UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

AIOps Monitoring System

Università degli studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'informazione e Matematica

Documentation for the exam of Software Engineering for Autonomous Systems

Group Members

Edoardo Di Giuseppe, Lorenzo Di Giandomenico, Mathis Goussard

Table of contents

Project Description

The **AIOps Autonomous System** aims to guarantee the reliability, availability and performance of distributed microservices architectures through autonomous self-healing and auto-scaling mechanisms.

Key objectives include the real-time detection of performance anomalies (such as CPU saturation or high latency), the autonomous execution of remediation actions (Scaling Up/Down, Restarting services) and the provision of clear, AI-generated explanations for every automated decision.

Goals of the System

The primary goal of the AIOps Autonomous System is to minimize system downtime and maximize service quality (QoS) through autonomous decision-making.

Key objectives include:

- **Proactive Anomaly Management:** Identify and address performance bottlenecks before they result in service outages.
- **Autonomous Remediation:** Efficiently execute scaling and healing actions (e.g., *Scale Up* on high load, *Restart* on critical failure) without human intervention.
- **Explainable Operations:** Provide a user-friendly interface that not only visualizes metrics but also explains the *rationale* behind every automated action using a Large Language Model (LLM).

Functional Requirements

Identifier	Name	Description
AIOPS-FR001	**Autonomous Remediation**	The system must be capable of automatically determining and executing corrective actions (Scale Up, Scale Down, Restart) based on detected anomalies.

Identifier	Name	Description
AIOPS-FR002	**Telemetry Integration**	The system must ingest real-time telemetry data from managed resources, including CPU usage, Memory usage, Service Time and active Instances.
AIOPS-FR003	**Anomaly Detection**	The system must analyze incoming metrics against defined thresholds to identify "Warning" (potential issue) states, "Critical" (failure) states , “Low Usage” states or “Normal” states.
AIOPS-FR004	**Generative Explainability**	The system must leverage a Generative AI model (LLM) to generate concise, human-readable explanations for every remediation plan proposed.
AIOPS-FR005	**Operations Dashboard**	Provide a real-time web interface for operators to monitor container metrics, view system topology and read the AI-generated remediation logs.

Non Functional Requirements

Identifier	Name	Description
AIOPS-NFR001	**Responsiveness & Latency**	The system should be able to detect an anomaly and trigger the corresponding remediation action in near real-time (within the configured analysis interval, e.g., < 10 seconds).
AIOPS-NFR002	**Scalability**	The system architecture (based on MQTT and Async Queues) should support an increasing number of

Identifier	Name	Description
		monitored containers, clusters and monitored metrics without performance degradation.
AIOPS-NFR003	**Usability & Clarity**	The dashboard should provide clear visualizations (color-coded values and thresholds) and the AI explanations must be concise, to allow operators to quickly understand the system state.
AIOPS-NFR004	**Resource Efficiency**	The AI components (LLM Service) must run efficiently on limited hardware, utilizing quantization and optimized inference parameters (e.g., reduced context window) to minimize CPU/RAM footprint.
AIOPS-NFR005	**Security**	The system must allow only authenticated MQTT communications and InfluxDB connections, ensuring security by restricting access to authorized users and protecting stored passwords through encryption.
AIOPS-NFR006	**Portability**	The system must ensure high portability to allow seamless deployment across different enviroments. This is achieved by utilizing Docker to containerize every microservice within the architecture.

Managed resources

The system manages a set of containerized services organized into logical clusters, simulating a Kubernetes environment.

- **Kubernetes Clusters:**

Logical groupings of containers (defined as Cluster 0, Cluster 1, etc... in the config.ini file). These represent distinct availability zones or functional groups within the infrastructure.

- **Docker Containers (Microservices):**

Specific instances of web services. The default configuration includes critical e-commerce components such as:

- auth-service (Authentication).
- payment-service (Transactions).
- cart-service (User session management).
- recommendation-service (AI-driven suggestions).

Sensors and Effectors

Both sensors and effectors functions are defined in the `managed_resources/webapp.py` file, and utilized in the `managed_resources/data_generator.py` file.

Sensors:

Simulates sensors for each container, generating metrics data (such as CPU usage, Memory usage, etc...) as specified in the config.ini file.

