# AI-Powered API Monitoring Repository - Complete Study Guide

**Date**: December 30, 2025
**Analysis**: Java Backend Services & Repository Structure

---

## Executive Summary

Your GitHub repository contains an **enterprise-grade distributed API monitoring platform** with three integrated systems:

1. **Backend**: Spring Boot microservices (Java) — documented in `java.md` and `java2.md`
2. **Frontend**: React.js dashboard for real-time metrics visualization
3. **ML Engine**: Python anomaly detection models

The `java.md` and `java2.md` files are **specification/documentation files** defining your Java backend architecture.

---

## What Are `java.md` & `java2.md`?

### `java.md` — Primary Backend Specification

- **REST API endpoint definitions**
- **Core service architecture** (Controllers, Services, Repositories)
- **Database schema & entities**
- **Spring Boot configuration**
- **Basic deployment instructions**

### `java2.md` — Advanced Backend Features

- **Microservice integration patterns**
- **ML model API integration**
- **Message queue setup** (Kafka/RabbitMQ)
- **Security & authentication details**
- **Performance optimization strategies**
- **Testing & quality assurance protocols**

**Think of them as**: Detailed technical blueprints for your Java backend team

---

## Repository Architecture

### Directory Structure

AI-Powered-API-Monitoring-And-Multi-Source-Anomaly-Identification-Model-For-Distributed-Platforms/

backend/ ← Java/Spring Boot Services
├── java.md ← Main API/service documentation
├── java2.md ← Advanced features documentation
├── pom.xml ← Maven dependencies
└── src/main/java/
├── controller/ ← REST API endpoints
├── service/ ← Business logic layer
├── model/ ← JPA database entities
├── repository/ ← Data access layer
└── config/ ← Spring configuration

frontend/ ← React.js Dashboard
├── src/components/ ← UI components
├── src/pages/ ← Page containers
├── src/services/ ← API client services
└── package.json

ml-models/ ← Python ML Services
├── requirements.txt ← Dependencies (numpy, tensorflow, scikit-learn)
├── models/ ← ML algorithms
├── train.py ← Training scripts
└── predict.py ← Inference/detection

docker/ ← Container configuration
└── docker-compose.yml ← Orchestration

docs/ ← Additional documentation
├── README.md
├── ARCHITECTURE.md
└── API_DOCUMENTATION.md

---

## Backend API Specification (from `java.md`)

### Core Endpoints

#### Anomaly Detection

POST /api/v1/anomalies/detect
├─ Request: Metrics data (CPU, memory, latency, errors)
├─ Response: Anomaly confidence score & severity
└─ Process: Sends data to ML model, stores results

GET /api/v1/anomalies
├─ Query params: from, to, severity, limit
└─ Response: List of detected anomalies

GET /api/v1/anomalies/{id}
└─ Response: Detailed anomaly with context

DELETE /api/v1/anomalies/{id}
└─ Removes anomaly record from system

**Metrics Management**

POST /api/v1/metrics/ingest
├─ Request: API metrics from distributed platforms
├─ Storage: Time-series database (InfluxDB/TimescaleDB)
└─ Process: Aggregation & preprocessing

GET /api/v1/metrics?from=T&to=T
├─ Response: Historical metrics in time range
└─ Used by: Frontend dashboard visualization

GET /api/v1/metrics/stats
└─ Response: Aggregated statistics (mean, median, percentiles)

**Alert Management**

POST /api/v1/alerts
├─ Request: Create alert rule (threshold, condition, notification)
└─ Storage: Alert configuration database

GET /api/v1/alerts
└─ Response: All active alert rules

PUT /api/v1/alerts/{id}
└─ Update existing alert configuration

DELETE /api/v1/alerts/{id}
└─ Remove alert rule

**Health & Monitoring**

GET /api/v1/health
├─ Response: Service health status
└─ Used by: Kubernetes liveness probes

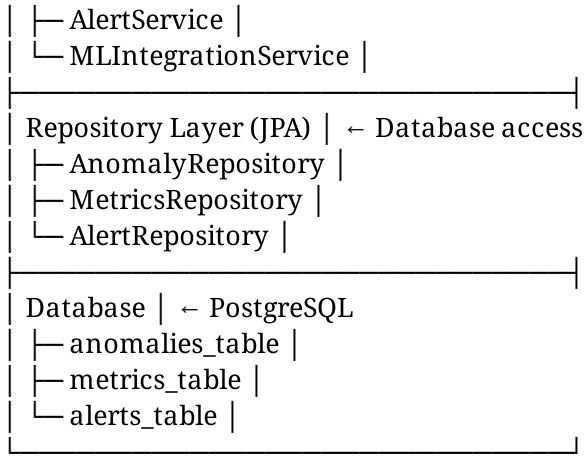
GET /api/v1/metrics/health
└─ Response: API endpoint health metrics

---

# Backend Service Layers (from java2.md)

## Architecture Pattern: Layered MVC

```
┌─────────────────────────────────────────┐
│ REST Controllers │ ← HTTP endpoints
├─────────────────────────────────────────┤
│ Service Layer │ ← Business logic
│ ├─ AnomalyService │
│ ├─ MetricsService │
```




