FUKURO: An Agent-Driven Framework for Real-Time Remote Host Resource Management

Muhammad Amirul Asraf Bin Mustafa[1], Kurk Wei Yi[2], Mohd Hariz Naim @ Mohayat[3], Kamarularifin Abd Jalil[4]

[1,2,3]Fakulti Teknologi Maklumat dan Komunikasi, Universiti Teknikal Malaysia Melaka (UTeM), Durian Tunggal, Melaka, 76100, Malaysia

[4]Department of Computer Technology and Network, Universiti Teknologi MARA, Shah Alam, Malaysia

DOI: https://dx.doi.org/10.47772/IJRISS.2025.910000556

Received: 20 October 2025; Accepted: 28 October 2025; Published: 18 November 2025

ABSTRACT

Remote host resource monitoring is a critical aspect of maintaining and operating distributed computing environments, especially in the era of DevOps where development and operations are integrated to support continuous delivery. As the number of remote systems and personnel involved in operational phases increases, existing monitoring solutions struggle with decentralization, limited accessibility, and lack of user-centric customization. This project introduces FUKURO (Fundamental Kernel Utilization Realtime Overseeing), an integrated monitoring system designed to centralize remote host management and improve efficiency through automation. The system comprises three core components: a Python-based agent installed on the remote host to collect performance metrics such as CPU, memory, disk, and network utilization; a NodeJS web service server that aggregates data into a centralized MySQL database and manages alerts; and a Flutter-based mobile application providing a cross-platform interface for real-time visualization and configuration. Through systematic testing, FUKURO successfully achieved real-time metric collection, threshold-based alert notifications, and seamless data synchronization between hosts and mobile devices. The results demonstrate that the system enhances accessibility, simplifies collaboration among DevOps teams, and reduces technical overhead for monitoring remote resources. Future enhancements aim to extend agent compatibility across multiple operating systems and integrate predictive analytics for proactive maintenance.

Keywords: Remote monitoring, DevOps, Agent-based system, Resource utilization, Flutter

INTRODUCTION

In modern computing environments, many applications and services run on distributed and remote hosts such as cloud servers, virtual private servers (VPS), and on-premises machines. Ensuring their optimal performance requires continuous monitoring of resource utilization, specifically CPU, memory, disk, and network metrics, since performance degradation in any of these components can lead to failures or downtime [1]. Effective monitoring provides early detection of anomalies, proactive fault recovery, and data-driven performance optimization to sustain high availability [1].

The adoption of DevOps has transformed how organizations build and maintain software systems by combining development and operations practices to enable continuous integration (CI), continuous delivery (CD), and continuous deployment (CDep). This methodology promotes automation, collaboration, and iterative improvement across teams [2]. However, as the number of services and stakeholders involved in software lifecycles increases, so does the complexity of managing and monitoring multiple hosts. Traditional tools such as Prometheus and Grafana, although powerful and open source, often require extensive configuration and operational expertise to be effectively implemented [3].

From a system architecture standpoint, remote monitoring solutions generally follow either agent-based or agentless models. Agent-based systems install lightweight agents on each host to collect metrics locally and

transmit them to a central server, providing deeper system insights and control. Conversely, agentless monitoring relies on network protocols such as Simple Network Management Protocol (SNMP) to retrieve metrics without additional software installation [4]. Another design consideration is whether to use a pull model, where the server periodically requests data from agents, or a push model, where agents send data at defined intervals. The push model is often preferred for distributed networks and NAT-protected environments due to its simpler connectivity and scalability [5].

Alerting and notification mechanisms are equally critical in ensuring that performance anomalies receive timely attention. Poorly tuned alert thresholds can overwhelm administrators, leading to alert fatigue and reduced responsiveness [6]. Recent approaches advocate for adaptive alerting systems that prioritize and filter alerts based on user roles and system context, improving accuracy and reducing cognitive overload [6]. In parallel, the integration of mobile and cross-platform interfaces has expanded monitoring accessibility, allowing users to manage and visualize host performance anytime and anywhere [7].

Motivated by these needs, this paper introduces FUKURO—a centralized, agent-based remote host monitoring system that integrates Python agents for metric collection, a Node.js backend for data aggregation, and a Flutter-based mobile application for visualization and notification. FUKURO employs a push-based agent communication model to simplify deployment and enhance scalability while incorporating customizable alerting and access-control features to address DevOps collaboration challenges identified in recent research [1]–[7]. The following sections discuss the system's background, related work, methodology, experimental results, and future research directions.

Background

Food allergies are increasingly recognized as a major global health issue with a rising prevalence across all age groups. Recent studies highlight that the incidence of both adult and infant food allergies has continued to grow worldwide, primarily due to changes in diet, food processing, and environmental factors [7]. Despite ongoing efforts to improve labelling and awareness, accidental allergen exposure remains common in daily food preparation and consumption. Conventional recipe databases and nutrition-tracking applications focus largely on nutrient values rather than allergen safety, leaving users to manually identify allergens—a process prone to error and inefficiency.

Advancements in artificial intelligence (AI) and machine learning (ML) have provided promising tools for improving food-safety monitoring and allergen detection. AI algorithms can analyse ingredient-level data, identify hidden allergenic proteins, and predict potential reactions using pattern-recognition models. Recent work by Yang et al. [8] demonstrated a novel AI-driven method using near-infrared spectroscopy (NIRs) for early detection of non-specific lipid transfer protein (nsLTP) allergens, enabling fast and non-destructive screening. Similarly, Li et al. [9] developed a portable fluorescence biosensing system enhanced with AI, capable of detecting multiple allergens simultaneously, marking a significant leap toward real-time, point-of-care allergen identification.

Beyond detection, the integration of AI into personalized nutrition has fostered the rise of precision nutrition—an approach that tailors dietary recommendations to an individual's genetic, physiological, and lifestyle factors. Deep learning models that combine microbiome and diet data have demonstrated effectiveness in predicting optimal nutrition strategies and allergy risk profiles [10]. These developments lay a strong foundation for intelligent applications that not only identify allergens but also assist users in modifying recipes according to their unique health needs.

