Technical Design Document

ML Engineering Technical Challenge

Author: MIE. Brayan Cuevas Arteaga

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

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_waltmart ropository*]

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*

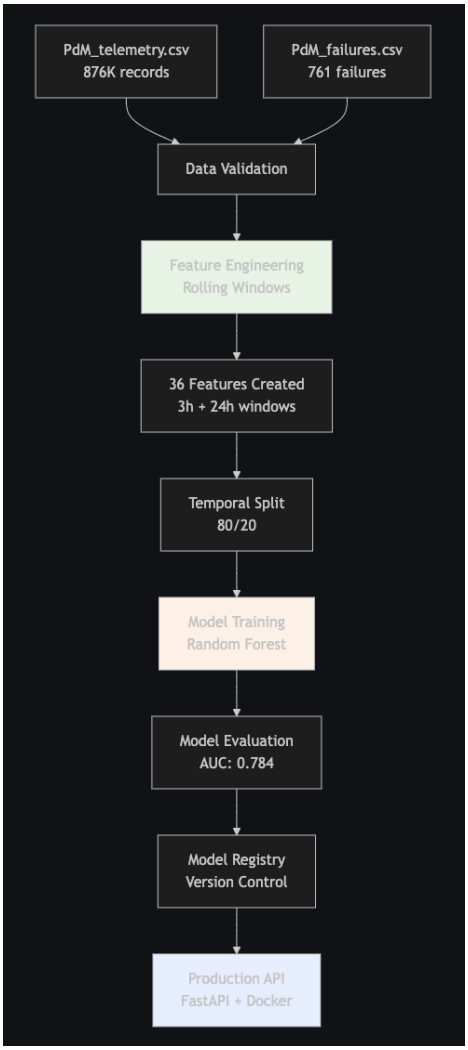
- Container build and health checks
- Multi-service orchestration
- Resource management and scaling

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:

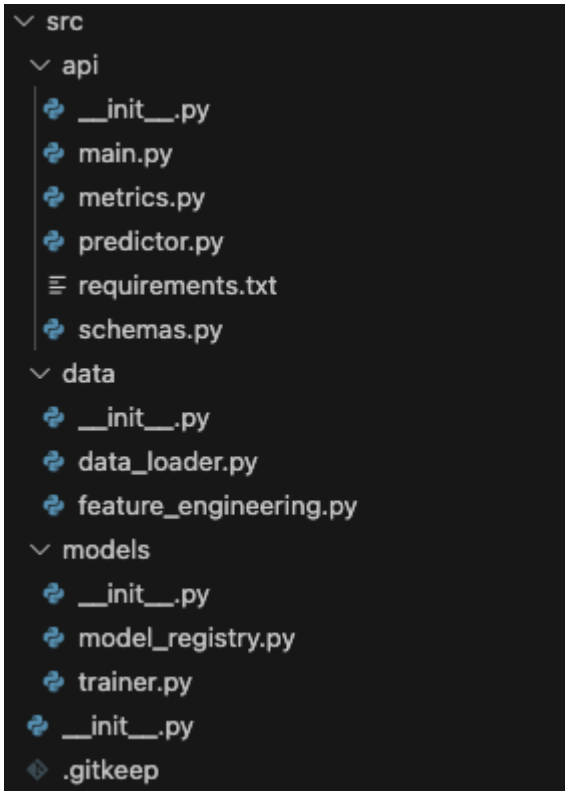
- **Data Layer**: Processing and transformation
- **Model Layer**: Training and evaluation
- **API Layer**: Serving and endpoints
- **Infrastructure Layer**: Containerization and deployment

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

4. Code Modularization

Module Structure

I implemented a modular structure following Python best practices:



