CLOUD NATIVE MONITORING APPLICATION

A Project Report

By

Rechael Lopes

DECLARATION

I, Rechael Vincent Lopes affirm that the attached work is entirely my own, except where the words or ideas of other writers are specifically acknowledged in accordance with accepted APA citation conventions. This project is specifically made for my personal use. I acknowledge that I have revised, edited, and proofread this paper, and I certify that I am the author of this paper. Any assistance I received in its preparation is fully and properly acknowledged and disclosed. I have also cited any sources from which I used data, ideas, theories, or words, whether quoted directly or paraphrased. I further acknowledge that this paper has been prepared by myself specifically for this project.

ABSTRACT

This document highlights the project plan for creating a Cloud Native Monitoring Application that is effective and efficient beyond just the basic applications. The plan covers a detailed Work Breakdown Structure with the technology used, workflow, and prototype. The designed project plan ensures the successful implementation of the application with reduced risks, downtime, and discrepancies.

Contents

CHAPTER 1: INTRODUCTION

Project Scope

The Cloud Native Monitoring Application is a full-stack solution aimed at real-time monitoring of cloud-native applications. It captures key system performance metrics including CPU usage, memory consumption, disk utilization, and network performance. The system consists of a Flask backend for data collection and API services, a PostgreSQL database for storing historical metrics, and a React frontend for interactive visualization.

The application provides:

- **Real-time monitoring**: Live updates of system metrics through REST APIs.
- **Interactive dashboards**: Graphical representation of metrics for quick comprehension.
- **Historical analysis**: Storage and visualization of historical data for trend identification.
- **Cloud-native deployment**: Hosted on scalable cloud infrastructure, ensuring accessibility from any browser.

The project is aimed at providing a scalable, reliable, and user-friendly monitoring tool for developers and system administrators to track the health of cloud-native applications.

Goals

- Build a cloud-native monitoring solution to track system metrics in real time.
- Provide an intuitive, interactive dashboard for visualizing system health.
- Enable storage and analysis of historical performance data.
- Ensure seamless integration between backend APIs and frontend visualizations.
- Deploy the solution in a cloud environment, ensuring accessibility and scalability.

Objectives

• Develop a monitoring tool capable of collecting and displaying real-time metrics.
• Enable users to visualize system health through graphical dashboards.
• Implement historical data tracking for trend analysis.
• Deploy the complete solution using a cloud-native approach.
• Integrate backend APIs with frontend visualization seamlessly.

Work Breakdown Structure

Phase	Tasks	Deliverables
1. Environmental Setup	- Install Python, Flask, Node.js, React - Set up virtual environments and dependencies - Initialize project repository	Development environment ready
2. Backend Development	- Design database schema (PostgreSQL) - Implement Flask API endpoints (/metrics, /history) - Integrate metric collection scripts (psutil, prometheus_client)	Functional backend APIs with database integration
3. Frontend Development	- Design dashboard layout - Implement real-time charts and tables - Responsive UI for metrics visualization	Interactive React dashboard
4. Integration	- Connect frontend to backend using fetch/axios calls - Implement real-time updates and error handling - End-to-end testing	Fully integrated system with live updates
5. Deployment	- Dockerize backend and frontend - Deploy on cloud platform (Render or AWS EKS) - Configure environment variables	Cloud-hosted, accessible monitoring tool
6. Testing & Debugging	- Unit testing for backend and frontend - Integration testing - Debug performance issues	Verified and reliable system
7. Documentation & Handover	- Prepare project manual - Create diagrams and screenshots - Provide deployment and usage instructions	Complete project documentation

CHAPTER 2: SYSTEM ANALYSIS

Proposed System

The proposed system is a cloud-native, full-stack monitoring application designed to provide real-time insights into the performance of cloud applications and infrastructure. It leverages Flask as the backend API service, React for frontend visualization, and PostgreSQL for storing historical metrics.

Key Features:

1. Real-Time Monitoring: Collects live metrics such as CPU, memory, disk usage, and network performance from cloud-hosted applications.
2. Interactive Dashboard: Displays metrics using charts, graphs, and tables with dynamic updates for user-friendly analysis.
3. Historical Data Tracking: Stores past performance data to identify trends, anomalies, and performance degradation over time.
4. Cloud-Native Deployment: Fully containerized using Docker and deployed on AWS EKS or Render for scalability and accessibility.
5. Scalability & Reliability: Designed to handle multiple instances of monitored applications simultaneously, ensuring consistent performance monitoring.
6. Alerting & Notifications (Optional Enhancement): Can be extended to send alerts when metrics exceed predefined thresholds.