The adoption of mobile health (mHealth) technology further enhances accessibility and real-time feedback in dietary management. Smartphones serve as effective platforms for hosting AI-powered food-safety applications, allowing users to receive instant allergen detection and substitution suggestions. However, most existing mobile apps are limited to static allergen lists and lack adaptive learning capabilities [11]. Thus, integrating AI, natural language processing (NLP), and mHealth technologies offers a timely solution to improve food safety and empower users with allergies to make informed, personalized dietary decisions [8], [9].

RELATED WORK

Monitoring systems for cloud, edge, and IoT platforms have recently moved beyond static data collection toward adaptive, intelligent, and context-aware approaches. One recent work by Calderon et al. [14] presents a monitoring framework that uses Elasticsearch and Apache Kafka to evaluate IoT platform performance. Their system captures performance metrics of edge nodes over time and emphasizes scalable storage and efficient dashboarding for long-term analysis. While effective for infrastructure with many edge devices, it does not deeply address real-time alerting or mobile interface concerns.

Another line of research focuses on managing the life-cycle of probes in monitoring solutions. Tundo et al. proposed a Monitoring-as-a-Service framework which automates the deployment and undeployment of probes in both container and VM environments [15]. This work reduces manual configuration effort and enables dynamic adaptation of what is being monitored, which aligns closely with agent-based architectures that need flexibility in changing measurement scopes.

Anomaly detection in logs is also a key area. MoniLog, introduced by Arthur Vervaet [16], is a log-based anomaly detection system designed for cloud computing infrastructures. It addresses the challenge of high volumes of logs by structuring log streams and learning from administrator feedback to classify anomaly severity. The approach is relevant where monitoring systems need to reduce noise and surface critical issues more intelligently.

Scheduling and queueing systems also influence how resource monitoring should be designed—not just to observe metrics but to take responsive action. “AI-enhanced modelling of queueing and scheduling systems in cloud computing” [17] demonstrates how ML models can improve throughput and resource utilization in cloud environments while predicting workload dynamics. Although that work focuses more on scheduling than monitoring, its insights on prediction and workload forecasting inform how a monitoring architecture might integrate predictive components for scaling or alert tuning.

Together, these works reflect recent trends: probe and metric collection flexibility, anomaly detection with feedback and forecasting for better resource allocation. What’s less present is integration with mobile interfaces, push-based agent communication, and combining alerting, logging, metrics, and prediction in a unified system. The proposed FUKURO system aims to fill those gaps.

Table1. Summary Of Existing System

Ref.	Focus Area	Main Contribution	Limitation / Gap
[14]	Cloud/IoT monitoring framework using Elasticsearch and Kafka	Developed a scalable performance evaluation framework for IoT systems; efficient metric storage and visualization	Limited support for real-time alerting and mobile accessibility
[15]	Dynamic monitoring probe management	Introduced automated deployment and undeployment of monitoring probes in containerized and virtualized environments	Does not address mobile or adaptive feedback integration
[16]	Log-based anomaly detection	Applied structured log analysis and adaptive learning to detect cloud anomalies	Focused on anomaly classification, not resource metrics or mobile delivery
[17]	Predictive modelling using machine learning	Improved scheduling and resource utilization through AI-based forecasting	Concentrates on queueing and workload modelling rather than monitoring

METHODOLOGY

The methodology of this study defines the systematic process used to design, develop, and evaluate the Remote Host Resource Monitoring via Agent system. It follows the software development life cycle (SDLC) framework consisting of four main phases: analysis, design, implementation, and testing.

Analysis

The analysis phase of the Remote Host Resource Monitoring via Agent project establishes the groundwork for understanding the technical and operational needs of a distributed monitoring environment. Traditional monitoring solutions often rely on centralized polling or agentless methods, which are inefficient and difficult to scale across heterogeneous infrastructures. System administrators face challenges in managing multiple remote servers and obtaining real-time visibility into system performance. To address these issues, the proposed FUKURO (Fundamental Kernel Utilization Realtime Overseer) system introduces lightweight agents that continuously collect host metrics, such as CPU, memory, disk, and network usage and transmit them securely to a central server for visualization and analysis.

As for requirement, the system must authenticate users, manage multiple nodes, and provide real-time data visualization through web and mobile dashboards. Security and scalability were identified as critical non-functional goals. Token-based authentication using JSON Web Tokens (JWT) and encrypted data transmission ensure secure communication between agents and the server, while modular design supports concurrent monitoring with minimal system overhead. The system's cross-platform compatibility allows deployment on both Linux and Windows environments, ensuring flexibility and broad applicability.

A feasibility study confirms that the proposed solution is technically, economically, and operationally viable. The adoption of open-source tools such as Python, Node.js, MySQL, and Flutter reduces development cost and simplifies integration with existing infrastructures. The accompanying use-case diagrams illustrate how metrics move from host to server and onward to visualization modules, ensuring consistent, real-time updates (Fig 1). Overall, the analysis demonstrates that an effective remote monitoring system must integrate secure communication, scalable data handling, and lightweight agents which principles that directly guided the system's architectural design described in the next section.

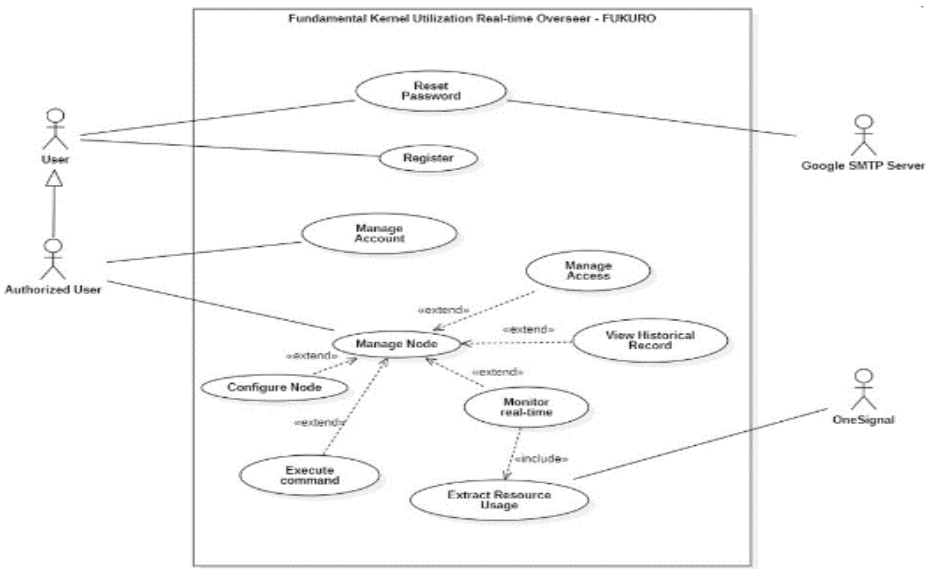

Fig 1. Use case diagram of FUKURO

Design

FUKURO system is structured around a multi-tier architectural pattern which is the presentation, application and data tier which each have its own responsibilities. In the presentation tier lies the component of the system which handles the interfacing with user which presents the data or functions to the user. In FUKURO the primary