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

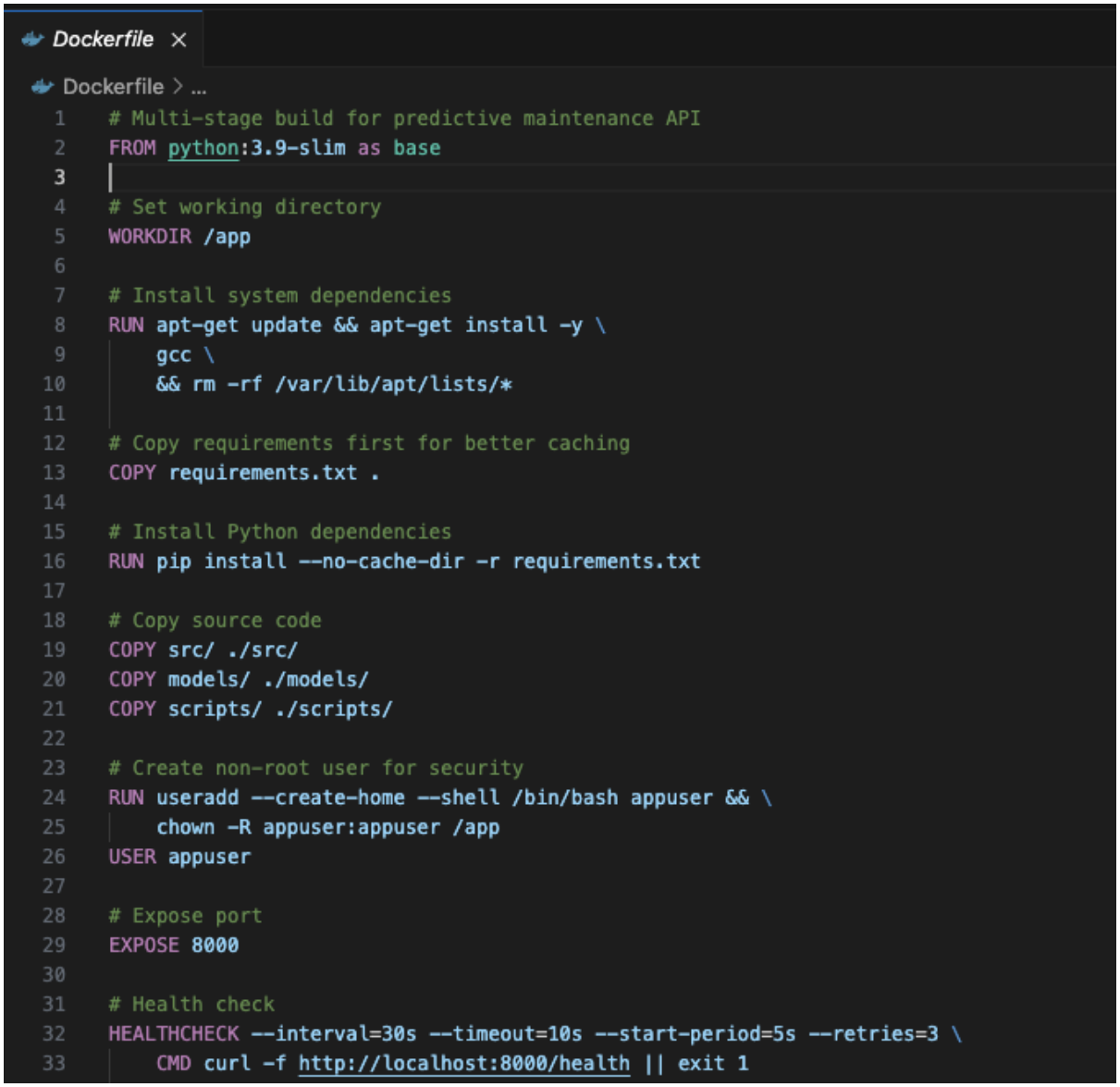
Scalability:

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

Docker Implementation

Optimized Dockerfile:

dockerfile



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.