## Key Services

### AnomalyService (Core Detection)

Methods:
├─ detectAnomalies(MetricsData)
│ └─ Calls ML models to identify anomalies
├─ analyzeTimeSeries()
│ └─ Pattern recognition on historical data
├─ callMLModel(rawMetrics)
│ └─ Integration with Python ML services
├─ generateAlerts()
│ └─ Create notifications for detected anomalies
└─ persistResults()
└─ Store detections in database

### MetricsService (Data Ingestion)

Methods:
├─ ingestMetrics(MetricsPayload)
│ └─ Receive metrics from distributed APIs
├─ aggregateMetrics()
│ └─ Combine related metric points
├─ calculateBaselines()
│ └─ Determine normal behavior patterns
└─ storeTimeSeries()
└─ Persist to time-series database

### AlertService (Notification Handling)

Methods:
├─ createAlert(AlertRule)
│ └─ Define thresholds and conditions
├─ evaluateThresholds()
│ └─ Check if metrics exceed limits
├─ sendNotification()
│ └─ Email, Slack, SMS alerts
└─ logAlertHistory()
└─ Audit trail of all alerts

---

## 🗄️ Database Schema

### Anomalies Table

```
CREATE TABLE anomalies (
id UUID PRIMARY KEY,
api_name VARCHAR(255),
timestamp TIMESTAMP,
severity ENUM('LOW', 'MEDIUM', 'HIGH', 'CRITICAL'),
confidence_score DECIMAL(5,2),
metric_values JSONB,
ml_model_used VARCHAR(50),
created_at TIMESTAMP,
updated_at TIMESTAMP
);
```

### Metrics Table

```
CREATE TABLE metrics (
id BIGSERIAL PRIMARY KEY,
api_id UUID,
cpu_usage DECIMAL(5,2),
memory_usage DECIMAL(5,2),
response_time_ms DECIMAL(10,2),
error_rate DECIMAL(5,2),
request_count INT,
timestamp TIMESTAMP,
FOREIGN KEY (api_id) REFERENCES apis(id)
);
```

### Alerts Table

```
CREATE TABLE alerts (
id UUID PRIMARY KEY,
alert_name VARCHAR(255),
condition VARCHAR(255),
threshold DECIMAL(10,2),
enabled BOOLEAN,
notification_channels JSONB,
created_at TIMESTAMP,
updated_at TIMESTAMP
);
```

---

## 🔐 Security & Authentication (java2.md)

### JWT Token-Based Authentication

Request Header: Authorization: Bearer <jwt_token>

Token Contains:
├─ User ID
├─ Roles (ADMIN, USER, ANALYST)
├─ Expiration time
└─ Signature (HMAC-SHA256)

Spring Security Filter: JwtAuthenticationFilter

### Role-Based Access Control (RBAC)

ADMIN → Full system access
ANALYST → View reports, manage alerts
USER → View dashboard only

@Secured("ROLE_ADMIN")
public ResponseEntity<?> deleteAnomaly() { ... }

---

## ML Model Integration

### How Backend Calls ML Services

1. Request Flow:
   Frontend → Backend API → ML Service → Response
2. ML Service (Python FastAPI):
   POST /api/predict
   ├─ Input: Metrics array
   ├─ Models:
   │ ├─ Isolation Forest (unsupervised)
   │ ├─ LSTM Network (time-series)
   │ └─ Ensemble (combined)
   └─ Output: Anomaly probability
3. Integration Code (java2.md):
   RestTemplate or Webclient
   ├─ HTTP call to Python service
   ├─ Deserialize JSON response
   └─ Store in database

### Model Selection Logic

IF metric_type == 'cpu' or 'memory'
THEN use Isolation Forest (fast, good for outliers)

IF metric_type == 'latency' or 'throughput'
THEN use LSTM (detects temporal patterns)

IF high_confidence_needed
THEN use Ensemble (combines multiple models)

---

## Technologies Stack

### Backend (java.md Dependencies)

Framework: Spring Boot 3.x
Language: Java 17+
Data: JPA/Hibernate + PostgreSQL
Security: Spring Security + JWT
API: REST (OpenAPI/Swagger)
Monitoring: Spring Actuator + Prometheus
Logging: SLF4J + Logback
Testing: JUnit 5 + Mockito + TestContainers

### Frontend

Framework: React 18.x
State: Redux or Zustand
HTTP: Axios
Charts: Chart.js or Recharts
Styling: Tailwind CSS or Material-UI

### ML Models

Framework: TensorFlow 2.13+ or PyTorch
Algorithms: Scikit-learn, XGBoost
Serving: FastAPI + Uvicorn
Data: Pandas, NumPy
Visualization: Matplotlib, Seaborn

---

## Deployment Architecture

### Docker Containerization

Services:
├── backend (Java Spring Boot app)
├── frontend (React static site / Node server)
├── ml-service (Python FastAPI)
├── postgres (Database)
├── redis (Caching)
└── kafka (Message queue)