Requirement Analysis

Requirement analysis defines the functional and non-functional requirements necessary to build the system.

Functional Requirements

1. The system shall collect real-time CPU, memory, disk, and network metrics from monitored nodes.
2. The system shall store historical performance metrics in a database.
3. The system shall provide RESTful API endpoints for fetching real-time and historical metrics.
4. The system shall display interactive charts and graphs on the dashboard.
5. The system shall support multiple concurrent monitored systems.
6. The system shall allow users to filter and view metrics based on time ranges.

Non-Functional Requirements

1. Performance: The system should fetch and display real-time metrics with minimal latency.
2. Scalability: The system should handle increasing numbers of monitored nodes without performance degradation.
3. Reliability: Ensure continuous monitoring with minimal downtime.
4. Usability: The dashboard should be intuitive, responsive, and accessible via web browsers.
5. Maintainability: Modular design and clean codebase for ease of updates and future feature integration.
6. Security: Protect API endpoints and database access using authentication and secure connections.

Constraints

- The system relies on cloud infrastructure (AWS EKS or Render) for deployment.
- Data collection accuracy depends on access to system metrics APIs or exporters.
- Historical data storage is limited by database capacity and retention policies.

Tech Stack

• Frontend: React.js (with Plotly.js for charts, custom CSS for UI)
• Backend: Flask (Python)
• Database: In-memory data structures (Deque for lightweight metric storage)
• Hosting: Render (Backend + Frontend deployment)
• Tools: Git, GitHub, npm, psutil (Python system monitoring library)
• APIs: REST API for metric fetching and historical data

CHAPTER 3: SYSTEM DESIGN

System Architecture

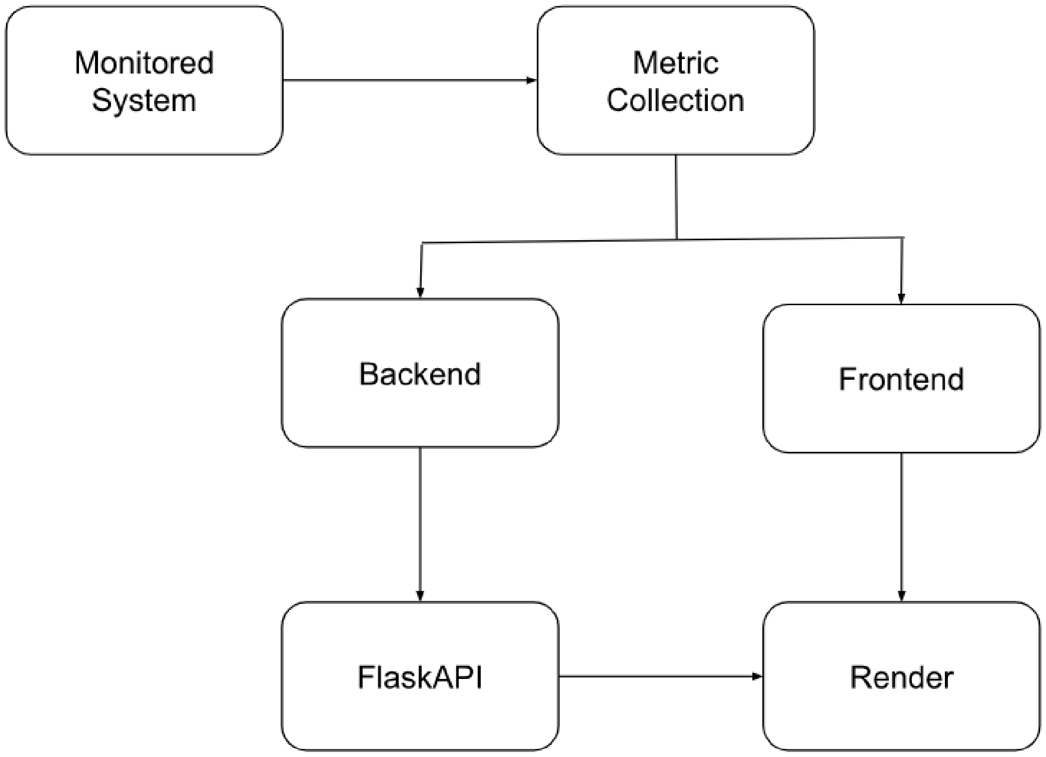
The architecture consists of a React-based frontend and a Flask backend. The backend collects metrics from the host system using the psutil library, while the frontend fetches these metrics via REST APIs and visualizes them using charts. Both components are hosted independently on Render, with the frontend calling the backend using its public API URL.

