SERVER HEALTH AND MANAGEMENT SYSTEM

by

YUSUF DEMİR - 150120032

YUSUF DUMAN - 150120023

CSE4197 Engineering Project 1

Analysis and Design Document

Supervised by:

Assoc. Prof. ÖMER KORÇAK

Marmara University, Faculty of Engineering

Computer Engineering Department

14.06.2025

Contents

1 INTRODUCTION

1.1 Problem Description and Motivation

In today's digital era, the internet forms the backbone of modern communication, commerce, and infrastructure. At the core of this ecosystem are servers, which act as centralized systems responsible for processing requests, storing data, and delivering information to client devices such as smartphones, laptops, IoT devices, and smartwatches. Servers are typically expected to operate continuously and without failure, as even brief periods of downtime can result in significant consequences.

Unplanned server outages can cause service interruptions, financial losses, reputational damage, and even introduce security risks. According to the ITIC 2024 Hourly Cost of Downtime Report, 90% of enterprises lose over $300,000 per hour of downtime, while 41% report losses between $1 million and $5 million per hour [1]. These statistics highlight the critical need for early detection, diagnosis, and resolution of server issues.

The motivation behind this project is to develop a proactive and lightweight server monitoring and management solution that helps reduce downtime and enhance system reliability. The system will track key performance indicators including CPU, RAM, disk, and network usage, in addition to service/process health and backup status. In the event of anomalies (such as resource overutilization, service failures, or missed backups) the system will alert the administrator in real-time through the web interface or other configured channels (e.g., Telegram).

Furthermore, the platform will provide administrators with tools to analyze logs, restart failing services, and monitor system health through a responsive and secure web interface. This capability will empower system administrators to act swiftly and maintain operational stability with minimal manual intervention.

1.2 Scope of the Project

This project aims to deliver a centralized, lightweight, and extensible server monitoring and management system designed specifically for Linux-based infrastructures. The core of the system is a central backend server, which coordinates all logic, data storage, and decision-making processes. Individual monitored servers (often distributed across various environments) run a minimal Python-based agent that collects relevant data and forwards it to the backend. By offloading processing to the central server, the project ensures that monitored systems remain unaffected in terms of performance.

The primary functionality of the system includes real-time monitoring of key server metrics, such as CPU, RAM, disk, and network usage. Network data is collected on a per-interface basis, including metrics like IP address, MAC address, and upload/download bandwidth. This information is visualized through an interactive dashboard in the web interface, allowing administrators to track system health at a glance.

Beyond resource metrics, the system also includes comprehensive service and process monitoring. It tracks the health of predefined critical services such as NGINX, MySQL, and SSH and includes an automatic recovery mechanism. In case of a service failure, the agent attempts to restart the service up to a configured number of times. If these attempts are unsuccessful, the backend pushes an alert to the system administrator. Additionally, the system displays all running processes on each monitored server along with their corresponding CPU and memory usage, providing deep insight into system activity.

All monitoring data is transmitted over a WebSocket (WSS) connection using token-based authentication during the initial handshake. This ensures secure and persistent communication between the agent and backend while allowing near real-time updates. Once received, data is stored immediately in a relational database by the backend, which then handles further tasks such as threshold evaluation, alert generation, and frontend data provisioning.

An integral component of the system is the backup management module, which allows administrators to configure directory-level backups via the web interface. Users can specify source directories, backup frequencies (such as hourly or daily), and remote server credentials. After configuration, backups are performed automatically using secure transfer protocol Rsync over SSH. The outcome of each backup (whether successful or failed) is logged and made available to the administrator through the web UI, ensuring transparency and traceability.

The project scope explicitly focuses on Linux-based environments (e.g., Ubuntu or Debian). Other operating systems, such as Windows Server, are not supported. Moreover, full system imaging and disaster recovery solutions are considered out of scope. The system also avoids complexity related to user-role hierarchies, instead assuming a single-administrator model for simplicity and security. While the web interface is responsive, the current implementation targets desktop users, and mobile applications are not included in the project's current scope.

In summary, the project delivers a functional and scalable solution that brings together real-time metric collection, intelligent alerting, service and process monitoring, and backup automation within a secure and centralized architecture. This makes it particularly well-suited for small to medium-sized businesses or self-hosted environments that require dependable server oversight without the complexity or overhead of enterprise-level solutions.

1.3 Definitions, Acronyms, and Abbreviations

Term / Acronym	**Definition**
Server Agent	A lightweight Python-based software component that runs on each monitored server. It collects system metrics, monitors processes and services, and communicates with the central backend. It does not make decisions but executes instructions received from the backend.
Backend	The central component of the system, responsible for data aggregation, decision-making, alert handling, storage, and communicating with both the server agents and frontend. Deployed as a web server with API and database functionality.
Frontend	A web-based user interface built using modern JavaScript technologies (e.g., React.js). It allows administrators to visualize metrics, configure backups, and manage alerts.

