A PBL-I Synopsis on

Metricly : A Server Monitor

Submitted to Manipal University Jaipur

Towards the partial fulfillment for the Award of the Degree of

BACHELOR OF TECHNOLOGY

In Computers Science and Engineering

2023-2027

By

Saurabh P

23FE10CSE00264

MANIPAL UNIVERSITY JAIPUR

Under the guidance of

Dr. Shikha Mundra

[illegible] 20/11/24

Signature of Supervisor

Department of Computer Science and Engineering

School of Computer Science and Engineering

Manipal University Jaipur

Jaipur, Rajasthan

1. Introduction

What is the Problem?

Server monitoring is crucial for maintaining performance and preventing issues, involving tracking metrics like CPU usage, memory consumption, disk usage, and network traffic. With the growth of cloud computing[5], Docker, and microservices, demand for effective monitoring tools has risen sharply.

Current tools, such as Prometheus and Grafana[4], though powerful, are complex and require significant setup and technical expertise, making them less accessible to small teams or individual developers. Simpler options like Glances are more user-friendly but lack depth, providing only basic overviews without detailed insights into containerized applications or specific processes. This limitation hinders administrators in diagnosing issues and optimizing resources effectively.

Therefore a gap exists for a user-friendly, intuitive server monitoring solution offering comprehensive, real-time insights with minimal setup beneficial for both beginners and experienced administrators seeking efficiency and ease of use.

Why is this a Problem?

The limitations of existing server monitoring tools impact users in several ways:

- **Complex Setup**: Tools like Prometheus and Grafana[4] offer scalability and detailed metrics but require extensive configuration. This complexity deters small teams and individual developers, leading them to opt for simpler tools that may lack key features.
- **High Resource Overhead**: Comprehensive solutions consume significant CPU and memory, making them unsuitable for smaller servers or personal setups.
- **Lack of Docker-Specific Features**: Effective monitoring of Docker containers is essential, yet many traditional tools do not natively support Docker metrics, requiring additional configurations.
- **Outdated User Interfaces**: Legacy tools such as Naglos[1] have outdated interfaces, which can be difficult to navigate compared to modern, interactive dashboards. This reduces user productivity and makes interpreting performance data more challenging.
- **Limited Customization and Alerts**: While tools like Netdata provide real-time monitoring, they often lack customization options for dashboards and alerts, limiting their effectiveness for users needing tailored notifications and visualizations.

How Can This Problem Be Solved?

Metricly aims to be a software based server monitoring system that addresses these issues. The key features of this solution include:

- **Real-Time Metrics Collection:**
  Efficiently tracks server metrics like CPU, memory, disk, and network performance using Python libraries like *psutil*.
- **Docker Integration:**
  Monitors container-specific metrics, including resource usage and network traffic, using *docker-py*, helping in container level performance insights.
- **User-Friendly Interface:**
  Provides an interactive web interface[2] (HTML, CSS, JavaScript) with real-time charts and graphs for easy visualization.
- **Custom Alerts:**
  Enables threshold based alerts for issue resolution through notifications when metrics exceed limits.
- **Scalable Design:**
  Lightweight, containerized architecture ensures seamless deployment, minimal overhead, and scalability for environments of any size.

Motivation

This server monitoring tool aims to address key challenges in existing solutions by:

- **Improving Efficiency:** Simplifying setup and offering a comprehensive approach to server management.
- **Enhancing User Experience:** Providing an intuitive interface with real-time visualizations for quick, easy insights.
- **Reducing Operational Burden:** Automating data collection and delivering actionable insights to streamline performance optimization.
- **Leveraging Modern Technologies:** Utilizing advancements in Python, web development, and containerization for innovative monitoring.
- **Ensuring Scalability:** A lightweight, Docker-based design makes it adaptable for various environments and long-term use.

This **software-based project** provides real-time monitoring of server and container metrics via web interface, using python for data collection, docker for containerization, and frameworks like React for a responsive UI, ensuring easy, hardware-independent deployment.