GUI (Graphical User Interface) will be handled by the mobile application which will be developed using Dart language and Flutter frameworks. On the other hand, a simple command line interface will be implemented solely for users to log in to agent applications with necessary credentials. The agent application side does not require any other GUI since all of the other processes are done in the background.

The application tier is where all logical processing units are placed. In FUKURO the application tier can be further segmented into two which is the web service server application and Agent application. The crust of the system, the agent application is responsible for extracting, calculating, and submitting the resource usage information to the server. The agent application will only interact with the server application via HTTP for the first-time verification purpose while the rest of the communication will be done over WebSocket connection.

In the server application side, there will be two main components which are the REST endpoints and web socket server. Each component is on the same single server application which makes the server application in general hybrid of REST and WebSocket architecture. Basically, the REST endpoints will provide most of regular services such as managerial operation while the web socket server will handle the live and continuous service such as receiving readings from agent application or serving real-time data to mobile application.

Lastly, the data tier only consists of the MySQL database which is the primary data storage of FUKURO system. The interaction between application tier and the data tier will be handled by the web service server application via SQL connection and the data transaction will be managed through SQL query.

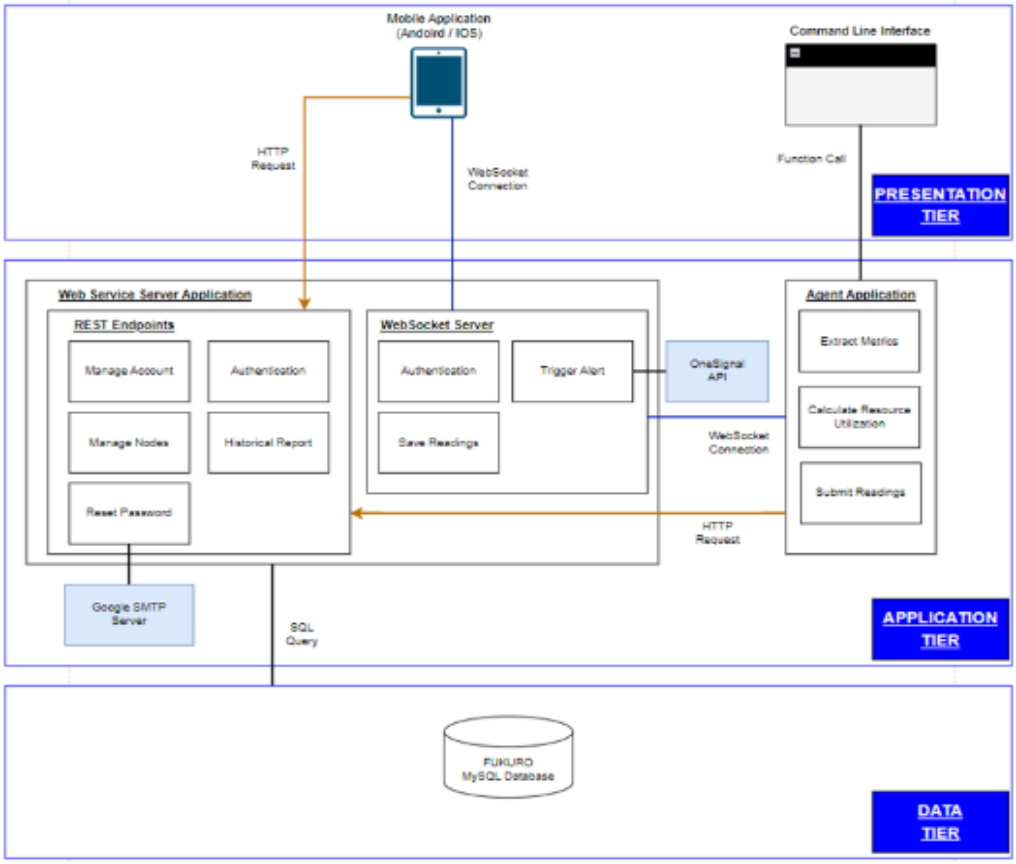

Fig 2 System Architecture of FUKURO

Development

The implementation of the FUKURO system was executed across three distinct sprints, adhering to the Agile methodology. The development process involved the concurrent implementation of the three core components: the web service server, the agent application, and the mobile application. The web service server was developed using Node.js and Express.js, creating a hybrid server that concurrently manages REST API endpoints for standard HTTP requests and a WebSocket server for real-time, persistent connections. Key modules implemented include a Database Controller for managing MySQL connection pools and preventing SQL

injection, a robust Authentication module using dynamically generated JWT (JSON Web Tokens) for secure user and agent verification, and a WebSocket Client Cache for tracking connected agents and facilitating direct communication.

The agent application was implemented in Python, specifically designed for deployment on Unix-based remote hosts. Its core functionality revolves around extracting and calculating the four key resource metrics: CPU, memory, disk, and network usage. The implementation involved creating dedicated classes for each metric, which parse data from the Linux */proc* filesystem. For instance, CPU utilization is calculated by taking two readings from */proc/stat* and applying formulas to derive the percentage used by user processes, system processes, and interrupts. A central *Monitoring Controller* class utilizes multi-threading to run these extraction processes concurrently without interfering with other system operations. The agent communicates with the server via a *Ws Client* class, which uses a listener pattern to handle incoming instructions from the server based on predefined paths, allowing for dynamic reconfiguration of intervals and thresholds.