WebSocket Secure (WSS)	A secure version of the WebSocket protocol (over TLS). Used for persistent, bidirectional communication between the server agents and the backend.
HTTPS	HyperText Transfer Protocol Secure. A secure version of HTTP used for encrypted communication between the frontend and the backend.
Authentication	The process of verifying the identity of a user or component. In this project, token-based authentication is used during the initialization of WebSocket connections.
Authentication Token	A cryptographic token (e.g., JWT) issued by the backend and used by agents or frontend clients to authenticate requests securely.
Rsync	A utility for efficiently transferring and synchronizing files across systems. It is used in this project for performing scheduled backups to remote servers.
Service	A background application or daemon running on the server, typically managed via systemd. Examples include NGINX, MySQL, and SSH. The system monitors these services for uptime and restarts them if failures are detected.
Process	An instance of a running program on the operating system. The project tracks resource usage (CPU, memory) of all running processes.
JWT	JSON Web Token — a compact, URL-safe token format used for authentication.
CLI	Command Line Interface — the method through which the system is deployed and configured. No GUI installation is supported.
UI/UX	User Interface / User Experience — the design principles applied in building the frontend.

Table 1: Definitions, Acronyms, and Abbreviations

2 LITERATURE SURVEY

Server monitoring and management have long been critical areas in systems administration, with several mature tools available in the open-source ecosystem. Among the most widely used solutions are Nagios, Zabbix, Prometheus (typically paired with Grafana), and Netdata. Each of these tools offers a distinct approach to monitoring infrastructure, but they often introduce trade-offs in terms of complexity, performance overhead, ease of use, and extensibility—particularly for small to mid-sized or self-hosted deployments.

- **Nagios** is one of the oldest and most modular monitoring tools. It operates on a plugin-based architecture where users define checks manually in configuration files. Nagios is flexible and reliable but comes with a steep learning curve. It lacks modern UI features out-of-the-box and requires substantial effort to scale across multiple servers or integrate with third-party visual dashboards. Managing service recovery and automation in Nagios typically requires writing custom scripts or using add-ons [2].
- **Zabbix** offers a more modern and integrated system compared to Nagios. It includes a web-based dashboard, built-in visualization tools, and a more structured approach to host and service monitoring. Zabbix supports agent-based and agentless data collection, offers triggers for alerts, and can monitor a wide range of metrics. However, it requires a full-stack deployment involving a database, server, frontend, and agent components. It can be resource-intensive and may be overkill for lightweight use cases or teams without dedicated IT infrastructure [3].
- **Prometheus** is a powerful tool designed around time-series data and metric-based monitoring. It is highly scalable and optimized for cloud-native environments, where containerized workloads generate large volumes of metrics. Prometheus uses its own query language (PromQL) and is often used alongside **Grafana** for visualization. While it excels in metric collection and alerting at scale, it does not provide native service monitoring, restart mechanisms, or multi-server aggregation without custom exporters and integrations. This makes initial setup and feature extension a non-trivial task for less experienced users [4].

- **Netdata** is an easy-to-deploy monitoring solution that focuses on real-time visualization. It provides detailed charts and graphs with very low setup requirements and near-zero configuration. While it is highly effective for quickly identifying performance bottlenecks on a single server, its open-source version does not support centralized control or alert automation across multiple nodes. Advanced features like team-based access or unified management are reserved for its commercial cloud platform [5].

These tools demonstrate the breadth and maturity of the server monitoring landscape but also reveal gaps in simplicity, integration, and targeted functionality. Most focus heavily on metrics but do not include practical features such as automated service recovery, directory-level backup scheduling, or a unified agent–backend–frontend design optimized for low-resource environments.

Our project aims to address these gaps by combining essential monitoring features with service control, process tracking, and backup management, all delivered through a clean and responsive web interface. The system prioritizes ease of deployment, low resource usage, and administrator convenience—attributes often missing from traditional, enterprise-oriented solutions.

Feature / Tool	**Nagios**	**Zabbix**	**Prometheus + Grafana**	**Netdata**	**Proposed System**
Ease of Setup	❌ Manual, high config burden	⚠️ Moderate	⚠️ Complex (PromQL, exporters)	✅ Easy	✅ Simple 3-component architecture
Resource Usage	⚠️ Moderate to high	⚠️ Moderate	⚠️ Moderate	✅ Low	✅ Designed for low-resource environments
Service Monitoring	✅ Yes (via plugins)	✅ Yes	⚠️ Limited with third-party	⚠️ Basic	✅ Yes, Built-in with auto-restart logic
Backup Management	❌ Not included	❌ Not included	❌ Not included	❌ Not included	✅ Included with scheduler
User friendly UI	❌ Minimal, not intuitive	⚠️ Steep learning curve	⚠️Flexible but complex	✅ Simple	✅ Simple and clean dashboard
Target Users	Enterprises, experienced admins	SMEs to large enterprises	Large-scale infrastructure	Hobbyists, SMEs	SMEs, developers, self-hosted users
License/Cost	Open-source, free	Open-source, free	Open-source (Grafana optional)	Open-source (Cloud paid)	MIT License (open-source)

Table 2: Overall Comparison

3 PROJECT REQUIREMENTS

This section outlines the functional and non-functional requirements of the server monitoring and management system. These requirements ensure the system fulfills its intended purpose within defined constraints, and each requirement is clearly specified for verifiability and implementation.

3.1 Functional Requirements

3.1.1 Real-Time System Monitoring

- **Description:** The system must monitor CPU, RAM, disk, and network metrics of each server in real time.
- **Inputs:** System metrics collected by the server agent.
- **Processing:** The agent collects and sends metrics over a WebSocket connection to the backend, which stores and evaluates data for anomalies.
- **Outputs:** Updated visualizations in the web UI and potential alerts.
- **Error/Data Handling:** If the connection drops or data is malformed, an error log is generated and the system attempts to re-establish the connection.