Effectors:

Provides functions to actively actuate the actions published on the executor MQTT topic on the corresponding containers.

Architectural Pattern

Self-Adaptation

The AIOps Autonomous System implements a self-adaptive logic designed to maintain system stability and performance in response to changing internal and external conditions.

Reason

Change in the Context (Workload Fluctuations): Adaptation is triggered when the monitoring system detects a deviation in the telemetry data (e.g., CPU usage spikes above 75%, Memory leaks or Service Time degradation).

Level

Application (Microservices) - Container Level: The adaptation actions target specific containers within the Kubernetes clusters. The system modifies the runtime parameters of these containers such as CPU usage, RAM usage, etc...

Time

Reactive: The system operates primarily on a reactive basis.
The *Analyzer* module evaluates the *current* state of the system against defined thresholds (e.g., "If CPU > 75% then Warning"). Actions are planned and executed *after* a violation is detected to return the system to a desired state.

Technique

Parameter: The system continuously monitors the resource consumption metrics (CPU, Memory, Service Time, etc...) of the managed containers.
It dynamically tunes the parameters to balance the workload: decreasing parameters such as CPU usage when metrics exceed safe thresholds (Scale Up) to degrade latency, and increasing them when resources are underutilized (Scale Down) to optimize efficiency. Additionally, it resets operational parameters to their initial state via restarts to heal critical failures.

Adaptation Control

This section discusses the architectural pattern employed for the autonomic manager, referring to the centralization vs. decentralization paradigm.

- **Approach:**

  External: The Adaptation Logic (Analyzer + Planner) is implemented as a distinct, external entity separate from the Managed Resources (Data Generator). They communicate asynchronously via a message bus (MQTT), decoupling the management logic from the application logic.

- **Adaptation Decision Criteria:**

  Rule-based: Adaptation is driven by thresholds defined in the config.ini file (e.g., maintaining Service Time < 300ms).

- **Degree of Decentralization:**

  Centralized: The system employs a **Centralized Adaptation Logic**.
  A single autonomic manager instance (composed of the *Analyzer* and *Planner*) is responsible for monitoring the context of the entire infrastructure (all clusters and all containers) and controlling the adaptation.

Adaptation Goals

Goal	Description	Formal Representation / Logic
Service Availability	**Recover from critical failures.**	**Severity(t) == Critical => Action(t) = Restart**
Performance Assurance	**Maintain Service Level Agreements (SLAs).**	**Severity(t) == Warning => Action(t) = ScaleUp**
Resource Efficiency	**Minimize infrastructure waste.**	**Severity(t) == UnderUsage => Action(t) = ScaleDown**

Monitored and Analyzed Characteristics

To support these goals, the system utilizes the following characteristics:

1. **Monitored Characteristics (Raw Data):**
    - **CPU Usage (%):** Immediate indicator of computational load.
    - **Memory Usage (MB):** Indicator of the application footprint and potential leaks.
    - **Service Time (ms):** Key performance indicator for user experience (latency).
    - **Instance Count:** Current deployment size.
2. **Analyzed Characteristics (Induced Reasoning):**
    - **Severity State:** The *Analyzer* computes a derived state for each container by comparing raw metrics against dynamic thresholds defined in the config.ini file.
        - **Critical:** Value > Threshold.
        - **Warning:** Value > Threshold * 0.6.
        - **Under Usage:** Value > Threshold * 0.2.
        - **Normal:** All other cases (Dead Zone).

All of the above characteristics (including monitored metrics) can be modified in the config.ini file.