The mobile application was implemented using the Flutter framework with Dart. It features a comprehensive GUI based on the initial designs, including screens for login, node management, real-time monitoring, and historical reports. The *FukuroRequest* class was created to standardize and simplify all HTTP communications with the server's REST API, automatically handling JWT authentication. For data visualization, the *syncfusion flutter charts* library was integrated. An abstract *Chart Data* class was implemented to provide a unified interface for all metric data types (CPU, memory, disk, network), enabling a single, reusable chart widget to display any metric by implementing a *get Val ()* method. Real-time monitoring was achieved by implementing a *Web Socket Client* in the mobile app that listens for live data streams forwarded by the server from the agent. Furthermore, push notification functionality was integrated using the *onesignal flutter* package, which subscribes the user's device to alerts triggered by the server based on metric thresholds.

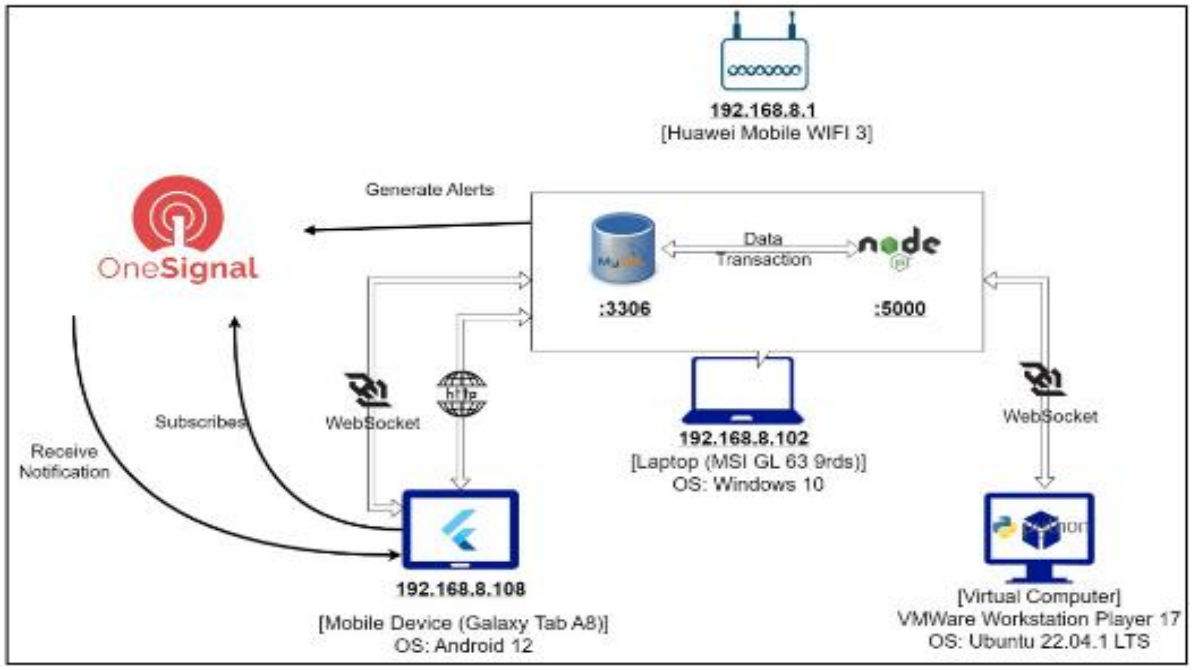

Fig 3. Local software development environment setup

Testing

The testing phase for the FUKURO system was conducted to rigorously verify its functional capability in accurately monitoring remote host resource usage. A comprehensive test plan was executed in a real-world environment, moving beyond the local development setup. The web service server was deployed on a Virtual Private Server (VPS) hosted on DigitalOcean, while the agent application was deployed on two distinct targets: a remote VPS to simulate a cloud host and a local virtual machine to simulate an on-premises host. This setup was designed to evaluate the system's performance under varied conditions and its ability to handle concurrent monitoring sessions.

The core testing strategy employed was black-box testing, focusing on the system's output in response to controlled stimuli. Each of the four primary metrics: CPU, memory, disk, and network usage was tested individually. Resource usage was artificially stimulated on the monitored hosts using command-line stress-testing tools. For CPU and memory, tools like *stress* and *stress-ng* were used to spawn worker processes that consumed a specific percentage of resources for a set duration. Disk I/O was stressed using *stress-ng* to perform intensive read and write operations. Network usage was tested by establishing a client-server connection between the two monitored hosts using the *iperf* tool, which generated a continuous stream of data for one minute.

RESULT

The Results section presents the outcomes of system testing for the FUKURO. It evaluates the system's performance in monitoring CPU, memory, disk, and network usage under different workloads. These results demonstrate the system's accuracy, responsiveness, and stability in collecting and displaying real-time resource data, confirming the effectiveness of the agent-based monitoring approach.

CPU Usage Test

This testing is conducted to verify FUKURO ability in monitoring the usage of CPU metric in both target host via the real-time monitoring feature. The testing is conducted for 30 second and after the first five second, the CPU usage is stimulated on the monitored host directly via their own command terminal using the commands mentioned in Table II. The "stress --cpu 1 --timeout 20" command is executed on the remote host deployed in Digital Ocean VPS while "stress-ng -c 0 -t 20s -l 75 –times" is executed on the local host virtual machine. After the commands executed, it will spawn processes to stimulate the CPU usage while the stress-ng command also allows specifying the target CPU usage to simulate which in this case is 75%.