3.1.2 Service Monitoring and Recovery

- **Description:** The system must monitor predefined services and attempt automatic recovery on failure.
- **Inputs:** Service status data from the agent.
- **Processing:** The agent sends service data to backend and agent attempts restart with the backend’s request. If unsuccessful, the backend logs the event and sends an alert.
- **Outputs:** Service recovery logs and alerts in the UI.
- **Error/Data Handling**: If restart fails after configured attempts, alert is escalated to the administrator.

3.1.3 Backup Management

- **Description:** Administrators should be able to configure and manage directory-level backups via the web interface.
- **Inputs:** User input from the web UI including source directory, schedule, and destination.
- **Processing:** The backend schedules backup tasks using Rsync over SSH based on user configuration.
- **Outputs:** Backup execution logs and success/failure indicators.
- **Error/Data Handling:** Failed backups trigger alert notifications and are logged.

3.1.4 Alerting System

- **Description:** The system should generate alerts when abnormal system behavior is detected.
- **Inputs:** Metric thresholds and failure events.
- **Processing:** The backend compares incoming data against defined thresholds and service states.
- **Outputs:** Alert messages displayed in UI and optionally sent via external channels (e.g., Telegram).
- **Error/Data Handling:** If alert delivery fails, retry mechanism is triggered and fallback logging is enabled.

3.1.5 Web Interface Management

- **Description:** A web-based UI must allow administrators to monitor metrics, manage services, configure backups, and view logs.
- **Inputs:** User interactions via web browser.
- **Processing:** The frontend sends requests to backend APIs to fetch, modify, or visualize system data.
- **Outputs:** Visual dashboards, configuration confirmation, and system logs.

- **Error/Data Handling:** Form validations, error messages, and fallback pages in case of API failures.

3.2 Nonfunctional Requirements

3.2.1 Performance:

- The system must handle metrics from at least 10 agents concurrently.
- Response time for UI updates should be less than 1 second under normal load.

3.2.2 Reliability:

- System uptime must be at least 99%.
- Mean time to recovery from failure: < 5 minutes.

3.2.3 Usability:

- UI must be responsive and accessible via modern browsers.
- Tooltips and error messages must be provided for all major functionalities.

3.2.4 Security:

- All data transmissions must use encrypted channels (WSS, HTTPS).
- Authentication via token-based system; no anonymous access.

3.2.5 Maintainability:

- Codebase must follow modular structure.
- Logs should capture system events to assist with debugging.

3.2.6 Portability:

- The agent must run on Ubuntu-based systems.

4 SYSTEM DESIGN

4.1 UML Use case Diagrams for the main use cases

Use Case 1: Add New Server and Monitor

This diagram represents the interactions between the **User**, **Backend**, and a newly added **Server Agent**.

- **Register Server**: The admin inputs server details (hostname, token, etc.) on the frontend to start monitoring a new server.
- **Initialize Agent**: The agent on the new server starts and authenticates with the backend using a token.
- **Backend-Frontend Sync**: Backend registers the agent and begins receiving metrics. The new server appears on the frontend dashboard.
- **Start Monitoring**: The server agent starts sending system and service metrics at regular intervals.
- **Visualize Data**: The user begins to see real-time metrics and status updates for the new server on the frontend.

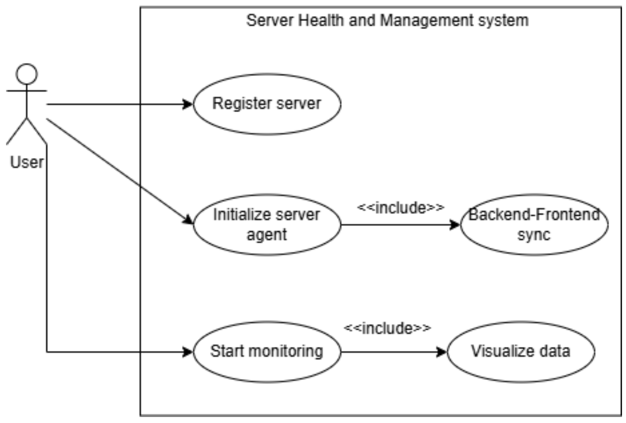


Figure 1: Use case 1

Use Case 2: Restart a Service

This diagram shows how the **Admin User** interacts with the system to restart a failed or misbehaving service.

- **Trigger Restart**: The admin selects a service and issues a "Restart" command through the frontend.
- **Send Command to Agent**: The backend receives the command and forwards it to the correct server agent over WSS.
- **Execute Command (Agent)**: The agent restarts the service on the target machine.
- **Report Result**: The agent sends the success/failure response back to the backend.
- **Frontend Confirmation**: The backend updates the frontend with the result, and the admin is notified.