Decision Function

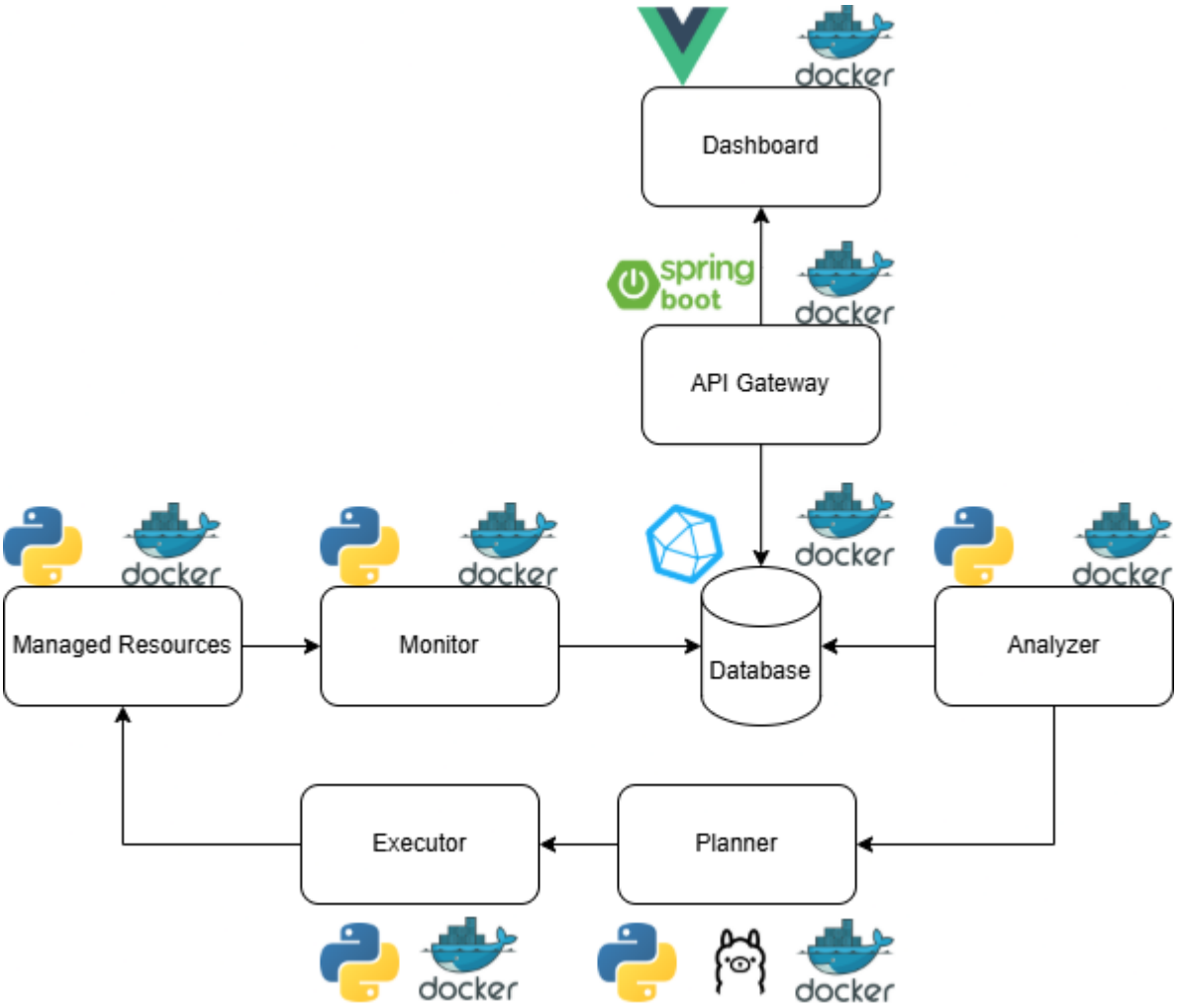
The decision function of the autonomic manager is based on a Hybrid Approach that combines **Rule-Based logic** for execution safety with **Generative AI** for reasoning and interpretability.

Rule-Based Component (Deterministic Execution):
The core remediation strategy relies on a deterministic Event-Condition-Action model. This ensures that critical operations (such as restarting a service or scaling the infrastructure) occur with sub-second latency and guaranteed predictability.

- Condition: The *Analyzer* evaluates the induced severity state (Warning, Critical, Under Usage) derived from the dynamic thresholds defined in the config.ini file.
- Action: The *Planner* maps these states to specific remediation commands using a strict look-up logic (e.g., IF Critical THEN Restart).

AI-Based Component (Generative Reasoning):
While the execution is rule-based, the system employs a Large Language Model (LLM) running via Ollama and set up in the .env file to process the decision context.

System Architecture



1. Managed Resources (Simulation Layer)

The Managed Resources represent the target software system that the AIOps framework monitors and adapts. In this project, this layer simulates a Kubernetes environment hosting distributed microservices.

