AIOps Multi-Service Monitoring & Anomaly Detection Platform

Technical Architecture & System Review Report

---

1. Executive Summary

What the system does

This system is an **AIOps control plane** that monitors multiple Spring Boot microservices in real time, detects abnormal behavior using machine learning, correlates anomalies with application logs, and surfaces actionable incidents through a React-based UI and Slack alerts.

It continuously:

- Pulls **CPU, memory, and latency metrics** from Prometheus
- Classifies system health using **static thresholds**
- Detects anomalies using **Isolation Forest ML models**
- Persists historical and anomalous metric data
- Correlates metric spikes with **structured application logs**
- Visualizes live health, incidents, and logs in a single dashboard

Why it exists

Traditional monitoring tells you *what* broke but not *why*. This system bridges that gap by combining:

- Metrics (Prometheus)
- Logs (Spring Boot structured JSON logs)
- ML-based anomaly detection
- Temporal correlation

The goal is **faster root cause analysis**, reduced MTTR, and a foundation for predictive operations.

High-level architecture overview

- **React (TypeScript)**: Control plane UI
- **FastAPI (Python)**: Metrics ingestion, ML inference, log correlation
- **Spring Boot**: Instrumented application emitting metrics and logs
- **Prometheus**: Metrics source
- **Filesystem-based storage**: Datasets, trained models, logs
- **Isolation Forest**: Per-service, per-metric anomaly detection
- **Slack**: Alerting channel

This is an **MVP-grade AIOps platform**, not yet enterprise-hardened, but architecturally sound.

---

2. Problem Statement & Objectives

Real-world problem

In microservice systems:

- Metrics live in one tool
- Logs live in another
- Alerts lack context
- Root cause analysis is slow and manual

Operators often detect incidents *after* users are impacted.

Objectives

Functional goals

- Monitor multiple services dynamically
- Detect metric anomalies automatically
- Correlate metrics with logs at event time
- Visualize incidents in real time
- Notify operators proactively

Non-functional goals

- Near real-time responsiveness (5–30 sec)
- Extensible to new services without redeploy
- Minimal operational overhead
- Clear separation of concerns

---

3. Technology Stack

Frontend: React + TypeScript + MUI

Why

- Strong typing for incident models
- Component-driven architecture
- Fast iteration for dashboards
- Mature UI library (Material UI)

Used appropriately for:

- Real-time polling
- Incident state tracking
- Filtering and sorting logic on client side

Backend: FastAPI (Python)

Why

- Lightweight, fast I/O
- Natural fit for ML inference
- Easy Prometheus and filesystem integration
- APScheduler for background retraining

FastAPI is used as a **control and intelligence plane**, not as a business API.

ML & Data: Python + scikit-learn

Why

- Isolation Forest suits unlabeled anomaly detection
- Rolling-window statistical features
- Offline training, lightweight inference

Metrics: Prometheus

Why

- Industry standard
- Native Spring Boot integration
- Histogram support for latency percentiles

Application: Spring Boot

Why

- Micrometer metrics
- Actuator exposure
- Structured logging
- Realistic failure simulation endpoints

Alerting: Slack Webhooks

Why

- Simple, effective MVP alert channel
- Easy replacement with PagerDuty/Opsgenie later

---

4. System Architecture

Logical architecture

```
Spring Boot Service(s)
  ├── Micrometer Metrics
  ├── JSON Logs
  ↓
Prometheus
  ↓
FastAPI AIOps Engine
  ├── Metrics Query
  ├── Threshold Classification
  ├── ML Inference
  ├── Dataset Persistence
  ├── Log Correlation
  ├── Slack Alerts
  ↓
React Control Plane
```

Component interactions

1. React polls `/metrics?service=X`
2. FastAPI queries Prometheus
3. Metrics are classified + inferred
4. Results persisted to datasets
5. Incidents created on threshold breach
6. Logs correlated using timestamp window
7. UI displays metrics + incidents
8. Slack notified for High/Critical states

Data flow (textual diagram)

```
Prometheus → FastAPI → CSV Datasets → ML Training
                                ↓
                          Models (.pkl)
                                ↓
                       Real-time Inference
                                ↓
                 Metrics + Logs → Incidents → UI
```

5. Frontend Design (React)

Folder structure

- `App.tsx`: Control plane orchestration
- `MetricCard.tsx`: Metric visualization component
- `components/`: Reusable UI elements

Component responsibilities

- **App.tsx**
  - Service discovery
  - Polling logic
  - Incident lifecycle management
  - Filtering, sorting, search
- **MetricCard**
  - Single-metric status visualization

State management

- Local React state with refs
- `incidentsRef` prevents duplicate incident creation
- No global state library used (acceptable for MVP)