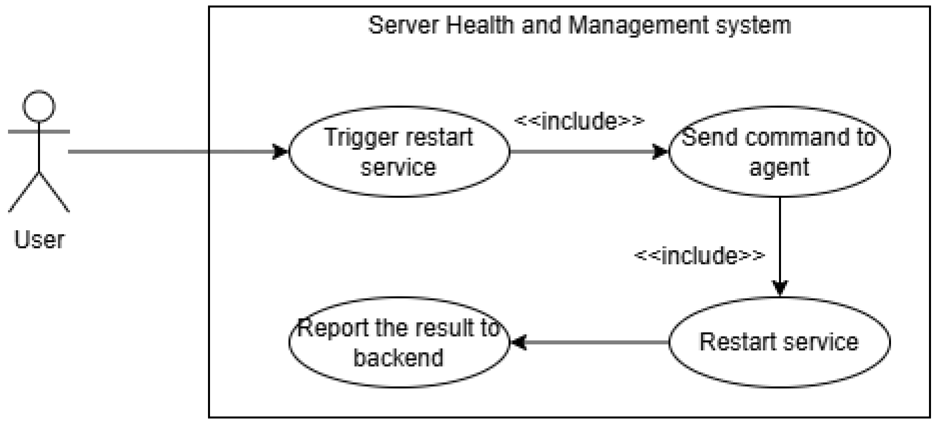


Figure 2: Use case 2

Use Case 3: Metric Threshold & Alert

This diagram outlines the process of setting an alert threshold and receiving a notification when it is exceeded.

- **Set Threshold**: The admin defines a new threshold rule (e.g., CPU > 85%) through the frontend.

- **Backend Stores Rule**: The rule is stored in the database and becomes active.
- **Monitor Metrics**: The backend continuously evaluates incoming metrics against defined thresholds.
- **Trigger Alert**: When the metric exceeds the threshold, an alert is triggered by the backend.
- **Notify User**: The alert is shown on the frontend and optionally sent via other channels (e.g., email, Slack).

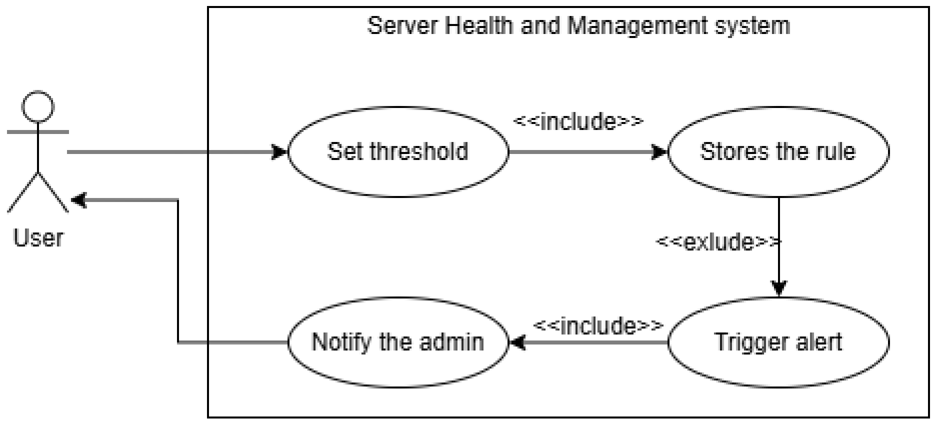


Figure 3: Use case 3

Use Case 4: Schedule Backup

This diagram illustrates how the **Admin User** schedules backups that the **Agent** performs.

- **Define Backup Task**: The user specifies backup time, path, and target (e.g., backup server) through the frontend.
- **Store Task in Backend**: The backend schedules the task and queues the instruction.
- **Dispatch to Agent**: At the scheduled time, the backend sends the backup command to the agent.

- **Execute Backup**: The agent performs the backup operation and reports back status.
- **Log & Notify**: Backend stores the result and optionally notifies the admin via the frontend.

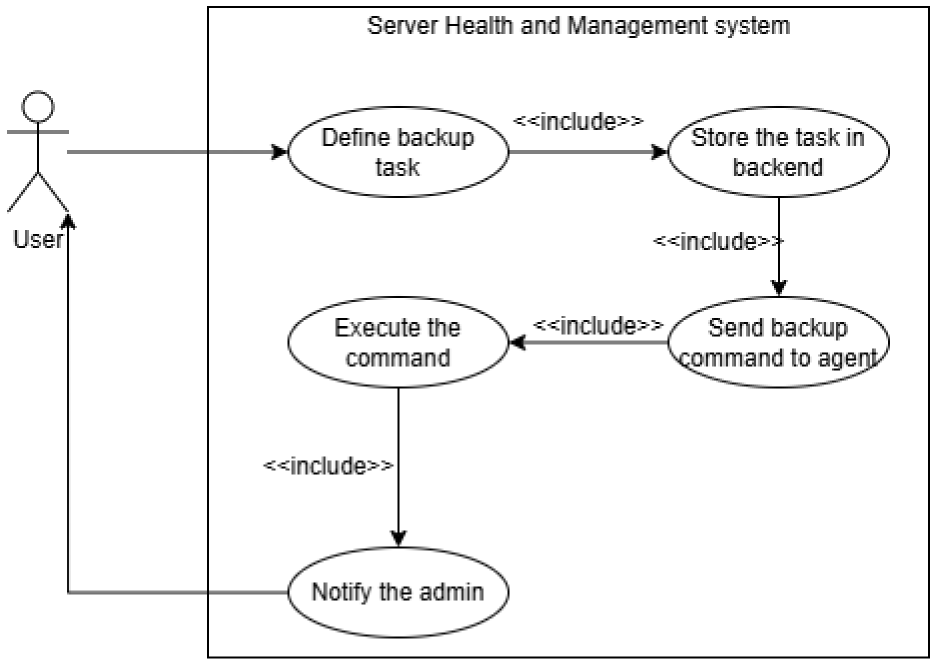


Figure 4: Use case 4

4.2 Database ER diagram

This section presents the ER (Entity-Relationship) diagram representing the data models of our project.