- **Components:**
    - **webapp.py (Data Model):** Defines the physics of the simulation.
        - **Container Class:** Represents a microservice instance. It holds the internal state (metrics) and defines the `tick()` method, which applies stochastic noise (random variation) to the metrics based on the config.ini file limits (min, max, noise). It also implements the logic for state transitions (e.g., Restart resets metrics to their initial values).

        - **Cluster Class:** Represents a logical grouping of containers. It acts as an aggregator for state updates and executes received actions (execute_action) by applying specific deltas (e.g., scale_up_delta) to the target container's metrics.
    - **data_generator.py (Simulation Engine):** The main execution loop.
        - **Initialization:** Loads the system topology and metrics physics from the config.ini file.
        - **Data Generation:** Runs an infinite loop that triggers the `tick()` update for every container and publishes the resulting metrics to MQTT topics (*AIops/metrics/...*) (acting as **Sensor**).
        - **Command Execution:** Listens to the *AIops/execute* topic (acting as the **Effector**). When a command is received (e.g., Scale Up) it modifies the container state immediately and pushes an update to the monitoring system to reflect the change.

2. Monitor (Telemetry Ingestion)

The Monitor component acts as the bridge between the volatile message bus and the persistent storage.

- **Technology: Telegraf**.
- **Role:**
    - Subscribes to the MQTT wildcard topic *AIops/metrics/#*.
    - Parses the incoming JSON telemetry data from the Managed Resources.
    - Forwards and writes this data into the InfluxDB time-series database.

3. Database (Knowledge Base)

- **Technology: InfluxDB**.
- **Role:** Serves as the **Knowledge Base** in the MAPE-K loop.
- **Functionality:** Stores historical time-series data for all the monitored metrics (CPU, Memory, Service Time, etc...). This persistence allows the Analyzer to query multiple metrics simultaneously to identify overall trends in the whole container.

4. Analyzer (Anomaly Detection)

The Analyzer is responsible for evaluating the health of the system by processing historical data against defined rules.

- **Logic (Polling Loop):**
    - Connects to InfluxDB and performs periodic queries (defined by the ANALYZER_INTERVAL variable in the config.ini file) to fetch the latest state of all containers.
- **Anomaly Detection Algorithm (evaluate_metrics):**
    - Iterates through every container and compares current metrics values against thresholds defined in the config.ini file.
    - **Severity Hierarchy:** Implements a priority logic to determine the single "Dominant Metric" for a container.
        1. **Critical:** Value > Threshold (Highest Priority).
        2. **Warning:** Value > 60% of Threshold.
        3. **Under Usage:** Value < 20% of Threshold.
        4. **Normal:** Dead Zone (No action required).
- **Output:** Aggregates findings into an "Anomaly Report" and publishes it to the *AIops/analyzer* topic, triggering the planning phase only if anomalies are detected.

5. Planner (Decision & Reasoning)

The Planner serves as the "Brain" of the autonomic manager, combining deterministic logic with AI-driven explainability.

- **Decision Function (decide_action):**
    - Subscribes to *AIops/analyzer*.
    - Uses a **Rule-Based** approach to map severity states to remediation actions:
        - **Critical** => `restart`
        - **Warning** => `scale_up`
        - **Under Usage** => `scale_down`
- **GenAI Integration:**
    - Before finalizing the plan, the Planner sends the proposed actions to the **LLM Service** (Ollama).
    - It requests a natural language explanation justifying *why* the action is necessary based on the specific metrics context.

- **Output:** Publishes a comprehensive "Remediation Plan" (containing the Action, Target and LLM Explanation) to the *AIops/planner* MQTT topic.

6. Executor (Action Enforcement)

The Executor acts as the "Effector" mechanism, translating high-level plans into specific commands understandable by the Managed Resources.

- **Logic:**
    - Subscribes to the *AIops/planner* topic.
    - **Normalization:** Parses the target identifiers (e.g., converting container_auth-service to auth-service and cluster_0 to Integer 0) to match the internal data model of the Data Generator.
    - **Command Dispatch:** Encapsulates the clean data into a command JSON object and publishes it to the *AIops/execute* topic.
- **Role in Loop:** This component closes the MAPE-K loop, ensuring that the decision made by the Planner is physically applied to the simulation layer before generating new data.