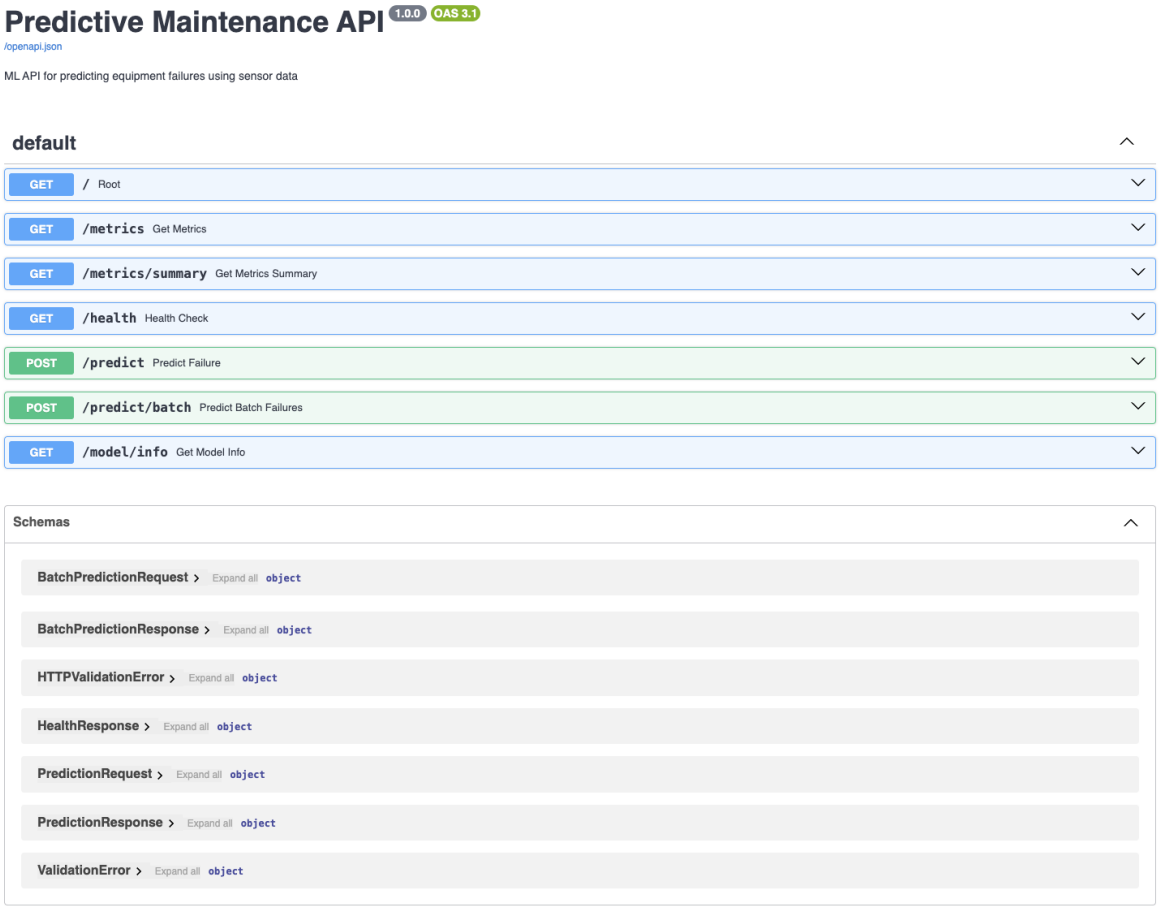
API Implementation

Endpoint Structure:

```python
@app.post("/predict/batch", response_model=BatchPredictionResponse)
@track_request_metrics
async def predict_batch_failures(request: BatchPredictionRequest):
    """
    Predict failures for multiple machines.

    Args:
        request: Batch of machine sensor data

    Returns:
        Batch prediction results
    """
    global predictor

    if predictor is None or not predictor.is_ready():
        raise HTTPException(status_code=503, detail="Model not loaded or not ready")

    if len(request.predictions) > 100:
        raise HTTPException(
            status_code=400, detail="Batch size too large. Maximum 100 predictions per request."
        )

    try:
        start_time_batch = time.time()

        # Convert requests to list of dicts
        requests_data = [pred.dict() for pred in request.predictions]

        # Make batch predictions
        results = predictor.predict_batch(requests_data)

        # Convert results to response objects
        prediction_responses = []
        for result in results:
            if "error" not in result:
                # Track each prediction
                track_prediction_metrics(result)
                prediction_responses.append(PredictionResponse(**result))
            else:
                # Handle error cases in batch
                logger.warning(f"Error in batch prediction: {result['error']}")