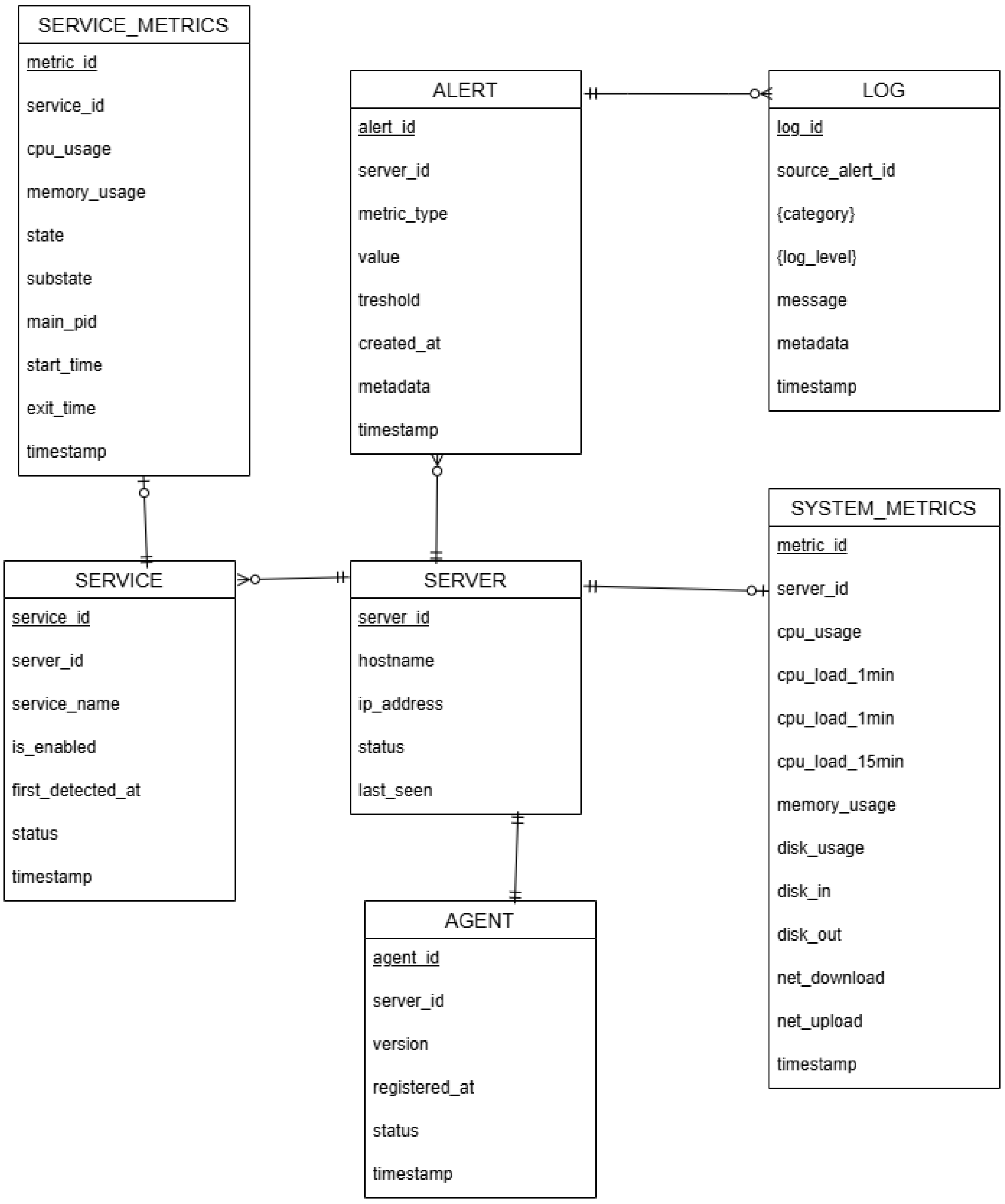


Figure 5: ER Diagram

4.3 User Interface (Preliminary version)

4.3.1 Login panel

- **E-mail Input Field**: Accepts user email with basic validation.
- **Password Input Field**: Accepts user password.
- **Remember Me**: Checkbox to keep users logged in on the device.
- **Forgot Password**: Link redirects to password recovery page.
- **Login Button**: Submits credentials to authenticate the user.