Table2 SUMMARY OF EXISTING SYSTEM

Metric	**Command**
CPU	stress --cpu 1 --timeout 20 stress-ng -c 0 -t 20s -l 75 --times
Memory	stress-ng --malloc 2 --malloc-bytes 70% -t 20s
Disk	stress-ng -d -2 -readahead 1 -t 20s
Network	iperf -s -w 2M -p 5005 iperf -c 139.59.233.99 -p 5005 -w 2M -t 60s

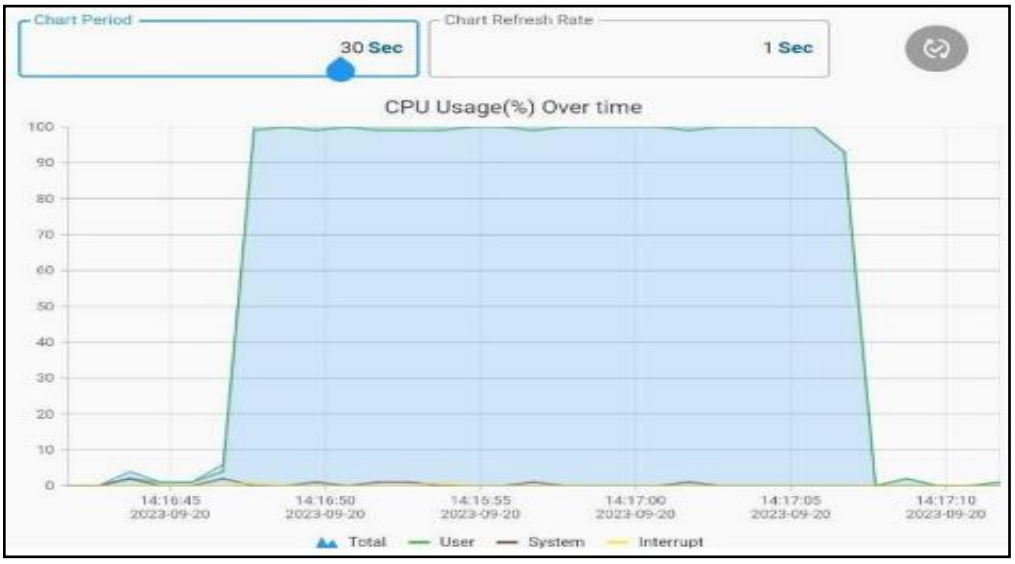

Fig 4. CPU test result on remote host

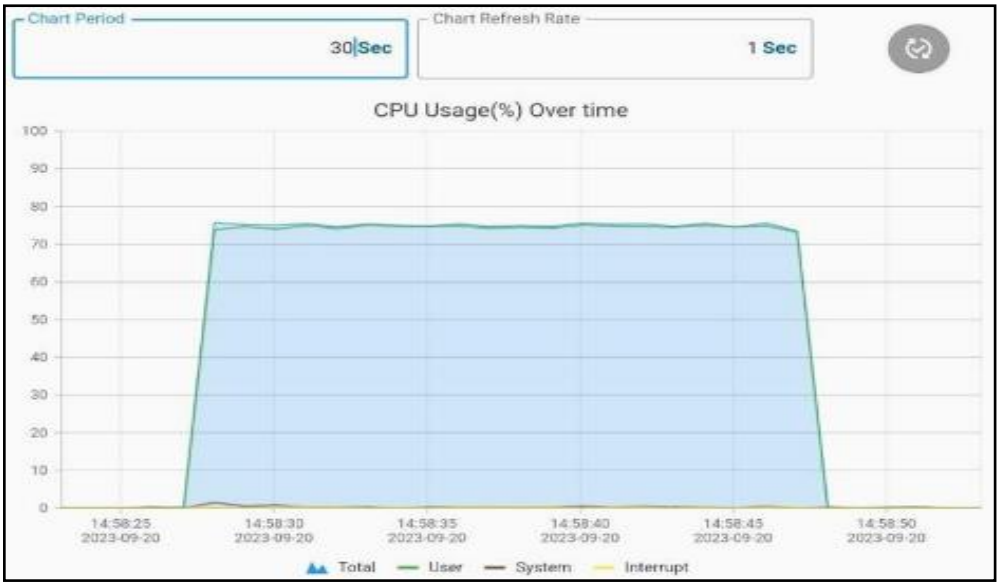

Fig 5. CPU test result on local VM host

Fig 4 and Fig 5 shows the outcome of the testing in the FUKURO mobile application real-time monitoring module. Previously it is mentioned that the "stress-ng" command executed on the local VM host includes the "-l 75" parameter which instruct the stressors to repeat random widespread memory read and writes and -l is parameter specifies the CPU load limit for the CPU. From the result observed in Figure 6.4 that shows the CPU usage fluctuates around 75%, it can be concluded that FUKURO is capable of accurately measure the current CPU usage on the monitored target.

Memory Usage Test

This testing is conducted to verify FUKURO ability in monitoring the usage of Memory metric in both target host via the real-time monitoring feature. The testing is conducted for 30 second and after the first five second, the Memory usage is stimulated on the monitored host directly via their own command terminal using the commands mentioned in Table II. The test is conducted in duration of 30 seconds while the commands executed 5 second into the testing to spawn processes to stimulate the Memory usage which will perform memory allocation and reallocation based on the malloc-bytes parameter.