2. Literature Survey

Here are the existing solutions with their pros and cons:

Tool/Method	**Pros**	**Cons**
Prometheus + Grafana[4]	Highly configurable and scalable	Complex setup, high resource usage on small servers
Naglos[1]	Extensive monitoring capabilities	Outdated user interface, manual configuration required
Glances	Lightweight and easy to use	Basic functionality, lacks detailed Docker monitoring
Netdata	Real-time monitoring with alerts	Limited customization options, high memory usage
Zabbix	Comprehensive features with automation	Steep learning curve, complex configuration
Datadog	Cloud-based with extensive integrations[5]	Expensive for small teams, high data usage
New Relic	Detailed performance metrics[6]	High cost, can be overwhelming for beginners
CAdvisor	Docker container-specific monitoring	Limited to container stats, lacks broader system insights
htop	Real-time system monitoring	Limited to local use, lacks historical data analysis
Dynatrace	AI-powered monitoring and analytics	High pricing, requires extensive configuration

Metricly bridges the gaps in existing tools by offering a lightweight, user-friendly monitoring system that combines real-time server and Docker metrics with an intuitive interface. Unlike complex or resource-heavy options, it provides a scalable, efficient, and cost-effective approach suitable for diverse users. This ensures a seamless balance of functionality, accessibility, and performance optimization[6].

3. Comparative Study

Feature	**Prometheus + Grafana[4]**	**Naglos[1]**	**Glances**	**Netdata**	**Metricly**
Ease of Setup	Complex	Moderate	Easy	Moderate	Easy
Docker Integration	High	Low	Low	Moderate	High
Resource Overhead	High	Low	Low	High	Low
User Interface	Modern	Outdated	Basic	Modern	Modern
Custom Alerts	Yes	Yes	No	Yes	Yes

This table summarizes the core features of various platforms. Metricly stands out by offering a wide range of features like docker integration, ease of setup, low resource overhead, modern interface and custom alerts.

4. Objectives

These objectives aim to create a comprehensive, efficient, and user-centric server monitoring solution that caters to a wide range of users and environments.

- **Enable Comprehensive Docker Monitoring:**
  Integrate docker tracking to capture container-specific resource utilization, such as CPU, memory, and network usage, ensuring detailed insights at the container level.
- **Simplified Data Visualization and Notifications**: Develop a responsive, modern web interface with real-time graphs and interactive charts, complemented by an integrated alert system to notify users of performance anomalies, enabling proactive issue resolution.

5. Planning of Work

The development of the server monitoring system is structured into five distinct phases, each focusing on specific objectives and deliverables. This phased approach ensures systematic progress, continuous evaluation, and the delivery of a robust and user-centric solution.

Phase 1: Requirements Gathering and Analysis

This phase involves a deep understanding of the problem domain, analyzing existing solutions, and identifying user needs to define the scope and goals of the project.

1. **Market Analysis:**
    - Conduct an in-depth study of current server monitoring tools, including Prometheus, Naglos[1], Netdata etc.
    - Identify strengths and limitations of these tools, such as ease of use, resource efficiency, and Docker monitoring[3] capabilities.
    - Analyze user reviews on platforms like reddit and other tech forums to pinpoint common pain points, such as complexity in configuration, high costs, or lack of customization.
2. **User Needs Assessment:**
    - Conduct surveys and interviews with potential users, including system administrators, developers, and small business owners.
    - Gather insights on desired features like real-time monitoring[2], Docker integration[3], customizable alerts, and an intuitive user interface.
    - Create user segments representing different audience segments to guide design decisions.
3. **Requirement Specification:**
    - Define functional requirements, such as real-time metric collection, Docker container tracking, and alert mechanisms.
    - Document non-functional requirements like scalability, minimal resource overhead, and ease of deployment.
    - Develop mockups of the user interface to visualize the proposed dashboard layout and functionality.

Phase 2: System Design and Architecture

In this phase, the project transitions from requirements to designing the technical architecture and workflows to ensure an efficient and scalable solution.