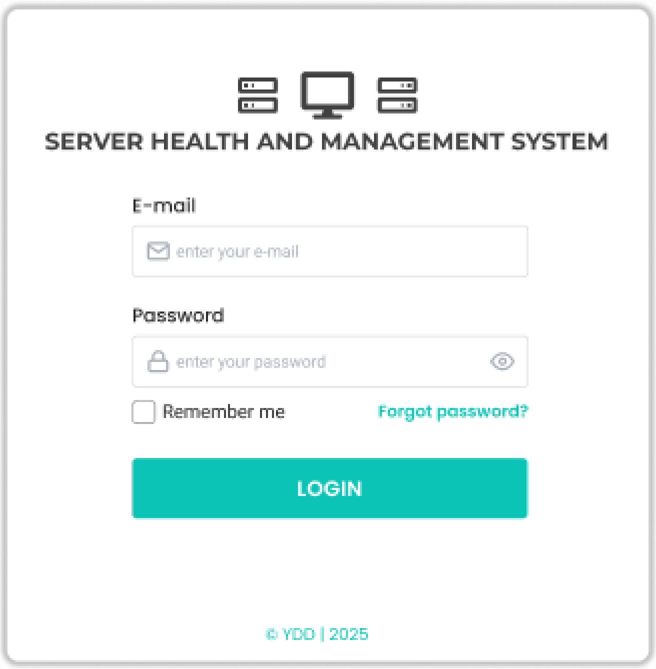


Figure 6: Login panel

4.3.2 Data Visualization Components

- **CPU Usage Gauge**: Displays real-time CPU load as a percentage using a gauge.
- **CPU Load History Bar Chart**: Shows normalized CPU load over 1, 5, and 15-minute intervals.
- **RAM Usage Gauge**: Displays current RAM usage in percentage and absolute values (used/total).
- **Disk I/O Bar**: Visualizes disk read (in) and write (out) speeds in MB/s.
- **Internet Speed Gauges**: Show current download and upload speeds in Mbps with separate gauges.
- **History Line Chart**: Displays a time-series graph of data. It will be used for most of the metrics that have historical data.

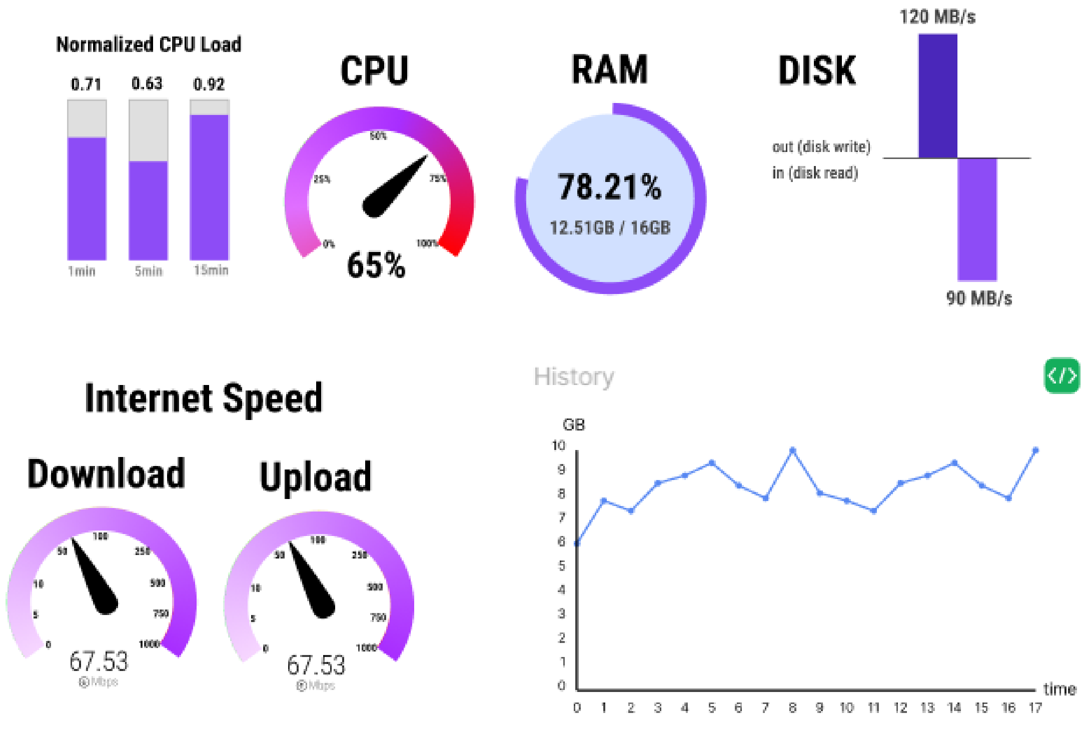
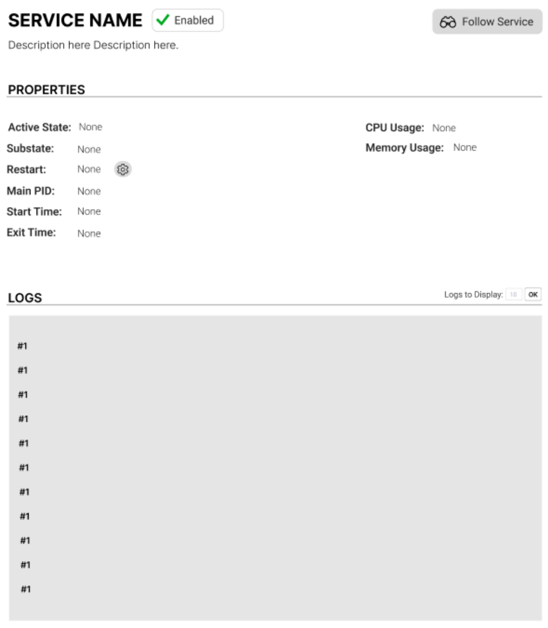


Figure 7: Data visualization components

4.3.3 Service Page

- **Service Header:**
  - Displays service name and description.
  - Shows current status.
  - Button for subscribing for status/log updates for the spesific service.
- **Properties Panel:**
  - Displays key service metrics.
- **Restart Button:** Allows manual restart of the selected service.
- **Logs Section:**
  - Shows logs for the specific service.
  - Allows user to select how many logs to display via a dropdown


SERVICE NAME ✓ Enabled

Description here Description here.