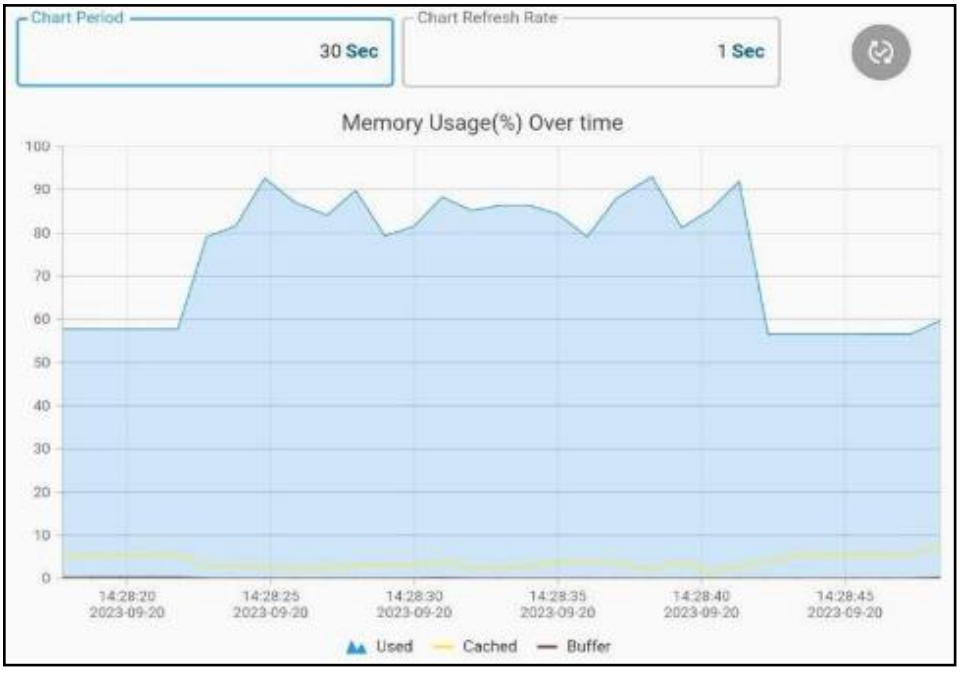

Fig 6. Memory test result on remote host

Fig 7. Memory test result on local VM host

The command executed in Fig 6 does specify the parameter "—malloc-bytes 70%". However, it is important to clarify that the 70% in this parameter does not represent the memory usage to achieve in the testing, but it is the size of the memory which the spawned process will allocate and reallocate based on the available memory at the time of execution which varies and changes continuously due to the nature of memory which has reclaimable portion. Hence, the usage will not fix to be 70% but instead will be approximately around current memory usage + 70% of currently available memory. Hence, the memory usage reaches 90% when the command executed on the remote host which used 57.73% memory and only have 42.27% memory available before the stimulation. Since 70% of 42.27% (available) is 29.59%, the expected outcome of the command execution should be memory usage around 87.32% which is the sum of currently used memory and the 70% of available memory (57.73% + 29.59%). Since the memory available will determine the size of memory to be allocated or reallocated by the stressor, when there are more memory available, the usage will increase more as depicted in Fig 7 which has clearly steeper slope when the stimulation starts compared to the chart in Figure 6.6. Thus, reading depicted by FUKURO in Figure 6.6 is indeed reasonable since it fluctuates around 87.32%.

Disk Usage Test

This testing is conducted to verify FUKURO ability in monitoring the usage of Disk metric in both target host via the real-time monitoring feature. The testing is conducted for 30 second and after the first five second, the Disk usage is stimulated on the monitored host directly via their own command terminal using the commands mentioned in Table II. After the commands executed, it will spawned processes to stimulate the disk usage which performs read and write process on dummy files and directories. The "-t 20s" is supposed to limit the stressor working time to 20 seconds but the actual execution time relies on the disk performance itself which results to the stressor finishes late in Local VM Host. The total duration of the test was 30 seconds, and the command is executed 5 seconds into the test producing readings.

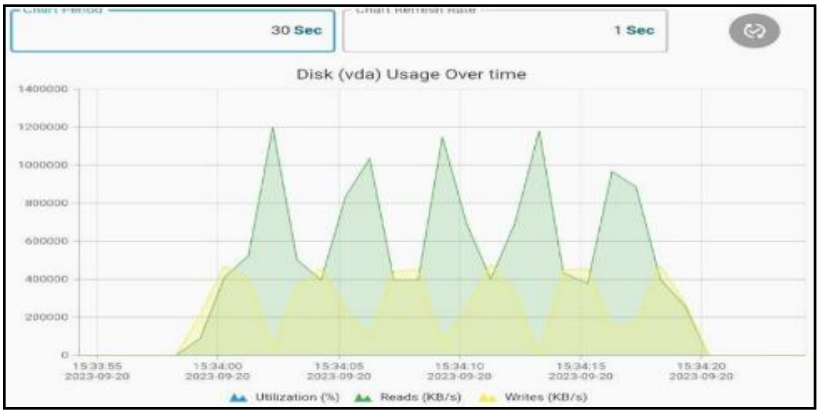

Fig 8 Disk test result on remote host

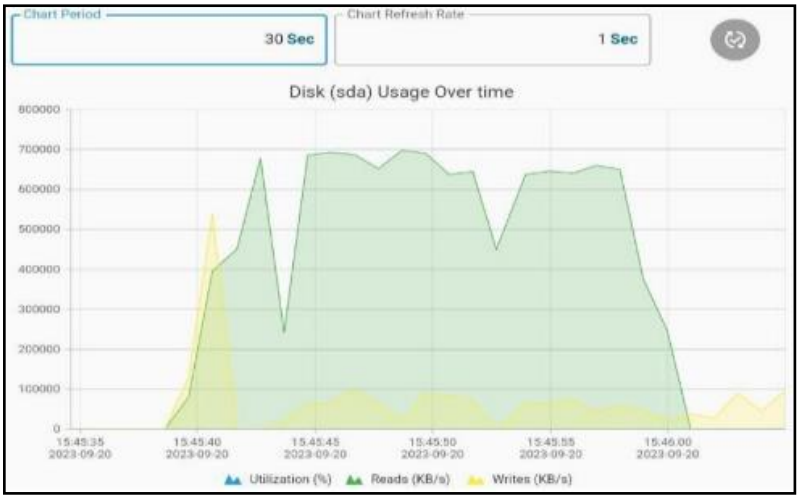

Fig 9. Disk test result on local VM host

Unlike CPU and Memory usage, disk usage is more to I/O (input and output) process which doesn't have clear value to use as reference to measure the accuracy of the readings. However, based on the result shown in Fig 8 and Fig 9 is capable of detecting the disk usage changes stimulated by the executed command successfully which is very distinct compared to the reading 5 seconds before and after the stimulation when the disk is idle.

Network Usage Testing

Network usage test is conducted simultaneously on both monitored targets which relates to each other in a simple client-server connection using the IPerf tools. The remote host deployed on the Digital Ocean will act as the server side and uses IPerf to listen to port 5005 while the local host on VM will act as client which connects to the remote host deployed on IP address 139.59.233.99. During the testing the IPerf tool will be used to send data from the client to server which both is the monitored host thus resulting to receive and transmit reading can be observed simultaneously.

The command executed basically establish client-server connection between both monitored host and the client which is the local VM host will send data continuously to the server which is the remote host for 60 second as defined by the arguments "-t 60s". The "-W 2M" sets the windows size to 2 Mega Byte which reflects how much data can be in the network according to iperf.fr (n.d.). The whole test duration for network testing is 70 seconds and the command is executed 5 seconds after the testing started.