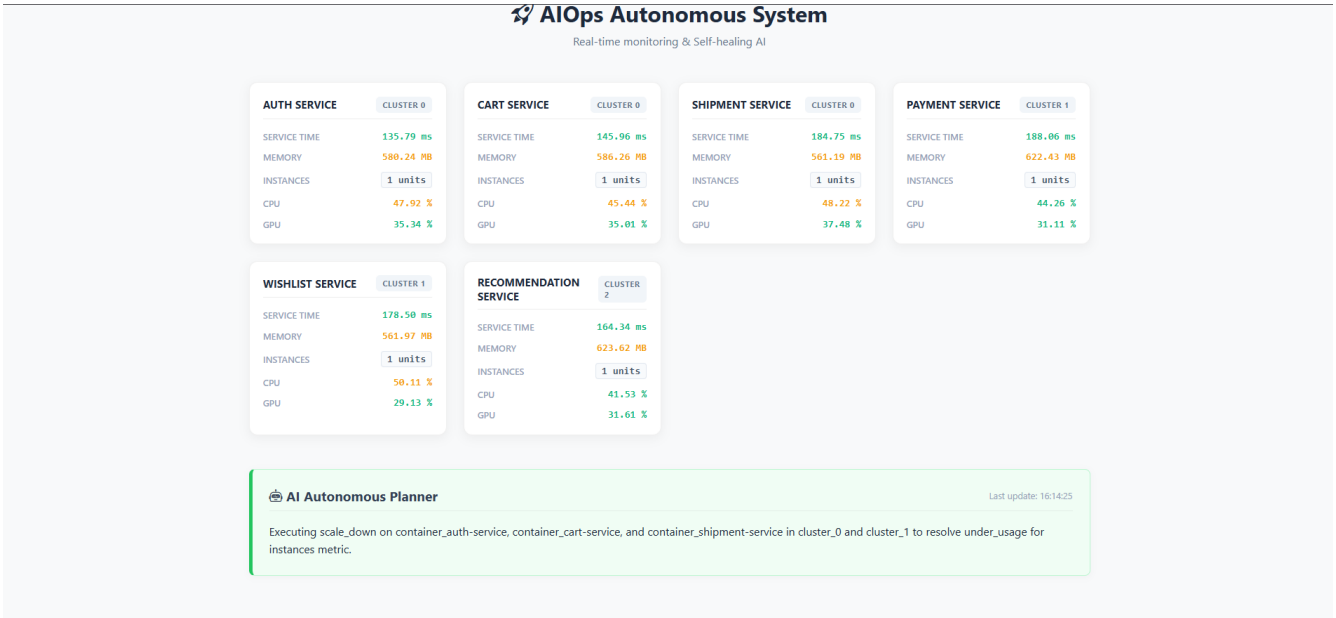
7. API Gateway

- **Technology:** Java Spring Boot.
- **Role:** Acts as the backend interface for the user interface.
- **Functionality:**
    - Exposes RESTful endpoints to the Dashboard.
    - Queries InfluxDB to retrieve historical metric data for visualization.
    - Acts as a **secure middleware**, preventing the frontend from directly accessing the database or the message broker.

8. Dashboard (User Interface)

- **Technology:** Web-based Frontend (Vue.js).
- **Role:** Provides observability and transparency into the autonomic process.
- **Features:**
    - **Telemetry Visualization:** Renders real-time cards of CPU, Memory and Service Time for each container.

- **LLM Log:** Displays the "Chat" interface where the LLM explanations for every automated action are shown, allowing operators to understand the rationale behind the system's self-healing behavior.



Conclusion

In conclusion, this project has successfully implemented a robust **Self-Adaptive System** based on the MAPE-K reference model, capable of autonomously managing a simulated microservices architecture. By orchestrating a complete feedback loop, from real-time telemetry monitoring to the execution of scaling and healing actions, the system effectively guarantees service reliability and performance stability under dynamic workloads.

Furthermore, the integration of a **Generative AI** module represents a significant advancement over traditional rule-based automation. By coupling the deterministic decision engine with an LLM-based reasoning layer, the system transcends simple automation to become an **Explainable AIOps** solution, offering operators not just a reactive dashboard but also a transparent log of *why* specific remediation actions were taken. All of this, combined with a responsive **Backend and User Interface**, ensures full observability and control, demonstrating the practical viability of AI-driven autonomy in modern distributed systems.