Workflow

1. User opens the dashboard URL hosted on Render.
2. The React frontend requests system metrics via REST API endpoints from the Flask backend.
3. The backend collects data using psutil, processes it, and returns it in JSON format.
4. The frontend updates charts and metrics every few seconds using polling intervals.
5. Alerts are displayed when threshold limits are breached (e.g., CPU > 85%).

Module Design

System Design



CHAPTER 4: IMPLEMENTATION

The implementation of the project is broken down as follows:

Phase 1: Environment Setup

The first phase focused on preparing a stable development environment for both backend and frontend.

Steps:

1. Installed Python 3.x and set up a virtual environment for the Flask backend.
2. Installed Flask and required Python packages (Flask-RESTful, psutil, prometheus_client, SQLAlchemy, etc.).
3. Installed Node.js and npm to set up the React frontend environment.
4. Initialized React project using create-react-app.
5. Installed frontend dependencies for charts and UI components (recharts, axios, material-ui).
6. Configured version control using Git and created repository structure for modular development.

Phase 2: Backend Development

This phase focused on implementing the core monitoring logic and APIs.

Steps:

1. Designed Flask API structure with endpoints for:
    - Real-time metrics (/metrics)
    - Historical metrics (/history)
2. Implemented metric collection scripts using psutil for CPU, memory, and system metrics.
3. Configured PostgreSQL database for storing historical data.
4. Integrated database models using SQLAlchemy to handle metric storage and retrieval.
5. Developed API endpoints to serve JSON responses to frontend requests.
6. Implemented error handling and validation for robust API responses.

Phase 3: Frontend Development

This phase involved designing a user-friendly dashboard for monitoring metrics.

Steps:

1. Designed the dashboard layout using React components for charts, tables, and metric cards.
2. Implemented real-time charts using Recharts to visualize CPU and memory usage over time.
3. Created historical metrics charts and tables for trend analysis.
4. Implemented responsive design to ensure proper rendering on different screen sizes.
5. Added UI enhancements for better readability (color coding, tooltips, and legends).

Phase 4: Integration

Integration ensured seamless communication between frontend and backend.

Steps:

1. Connected frontend React components to backend Flask APIs using fetch and axios calls.
2. Implemented data polling for real-time metric updates.
3. Verified JSON responses and rendered data accurately on charts and tables.
4. Handled edge cases such as API downtime or missing data gracefully.
5. Conducted end-to-end testing to ensure data flow consistency.

Phase 5: Deployment

Deployment made the application accessible for real-world usage.

Steps:

1. Dockerized the Flask backend for containerized deployment.
2. Built the React frontend for production.
3. Hosted both backend and frontend on Render.
4. Configured environment variables for database connections and API endpoints.
5. Verified live deployment and ensured proper communication between frontend and backend.

Phase 6: Testing & Debugging

The final phase focused on ensuring functionality, reliability, and accuracy.

Steps:

1. Conducted unit testing for backend API endpoints using Python’s unittest framework.
2. Validated database read/write operations for historical metrics.
3. Tested frontend components for accurate data rendering and responsive design.
4. Conducted integration testing to ensure real-time updates worked correctly.
5. Debugged issues such as incorrect metrics, chart rendering errors, and API latency.
6. Optimized performance for faster data retrieval and chart updates.

Results

The Cloud Native Monitoring Application successfully provides a real-time, cloud-hosted performance dashboard. All system metrics are updated dynamically, and historical trends help in identifying performance bottlenecks. The integration between Flask and React through Render ensures scalability and cross-platform accessibility.

Conclusion

This project demonstrates the end-to-end development and deployment of a cloud-native monitoring application. By combining Python’s backend capabilities with a modern React frontend, it provides a responsive, efficient, and scalable solution for monitoring infrastructure performance in real-time.

CHAPTER 5: FUTURE WORK

- Integrate authentication for multiple users.
- Store metrics persistently using a cloud database (e.g., MongoDB, PostgreSQL).
- Add alert notifications via email or Slack.
- Implement containerized monitoring for distributed systems using Docker and Kubernetes.

CHAPTER 6: REFERENCES AND APPENDIX

Project Link: https://cloud-native-monitoring-application-mp0v.onrender.com/

GitHub Repository: https://github.com/RechaelLop/Cloud-Native-Monitoring-Application