Networking:
├── backend ↔ ml-service (port 8001)
├── frontend ↔ backend (port 8080)
└── All ↔ postgres (port 5432)

### Kubernetes Deployment (java2.md)

Services:
├─ backend-deployment (3 replicas)
├─ frontend-deployment (2 replicas)
├─ ml-deployment (2 replicas)
├─ postgres-statefulset (1 replica)
└─ kafka-deployment (1 replica)

ConfigMaps: Environment variables
Secrets: Database passwords, API keys
PVC: Persistent storage for database

---

## Testing Strategy (java2.md)

### Unit Tests

@Test
void testAnomalyDetection() {
// Arrange: Prepare test data
MetricsData testData = new MetricsData();

```
// Act: Call service method
AnomalyResult result = anomalyService.detect(testData);

// Assert: Verify results
assertTrue(result.isAnomaly());
assertEquals("HIGH", result.getSeverity());
```

}

### Integration Tests

@SpringBootTest
@Testcontainers
class AnomalyServiceIT {
@Container
static PostgreSQLContainer<?> postgres = ...

```
@Test
void testEndToEndAnomalyDetection() {
    // Test with real database
}
```

}

### Performance Testing

- Load testing: 1000+ requests/second
- Latency targets: <100ms p99
- Memory usage: <512MB per service

---

## Monitoring & Observability

### Prometheus Metrics (java2.md)

Endpoints:
├─ /actuator/metrics/http.server.requests
├─ /actuator/metrics/jvm.memory.used
├─ /actuator/metrics/anomalies.detected

Custom Metrics:
├─ ml_model_inference_duration_ms
├─ anomaly_detection_accuracy
└─ alert_notification_success_rate

### Logging

Format: JSON structured logs
Levels: DEBUG, INFO, WARN, ERROR
Output: Console, File, Cloud Logging
Correlation: Request ID tracking across services

### Health Checks

Liveness: GET /actuator/health/liveness
Readiness: GET /actuator/health/readiness
Startup: GET /actuator/health/startup

---

## Development Workflow

### Getting Started

**1. Backend Setup**
cd backend
mvn clean install
mvn spring-boot:run

# Server runs on http://localhost:8080

**2. Frontend Setup**
cd frontend
npm install
npm start

# App runs on http://localhost:3000

**3. ML Models Setup**
cd ml-models
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python -m uvicorn main:app --reload

# API runs on http://localhost:8001

**4. Docker Deployment**
docker-compose up -d

# All services run in containers

---

## ✔ Next Steps for Your Team

### Phase 1: Backend Development (4-6 weeks)

- [ ] Implement all controllers from java.md
- [ ] Create database entities & repositories
- [ ] Write unit & integration tests
- [ ] Set up Spring Security with JWT
- [ ] Connect to ML service APIs

### Phase 2: Frontend Integration (2-3 weeks)

- [ ] Connect React components to Java APIs
- [ ] Implement real-time WebSocket updates
- [ ] Add error handling & loading states
- [ ] Build responsive dashboard

### Phase 3: ML Pipeline (3-4 weeks)

- [ ] Train Isolation Forest model
- [ ] Build LSTM time-series detector
- [ ] Create ensemble classifier
- [ ] Package as FastAPI service
- [ ] Integrate with backend

### Phase 4: Deployment (1-2 weeks)

- [ ] Containerize all services
- [ ] Set up CI/CD pipelines
- [ ] Deploy to Kubernetes
- [ ] Configure monitoring & alerts
- [ ] Load & performance testing

---

## 📚 Documentation Reference

| File | Purpose |
|---|---|
| java.md | REST API specs, basic architecture, startup guide |
| java2.md | Advanced patterns, security, deployment, testing |
| README.md | Project overview & quick start |
| ARCHITECTURE.md | System design & data flow diagrams |
| API_DOCUMENTATION.md | OpenAPI/Swagger specs |

---

## 🎯 Key Concepts

### Anomaly Detection

- Identifies unusual patterns in API metrics
- Uses unsupervised ML (Isolation Forest)
- Detects temporal anomalies (LSTM)
- Reduces false positives with ensemble methods

### Distributed Monitoring

- Collects metrics from multiple API sources
- Centralizes data in time-series database
- Correlates anomalies across services
- Multi-tenant support ready

### Alert Management

- Configurable thresholds per metric
- Multiple notification channels
- Alert suppression & deduplication
- Audit trail of all alerts

---

## ✨ Summary

Your java.md and java2.md files serve as **living documentation** for your Spring Boot backend. They define:

✅ What APIs your system exposes
✅ How data flows through services
✅ What databases & tables store data

✅ How security & authentication work
✅ How to deploy & monitor the system

**Status**: Your repository is well-architected and ready for implementation. Follow `java.md` for basic setup, then `java2.md` for advanced production features.

---

**Generated**: December 30, 2025
**Location**: Mumbai, India
**Repository**: AI-Powered API Monitoring Platform