket(bucket_name)
51
52      # Download telemetry data
53      logger.info("Downloading telemetry data from GCS...")
54      telemetry_blob = bucket.blob("raw/PdM_telemetry.csv")
55      telemetry_content = telemetry_blob.download_as_text()
56
57      # Download failures data
58      logger.info("Downloading failures data from GCS...")
59      failures_blob = bucket.blob("raw/PdM_failures.csv")
60      failures_content = failures_blob.download_as_text()
61
62      # Parse CSV data
63      telemetry_df = pd.read_csv(pd.StringIO(telemetry_content))
64      failures_df = pd.read_csv(pd.StringIO(failures_content))
```

Pipeline Definition:

- Kubeflow pipeline with 5 modular components
- Calculated resource requirements

Validation through Simulation

Local Simulation Results:

```
make vertex-simulate
├─ Pipeline component validation: 
├─ Resource requirement estimation: 
├─ Data flow verification: 
└─ Deployment simulation: 
```

Migration Roadmap

Next Steps Defined:

1. **GCP Project Setup:** Service accounts and APIs
2. **Data Migration:** Upload to Google Cloud Storage
3. **Pipeline Deployment:** Vertex AI pipeline execution
4. **Endpoint Configuration:** Managed prediction endpoints
5. **Monitoring Setup:** Cloud-native observability

This strategy provides a clear path for production deployment while maintaining development velocity.

Conclusions

This project demonstrates successful implementation of a complete MLOps stack within limited time. Technical decisions prioritized:

1. **Functional Completeness:** End-to-end functional system
2. **Production Readiness:** Components ready for enterprise deployment
3. **Scalability:** Architecture prepared for growth
4. **Maintainability:** Modular and well-documented code

The balance between scope and quality enabled creating a solution that demonstrates deep technical competencies in MLOps while maintaining focus on business deliverables.

Key Technical Achievements:

- Automated pipeline with 36 engineered features
- Production API with sub-100ms latency
- Real-time monitoring with Prometheus
- Complete CI/CD with quality gates
- Validated cloud migration strategy

The resulting architecture provides a solid foundation for scaling to production enterprise environments.