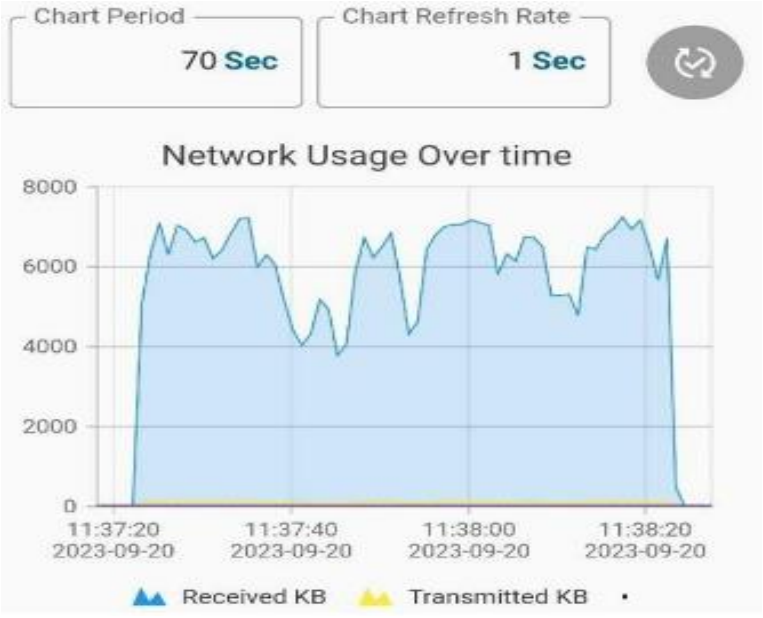

Fig 10. Network test result on remote host

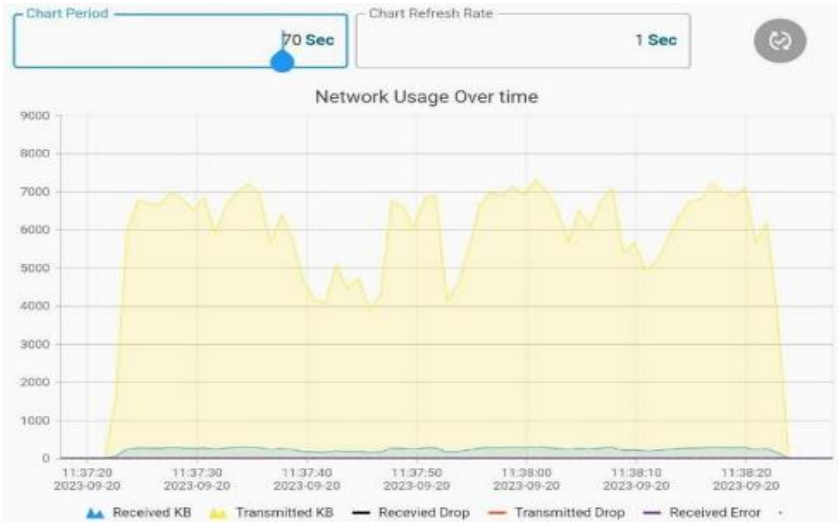

Fig 11Network test result on local VM host

The results shown in Fig 10 and Fig 11 shows that the receives on the remote host side almost always balances the transmit from the local VM host side since they are interacting with each other. The following Table III shows the calculated average based on the detailed data for every second extracted by FUKURO.

Table3 Summary of Existing System

Metric	**Duration Considered**	**Average (Kilo Bytes/Second)**	**Average (Mega bits/second) [KB x 0.008]**
Remote Host Receive	11:37:23 ~ 11:38:22	6143.513833	49.15
Local VM Host Transmit	11:37:22 ~ 11:38:21	6063.078167	48.50

Although the value of the average speed in Table III very near but is not exactly equals to the value measured by IPerf tool, it is most likely because the IPerf tools measure only the transaction within its client-server while FUKURO monitor the whole network usage from all processes. Furthermore, the remote host being monitored is also the same VPS which FUKURO server is deployed during this testing. The network transaction to retrieve and send data from agent to the mobile application is also included in FUKURO readings which makes it is normal for the remote host receive value to have larger difference with IPerf tool compared to the difference between the local VM host and its respective IPerf measurement since the local VM does not have significant network application running other than the IPerf tools itself during the testing. It can be concluded that FUKURO is indeed capable of monitoring network usage accurately.

In term of performance that has been tested, the FUKURO has shown a baseline performance with minimal affect to both monitored hosts and the server side receiving data. That ware no reported case that the monitored hosts are experiencing lag nor service termination during the data collection periods. Users also had not experiencing any difficulties nor complaining drop in performance when using the remote hosts for routine tasks. This has shown that the configuration by the FUKURO has proved to be effective to continuously monitor remote hosts with minimum resource usage.

CONCLUSION

The Remote Host Resource Monitoring via Agent project successfully achieved its objective of developing a secure, reliable, and real-time monitoring framework for distributed systems. The proposed FUKURO (Fundamental Kernel Utilization Realtime Overseer) system integrates lightweight agents, a central server, and user-friendly web and mobile clients to provide continuous visibility into host performance. By employing WebSocket communication, the system supports near-instant data transmission between agents and the server, ensuring that metrics such as CPU, memory, disk, and network usage are updated in real time. The use of open-

source technologies including Python, Node.js, MySQL, and Flutter not only reduced development cost but also enhanced portability and scalability across multiple platforms.

Testing and evaluation confirmed that the system operated effectively under various workload conditions. Stress simulations using tools like *stress-ng* and *iperf* demonstrated the stability of the monitoring process, with minimal data loss and latency even when multiple agents were connected simultaneously. The alert mechanism functioned accurately by notifying users when predefined resource thresholds were exceeded, enabling administrators to respond promptly to potential performance issues. Both the mobile and web dashboards proved responsive and user-friendly, offering intuitive visualization through color-coded charts and synchronized updates.

In conclusion, the FUKURO system demonstrates that an agent-based approach can deliver efficient, scalable, and secure real-time monitoring for modern distributed environments. The integration of asynchronous data handling, lightweight agents, and cross-platform visualization addresses the limitations of traditional monitoring systems. Overall, the project provides a practical foundation for future research and development in intelligent monitoring solutions that combine automation, adaptability, and predictive capabilities.

Future Works

Although the FUKURO (Fundamental Kernel Utilization Realtime Overseer) system demonstrates strong potential as an effective monitoring solution, there remains considerable room for enhancement to further improve its scalability, compatibility, and analytical capability. Several key improvements are proposed to refine its overall functionality and long-term usability.