PROPERTIES

Active State: None

Substate: None

Restart: None

Main PID: None

Start Time: None

Exit Time: None

CPU Usage: None

Memory Usage: None

LOGS

Logs to Display: 10

```
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
```

Figure 8: Service page

4.4 Test Plan

The testing approach follows a modular and integration-based strategy, covering the agent, backend, and frontend components of the system. Core functionalities will be tested locally, and distributed behavior will be tested using multiple VPS servers located in different geographic regions. Both manual and automated methods will be used, depending on the component.

- **Backend** testing focuses on API correctness, alert logic, and data processing. Unit tests will be implemented using PyTest (for Python) or Jest (for Node.js). WebSocket communication will be tested with real agents. The backend will be deployed on a VPS to evaluate its performance with remote agents under varying network conditions.
- **Agent** testing validates metric collection and transmission stability. Initially, system resource collection functions will be tested locally. Then, agents will be deployed to Ubuntu-based VPS servers across US, Europe, and Asia to simulate real-world usage. Stress scenarios will be applied by increasing metric update frequency, and error handling will be verified through intentional network disruptions.

- **Frontend** testing ensures UI functionality, responsiveness, and live data rendering. Manual exploratory testing will cover workflows like command execution and alert display. Compatibility will be checked across modern browsers (Chrome, Firefox). React Testing Library may be used for basic component tests.
- The test environment includes Ubuntu VPS servers, a PostgreSQL test database, and backend instances hosted on cloud infrastructure. Development and testing tools include Postman, PyTest or Jest, and browser dev tools.

Estimated Timeline:

- **Week 1**: Local agent and backend tests, API testing and UI checks
- **Week 2**: Distributed testing on VPS (agent–backend communication)
- **Week 3**: Full system integration, UI polish

5 SOFTWARE ARCHITECTURE

The software architecture of the server monitoring and management system is based on a modular client-server design that separates data collection, decision-making, and user interaction. It consists of three main components:

- **Server Agent** (client)
- **Backend Server**
- **Frontend (Web UI)**

Each component communicates via secure channels to ensure a scalable and maintainable system.

Component Overview:

1. **Server Agent (Python-based)**

   a. Collects system metrics
   b. Monitors processes and services
   c. Sends data to backend via secure WebSocket (WSS)
   d. Receives commands (e.g., service restart) from the backend

2. **Backend Server (Python + PostgreSQL)**
   a. Manages system logic and data aggregation
   b. Evaluates metric thresholds
   c. Stores monitoring data in the database
   d. Handles alert generation and notification delivery
   e. Exposes APIs for frontend communication

3. **Frontend (React.js)**
   a. Visualizes system metrics and status
   b. Provides admin controls (e.g., backup configuration)
   c. Displays logs, alerts, and historical trends
   d. Communicates with backend via HTTPS

Data Flow:

- Agents collect metrics → Send via WSS → Backend receives → Data stored in DB → Frontend fetches via REST API.

Control Flow:

- Backend issues commands (e.g., restart service) → Agent executes → Reports back.
- Admin configures backup in frontend → Backend updates agent instructions.

Modular Design:

- agent/: lightweight script, easily portable
- backend/api/: REST APIs for web interface
- backend/alerting/: threshold rules, notification logic
- frontend/components/: modular, reusable visual components

6 TASKS ACCOMPLISHED

6.1 Current state of the project

At this stage, several foundational parts of the system have been successfully completed, while others are underway. The current progress includes:

- **Research & Requirement Analysis:**
    - Completed literature review on existing monitoring tools (e.g., Prometheus, Zabbix).
    - Defined functional and non-functional requirements.
    - Selected technology stack: Python (Agent), Node.js/Python (Backend), PostgreSQL, React.js.
- **Architectural Design:**
    - Designed modular, client-server architecture.
    - Defined communication protocols between components (WSS, REST).
    - Documented module responsibilities and data/control flow.
- **Server Agent (Client):**
    - Implemented in Python.
    - Successfully collects core system metrics (CPU, RAM, Disk, Network).
    - Monitors critical services and packages data in JSON format.
    - Sends metrics to backend over secure WebSocket.
- **Backend (Partial Implementation):**
    - Basic server logic established using Flask (Python).
    - Capable of receiving and logging agent data.

    - Initial database schema created in PostgreSQL.
    - REST API endpoints under development.

Preliminary Results:

- The agent has been tested on local Ubuntu instances, confirming accurate metric collection.
- Real-time data transmission to the backend is functional.
- Sample data entries confirmed in the database.