API integration

- Axios-based polling
- Pull model instead of push
- Event timestamp passed for log correlation

UI/UX considerations

- Clear severity coloring
- Operator-centric filtering
- Monospace logs for readability
- Manual incident resolution

Error handling (gap)

- No retry logic
- No API error UI
- Should add:
    - Axios interceptors
    - Error banners
    - Backoff strategy

---

6. Backend Design (FastAPI)

Structural responsibilities

- Metrics ingestion
- Classification
- ML inference
- Data persistence
- Log correlation
- Scheduling

Key modules

- `app_v2.py`: Main API
- `train_v2.py`: Offline training
- `feature_engineering.py`: Feature pipeline
- `config.yaml`: Training config (partially unused)

Controllers

- `/services`
- `/metrics`
- `/logs`
- `/retrain`

ML lifecycle

- Per-service, per-metric models
- Rolling window training
- Cached models in memory
- Daily retraining via APScheduler

Exception handling (gap)

- Mostly print-based
- No global exception handler
- No structured API errors

Recommendation

- Add FastAPI exception middleware
- Standard error response schema

---

7. Database Design

Current approach

- CSV-based persistence
- Directory-per-service structure

```
datasets/
  service-name/
    history/
      cpu.csv
      mem.csv
      lat.csv
    anomaly/
      cpu.csv
      mem.csv
      lat.csv
```

Pros

- Simple
- Transparent
- Easy debugging

Cons

- Not concurrent-safe
- No indexing
- No retention policy
- Not production-grade

Production recommendation

- Time-series DB (TimescaleDB / InfluxDB)
- Or object storage + Parquet

---

8. Python Scripts

`train_v2.py`

- Discovers services dynamically
- Loads rolling window data
- Trains Isolation Forest models
- Saves models to disk

`feature_engineering.py`

- Rolling mean, std
- Delta
- Z-score normalization
- Stateless and reusable

Performance considerations

- Training scales linearly per service
- Acceptable for small to medium environments
- Needs job queue for large fleets

---

9. API Design

Key endpoints

Endpoint	Purpose
GET /services	Discover monitored services
GET /metrics	Fetch metrics + inference
GET /logs	Correlate logs
POST /retrain	Manual retraining

Authentication

- None (development-only)

Error responses (gap)

- No consistent error format
- No HTTP status mapping

---

10. Security Considerations

Current state

- No authentication
- Open CORS
- Local file system access
- Slack webhook in env

Risks

- Unauthenticated control endpoints
- Log file path hardcoded
- Command execution via `os.system`

Required fixes

- API auth (JWT / mTLS)
- Secrets manager
- Path sanitization
- Replace `os.system` with subprocess
- Role-based access for retraining

11. Deployment & Environment Setup

Local setup

- Prometheus running locally
- Spring Boot on port 8989
- FastAPI on port 9092
- React on 5173

Environment variables

- `SLACK_WEBHOOK_URL`

Missing

- Dockerfiles
- Compose or Helm charts
- CI pipeline

Production strategy

- Containerize all components
- Centralized logging
- External model storage
- Blue-green deployment for retraining

---

12. Testing Strategy

Current state

- No tests present

What should be added

- Unit tests for feature engineering
- Mocked Prometheus queries
- API contract tests
- UI component tests
- Load tests for polling pressure

---

13. Logging, Monitoring & Error Handling

Logging

- Spring Boot emits structured JSON
- FastAPI uses print statements (weak)

Monitoring

- Prometheus covers only app metrics
- No monitoring of FastAPI itself

Recommendations

- Structured logging in FastAPI
- Self-monitoring metrics
- Alert on training failures

14. Performance & Scalability

Bottlenecks

- Polling-based frontend
- CSV file writes
- Single-node ML training

Scaling ideas

- WebSockets for push updates
- Async writes
- Distributed training jobs
- Caching Prometheus queries

---

15. Limitations & Risks

- Not multi-tenant
- No persistence for incidents
- No auth
- File-based storage
- Single point of failure (FastAPI)

These are **acceptable for MVP**, not for enterprise.

---

16. Future Enhancements

- Incident persistence (DB)
- Root cause ranking
- Service dependency graph
- Predictive forecasting
- Auto-remediation hooks
- Role-based UI
- SLA dashboards

---

17. Conclusion

System maturity

Strong MVP / Advanced Capstone level

Readiness

- Excellent for demos, learning, startups
- Not production-safe without hardening

Final evaluation

This is a **well-thought AIOps system** with:

- Clear architectural separation
- Real ML usage (not cosmetic)
- Practical log correlation
- Operator-friendly UI

With security, persistence, and deployment improvements, this can evolve into a **real production AIOps platform**.