1. **Backend Design:**
    - Plan the core architecture for real-time data collection using Python libraries like psutil for system metrics and docker-py for Docker metrics[3].
    - Design a SQL database schema using MongoDB/MySQL to efficiently store time-series data.
    - Define API endpoints for seamless communication between the backend and the frontend.
2. **Frontend Design:**
    - Choose React or Vue.js for creating a dynamic and responsive web interface.
    - Design interactive dashboards with charts, graphs, and tables using visualization libraries like Chart.js or D3.js.
    - Ensure cross-device compatibility by incorporating responsive design principles.
3. **Deployment Architecture:**
    - Plan for a lightweight containerized application using Docker to simplify deployment.
    - Design scalable deployment strategies using Docker Compose for orchestrating multi-container environments.

Phase 3: Development and Implementation

This phase focuses on implementing the system's functionality, covering both backend and frontend components, and establishing an efficient development workflow.

1. **Environment Setup:**
    - Install and configure Python, Docker, and MongoDB/MySQL as core technologies.
    - Set up a version control system using Git and GitHub for collaborative development.
    - Create a virtual Python environment to manage dependencies and ensure consistency.

2. **Backend Development:**
    - Implement real-time metric collection using psutil to gather CPU, memory, disk, and network usage.

   - Integrate Docker API[3] (docker-py) to monitor[2] container-specific metrics, such as resource allocation and network statistics.
   - Develop API endpoints for retrieving and sending data between the backend and frontend.
   - Optimize data storage and retrieval in SQL databases to handle large volumes of real-time metrics efficiently.
3. **Frontend Development:**
   - Build a responsive web application using React or Vue.js, focusing on user experience and interactivity.
   - Develop real-time visualizations using Chart.js or D3.js for performance metrics[6] and trends.
   - Incorporate navigation elements, filters, and customization options for enhanced usability.

Phase 4: Testing and Quality Assurance

Testing is critical to ensure the system meets functional requirements and performs reliably in real-world scenarios.

1. **Unit Testing:**
   - Develop unit tests for backend components to verify data collection accuracy and functionality.
   - Test frontend components to ensure correct rendering of charts, tables, and user interactions.
2. **Integration Testing:**
   - Test the interaction between backend APIs and the frontend dashboard to ensure seamless data flow.
   - Validate the integration of Docker monitoring features for real-time container metrics.
3. **Performance Testing:**
   - Simulate high loads to evaluate the system's resource efficiency and scalability.
   - Test responsiveness and data retrieval times for the web interface under various conditions.
4. **User Testing:**
   - Conduct beta testing with target users, including developers and administrators, to gather feedback on usability and functionality.
   - Identify and resolve issues related to the user interface, feature accessibility, or performance bottlenecks[6].

Phase 5: Deployment and Maintenance

The final phase focuses on deploying the system in a production environment and ensuring long-term usability through documentation and updates.

1. **Deployment:**
    - Containerize the application using Docker to simplify installation and ensure consistency across environments.
    - Automate deployment with Docker Compose, enabling quick and hassle-free setup for users.
    - Optimize the system for both local and cloud-based environments[5], offering flexibility to different user groups.
2. **Documentation:**
    - Create comprehensive guides for installation, configuration, and troubleshooting, tailored to technical and non-technical users.
    - Develop a user manual/wiki explaining dashboard features, metric interpretation, and alert customization.
3. **Future Enhancements and Maintenance:**
    - Outline plans for additional features, such as historical data analysis, advanced alerting, and support for more monitoring plugins.
    - Establish a maintenance schedule for regular updates, bug fixes, and user support.

6. Bibliography/References

1. Renita, J., & Elizabeth, N. E. (2017, March). Network's server monitoring and analysis using Nagios. In *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)* (pp. 1904-1909). IEEE.
2. Popa, S. (2008). WEB Server monitoring. *Annals of University of Craiova-Economic Sciences Series*, *2*(36), 710-715.
3. Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, *17*(3), 228.
4. Leppänen, T. (2021). Data visualization and monitoring with Grafana and Prometheus.
5. Felani, R., Al Azam, M. N., Adi, D. P., Widodo, A., & Gumelar, A. B. (2020). Optimizing virtual resources management using Docker on cloud applications. *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*, *14*(3), 319-330.
6. Kaur, J., & Reddy, S. R. N. (2019). Testing of Linux Performance Monitoring Tools on ARM Based Raspberry Pi. *Journal of computational and theoretical nanoscience*, *16*(9), 3955-3960.