6.2 Task Log

Meeting #1

- **Date:** 25 February 2025
- **Location:** In person (advisor’s office)
- **Period:** 1 hour
- **Attendees:** Yusuf Demir, Yusuf Duman, Assoc. Prof. Ömer Korçak
- **Objectives:** Presentation of the initial system architecture and project idea.
- **Decisions and Notes:** We presented a modular server monitoring system architecture that includes agents, a backend server, and a web-based frontend interface. During this phase, we discussed the overall feasibility of the project, identified potential target users, and defined the initial monitoring scope. Our advisor approved the general system structure and advised us to prioritize simplicity and modularity in the first iteration to ensure a manageable and scalable development process.

Meeting #2

- **Date:** 1 April 2025
- **Location:** In person (advisor’s office)

- **Period:** 1 hour
- **Attendees:** Yusuf Demir, Yusuf Duman, Assoc. Prof. Ömer Korçak
- **Objectives:** Advisor review and feedback on the Project Specification Document (PSD).
- **Decisions and Notes:** We received feedback specifically on our problem definition, the clarity of the project scope, and the articulation of objectives. The advisor emphasized the importance of referencing recent and relevant literature, and establishing concrete technical goals. It was also recommended that we refine the functional role of the server agent and eliminate vague or ambiguous language in the project's objectives section. As a result of this meeting, we agreed to revise the Related Works section and committed to finalizing the Project Specification Document (PSD) by mid-April.

Meeting #3

1. **Date:** 15 May 2025
2. **Location:** Online
3. **Period:** 1 hour
4. **Attendees:** Yusuf Demir, Yusuf Duman, Assoc. Prof. Ömer Korçak
5. **Objectives:** Review of encryption mechanisms and inter-component communication design.
6. **Decisions and Notes**: We agreed to remove the application-layer AES encryption mechanism, as it was deemed redundant in the presence of HTTPS. Instead, we finalized the decision to rely on HTTPS for secure communication between the frontend and backend. Furthermore, the advisor approved our proposed transition to WebSocket Secure (WSS) for communication between backend and agent, accompanied by a token-based handshake mechanism for secure connection initialization.

Meeting #4

- **Date:** 18 May 2025
- **Location:** Online
- **Period:** 1 hour
- **Attendees:** Yusuf Demir, Yusuf Duman, Assoc. Prof. Ömer Korçak
- **Objectives:** Implementation planning for backend-agent communication.
- **Decisions and Notes**: We finalized WSS as the standard channel for transmitting monitoring data from agents to the backend. Token-based authentication will be applied only during the initial connection setup to balance security with performance. In line with these updates, the system architecture diagrams were revised to reflect the newly adopted WSS protocol and communication structure.

6.3 Task Plan with Milestones

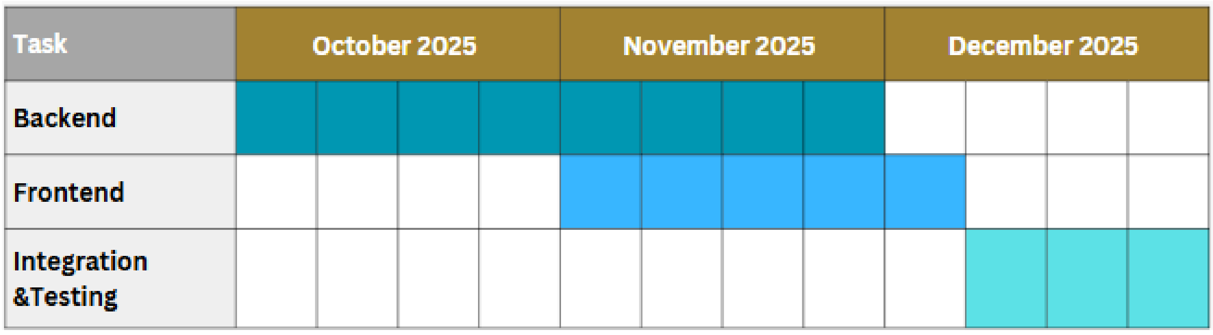
Task No	**Task Description**	**Expected Output**	**Months**					
			1	**2**	**3**	**4**	**5**	**6**
T1	Requirement Analysis	Detailed functional & non-functional requirements	+					
T2	Project Specification Document (PSD)	Approved PSD including scope, objectives, plan	+					
T3	System Design	Architectural diagrams and modular structure		+				
T4	Server Agent Development	Python agent collecting and sending metrics		+				

T5	Backend Development	API endpoints, database schema, alerting logic			+	+		
T6	Frontend Development	Web UI for monitoring, control, and visualization				+	+	
T7	Integration & Testing	Working full-stack prototype with test cases						+

Table 3: Task timeline

Timeline with Milestones

Milestone 1: Core System Development (Tasks 5, 6)
Milestone 2: System Integration & Testing Completed (Task 7)



Task	October 2025				November 2025				December 2025			
Backend	■	■	■	■	■	■	■	■				
Frontend					■	■	■	■	■			
Integration &Testing										■	■	■

Figure 9: task timeline gantt chart

7 REFERENCES

[1] ITIC. ITIC 2024 Hourly Cost of Downtime Report. Available at: https://itic-corp.com/itic-2024-hourly-cost-of-downtime-report/

[2] Nagios. Nagios Documentation. Available at: https://www.nagios.org/documentation/

[3] Zabbix. Zabbix Documentation. Available at: https://www.zabbix.com/documentation

[4] Prometheus. Prometheus Documentation. Available at: https://prometheus.io/docs/

[5] Netdata. Netdata Documentation. Available at: https://learn.netdata.cloud/docs