One of the most significant enhancements involves releasing agent applications for additional operating platforms. The current implementation relies heavily on Linux-based */proc* files for metric extraction, which limits the system's compatibility with non-Linux environments. To broaden its applicability, new agent applications should be developed for other popular platforms such as Windows and macOS while maintaining the same logical structure and communication protocols. This can be achieved by reusing most of the existing Python codebase and modifying only the model classes responsible for data extraction to align with the target operating system's resource access methods. Due to the object-oriented design (OOP) principles already adopted, this enhancement would require changes to only a few classes, enabling rapid adaptation and ensuring seamless interaction with the existing server architecture.

Another proposed improvement is the addition of a critical process snapshot feature. This feature would help users identify the root causes of unusually high resource usage by capturing the most resource-intensive processes during critical events. When a monitored metric exceeds a predefined threshold, the agent could automatically retrieve detailed process information, rank the processes in descending order of resource consumption, and submit the top five entries to the server. The data would be stored in a dedicated database table for future analysis. By implementing this module, users would gain deeper insight into performance anomalies and be able to take targeted actions to resolve system bottlenecks or prevent potential failures.

Finally, implementing a data retention and warehousing mechanism is essential to maintain long-term system efficiency. As the FUKURO database continues to grow due to frequent metric readings, query performance may degrade over time. To address this, a data retention policy can be established to aggregate historical readings into larger intervals as they age. For instance, converting data older than one month into hourly averages and data older than one year into daily summaries. Additionally, inactive or dormant nodes with no recent updates can be migrated into a separate data warehouse to reduce the load on active datasets. This approach optimizes database performance, minimizes storage usage, and ensures that users can still access historical insights without compromising system responsiveness.

ACKNOWLEDGEMENT

The authors would like to express gratitude to Fakulti Teknologi Maklumat dan Komunikasi (FTMK), Universiti Teknikal Malaysia Melaka (UTeM) for their invaluable support and resources provided throughout this research.

REFERENCES

1. Chen, L., Xian, M., & Liu, J. (2020). Monitoring system of OpenStack cloud platform based on Prometheus. Proceedings of the 2020 International Conference on Computer Science and Network Security. Retrieved from https://www.semanticscholar.org/paper/Monitoring- System -of- OpenStack-Cloud-Platform-Based-Chen-Xian/59a1fbf94860a0f16dfc1f1fad529b46df3a8719
2. Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629
3. Pragathi, P., & others. (2024). Implementing an effective infrastructure monitoring solution with Prometheus and Grafana. International Journal of Computer Applications, 186(38), 30–36. Retrieved from https://ijcaonline.org/archives/volume186/number38/pragathi-2024-ijca-923873.pdf/
4. Grafana Labs. (2023, September 21). Introducing agentless monitoring for Prometheus in Grafana Cloud. Grafana Blog. Retrieved from https://grafana.com/blog/2023/09/21/introducing- agentless-monitoring-for-prometheus-in-grafana-cloud/
5. Zhang, D. (2022, June 14). Pull or push: How to select monitoring systems? Alibaba Cloud Blog. Retrieved from https://www.alibabacloud.com/blog/pull-or-push-how-to-select-monitoring- systems 599007
6. Voutsas, F., Leivadeas, A., & Katopodis, S. (2024). Mitigating alert fatigue in cloud monitoring systems. Computer Networks, 217, 110543. https://doi.org/10.1016/j.comnet.2024.110543
7. Rawoof, F. M., Tajammul, M., & Jamal, F. (2022). On-premise server monitoring with Prometheus and Telegram bot. International Journal of Scientific Research in Engineering and Management, 6(4), 42–47. Retrieved from https://www.researchgate.net/publication/359965776 On-Premise Server_Monitoring_with_Prometheus_and_Telegram_Bot
8. Costa, B., Bachiega Jr, J., de Carvalho, L. R., Rosa, M., & Araujo, A. (2022). Monitoring fog computing: A review, taxonomy and open challenges. arXiv preprint arXiv:2206.07091. https://arxiv.org/abs/2206.07091
9. Das, R., Dey, N., & Bandyopadhyay, S. (2023). A review on fog computing: Issues, characteristics, and challenges. Journal of Parallel and Distributed Computing, 174, 103–115. https://doi.org/10.1016/j.teler.2023.100049
10. Ismail, A. A., Khalifa, N. E., & El-Khoribi, R. A. (2024). A survey on resource scheduling approaches in multi-access edge computing environments: A deep reinforcement learning study. Cluster Computing. https://doi.org/10.1007/s10586-024-04893-7
11. Wang, Z., Zhou, L., & Xu, T. (2024). Deep reinforcement learning-based methods for resource scheduling in cloud computing: A comprehensive review. Artificial Intelligence Review, 57(5), 6213–6248. https://doi.org/10.1007/s10462-024-10756-9
12. Hoque, M. M., Alam, M., & Rahman, M. (2024). Achieving observability on fog and edge computing using open-source tools: A testbed evaluation. arXiv preprint arXiv:2407.00035. https://arxiv.org/abs/2407.00035
13. Yang, T., Shen, J., Su, Y., Ren, X., & Lyu, M. R. (2022). Characterizing and mitigating anti-patterns of alerts in industrial cloud systems. arXiv preprint arXiv:2204.09670. https://arxiv.org/abs/2204.09670
14. Calderon, G., del Campo, G., Saavedra, E., et al. (2024). Monitoring framework for the performance evaluation of an IoT platform with Elasticsearch and Apache Kafka. Information Systems Frontiers, 26(6), 2373–2389. https://doi.org/10.1007/s10796-023-10409-2
15. Tundo, A., Mobilio, M., Riganelli, O., & Mariani, L. (2023). Automated Probe Life-Cycle Management for Monitoring-as-a-Service. arXiv preprint arXiv:2309.11870. https://arxiv.org/abs/2309.11870
16. Vervaet, A. (2023). MoniLog: An automated log-based anomaly detection system for cloud computing infrastructures. arXiv preprint arXiv:2304.11940. https://arxiv.org/abs/2304.11940
17. Chaudhary, H., & Sharma, G. (2025). AI-enhanced modelling of queueing and scheduling systems in cloud computing. Discover Applied Sciences, 7, Article 276. https://doi.org/10.1007/s42452-025-06755-2