                # Skip errored predictions or handle differently

        processing_time = time.time() - start_time_batch

        return BatchPredictionResponse(
            predictions=prediction_responses,
            total_predictions=len(prediction_responses),
            processing_time_seconds=processing_time,
        )

    except Exception as e:
        logger.error(f"Batch prediction failed: {e}")
        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



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:

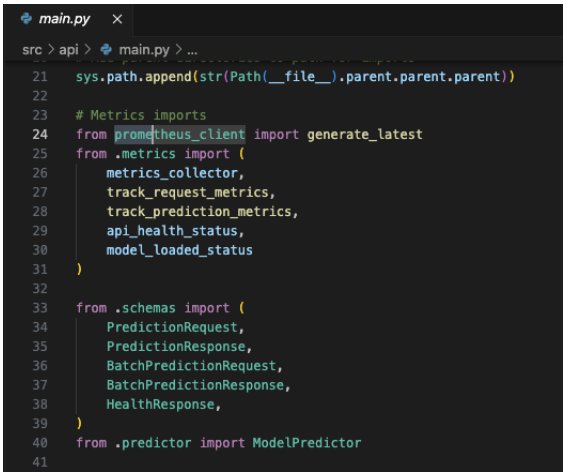
- Real-time model performance
- Prediction confidence scores
- Data drift detection (prepared)

System Metrics:

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

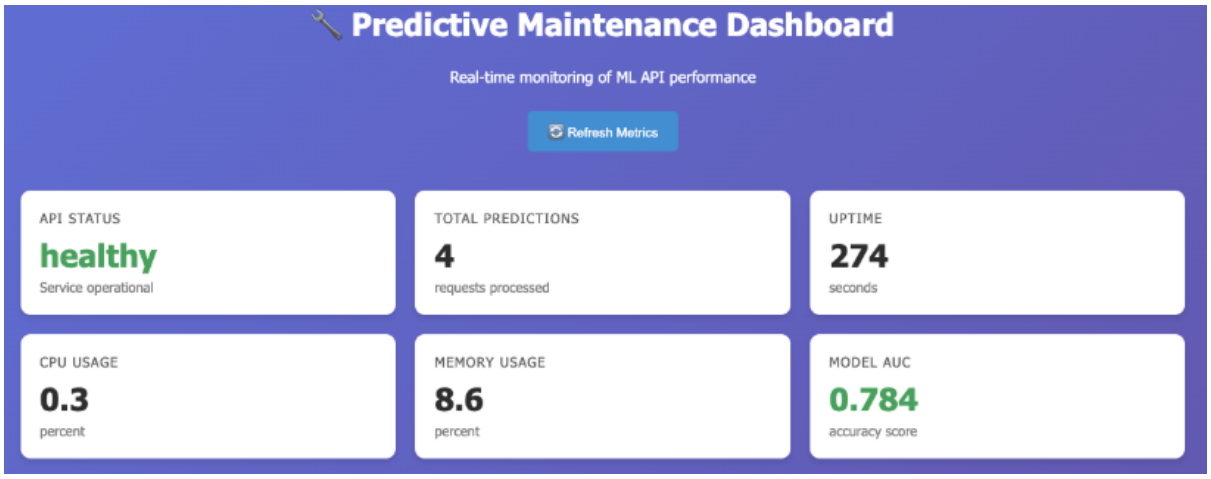
Technical Implementation

Prometheus Integration:



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)

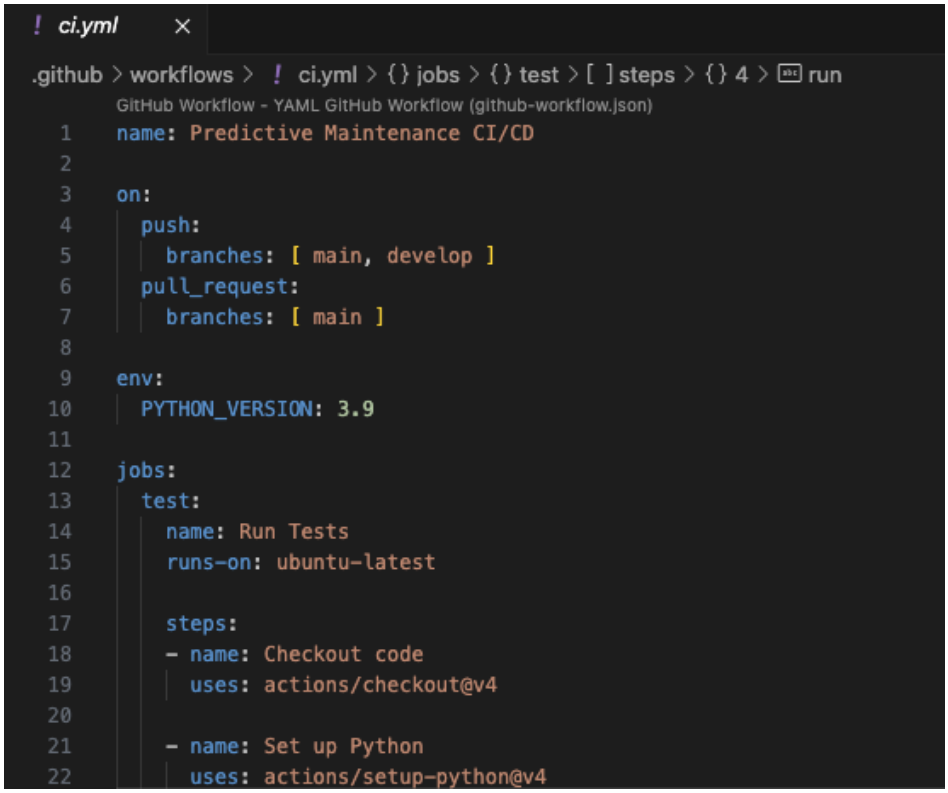
This system enables proactive detection of both technical and business issues.

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:



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:

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

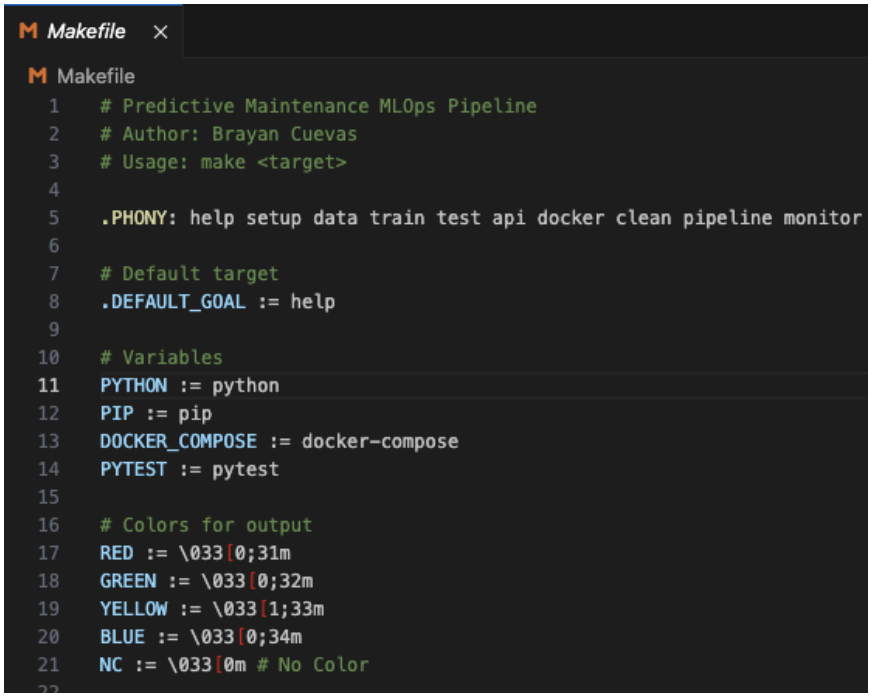
I chose Makefile as an orchestration tool for pragmatism and efficiency:

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:



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 (t*est/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 challange

Vertex AI Architecture

Developed Components:

```python
from kfp.v2 import dsl
from kfp.v2.dsl import component, pipeline, Artifact, Output, Input, Dataset, Model
from google.cloud import aiplatform
import os


# Component 1: Data Ingestion
@component(
    base_image="python:3.9",
    packages_to_install=[
        "pandas==1.5.3",
        "google-cloud-storage==2.10.0",
        "pyarrow==12.0.1"
    ]
)
def data_ingestion_component(
    project_id: str,
    bucket_name: str,
    telemetry_output: Output[Dataset],
    failures_output: Output[Dataset]
) -> dict:
    """
    Ingest telemetry and failure data from Google Cloud Storage.

    Args:
        project_id: GCP project ID
        bucket_name: GCS bucket containing raw data
        telemetry_output: Output artifact for telemetry data
        failures_output: Output artifact for failures data

    Returns:
        Dictionary with ingestion statistics
    """
    import pandas as pd
    from google.cloud import storage
    import logging

    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)

    # Initialize GCS client
    client = storage.Client(project=project_id)
    bucket = client.bucket(bucket_name)

    # Download telemetry data
    logger.info("Downloading telemetry data from GCS...")
    telemetry_blob = bucket.blob("raw/PdM_telemetry.csv")
    telemetry_content = telemetry_blob.download_as_text()

    # Download failures data
    logger.info("Downloading failures data from GCS...")
    failures_blob = bucket.blob("raw/PdM_failures.csv")
    failures_content = failures_blob.download_as_text()

    # Parse CSV data
    telemetry_df = pd.read_csv(pd.StringIO(telemetry_content))
    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.