

Development of A Control Delay Layer for Data Transmission Stability in Remote Patient Monitoring System

Ahmad Sidik Rofiudin¹, Tien Fabrianti Kusumasari², Sinung Suakanto³
^{1,2,3}Information System, Telkom University, Indonesia

Article Info

Article history:

Received Jan 03, 2026

Revised Jan 20, 2026

Accepted Feb 12, 2026

Keywords:

Backend System

Data Transmission Stability

Digital Health System

Load Testing

Remote Patient Monitoring

ABSTRACT

Remote Patient Monitoring (RPM) systems generate continuous health data that must be reliably processed at the backend to support timely clinical decision-making. Many real-world RPM deployments rely on synchronous request handling, which can lead to service degradation and request failures under high concurrency. This study designed and evaluated a Control Delay Layer (CDL) as an application-layer mechanism to improve backend request-handling stability in a web-based RPM system. The proposed mechanism decouples data reception from permanent storage through temporary buffering and deferred batch processing while regulating data submission behavior at the service level. System behavior before and after CDL implementation was examined using controlled load testing under identical scenarios. The evaluation employed service-level performance metrics, including request failure rate, response time distribution, and computational resource utilization. Experimental results show that the baseline monolithic system experienced an average request failure rate of approximately 14% under peak load, whereas no request failures were observed after CDL implementation. The CDL enabled system maintained consistent response-time behavior and stable resource utilization at higher concurrency levels. These findings demonstrate that backend-level request-handling control can effectively enhance system stability under high load conditions without requiring device-level modifications, providing a complementary approach for scalable and resilient digital health systems.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Tien Fabrianti Kusumasari,

Department of Engineering,

Telkom University,

Jl. Telekomunikasi No.1, Sukapura, Kec. Dayeuhkolot, Kabupaten Bandung, Jawa Barat 40257.

Email: tienkusumasari@telkomuniversity.ac.id

1. INTRODUCTION

Healthcare transformation in Indonesia over the past five years has moved massively towards digitization, driven by the urgency of treating non-communicable diseases (NCDs), whose prevalence continues to increase [1]. Riskesdes data shows an increase in cases of hypertension and other chronic diseases that require a continuous monitoring model without limiting patient mobility. In this context, the Internet of Things (IoT)-based Remote Patient Monitoring (RPM) system has emerged as a strategic solution that enables the independent and real-time collection of patients' vital signs, bridging the distance between patients at home and medical personnel at healthcare facilities. RPM is critical in managing diseases such as heart disease, preeclampsia, hypertension, diabetes, and other conditions that require continuous monitoring [2].

The success of RPM implementation is not only determined by the sophistication of the sensors, but also heavily depends on the reliability of the information technology infrastructure in ensuring stable data transmission [3]. The continuous flow of vital sign data requires a robust backend system. Even the slightest delay or failure in data transmission can have fatal

consequences for the accuracy of diagnosis and speed of medical intervention [4], [5], [6]. Therefore, the biggest challenge in RPM development today is no longer how to collect data, as sensor innovation and IoT integration have developed rapidly [7], [8] but rather how to manage data traffic to remain stable when the number of users increases dramatically [7], [9].

Various strategies have been developed to optimize data transmission in RPM systems, with the Edge Computing-based approach moving some computational processes closer to the data source, thereby reducing latency and reliance on central servers [9]. However, this approach requires additional edge infrastructure and management complexities that are not always easy to implement at scale. Scaling strategies, both vertical and horizontal, are commonly used to increase system capacity during load spikes. Although effective at increasing throughput, scaling tends to be reactive and can increase operational costs when the load fluctuates [10]. On the other hand, lightweight protocol optimization at the communication layer is also widely used, such as publish-subscribe using MQTT [11] and data compression and aggregation techniques [12], which have become reliable solutions for ensuring bandwidth efficiency and delivery reliability. Although these methods are very successful in terms of transmission speed and data integrity, most studies focus more on data transfer efficiency rather than controlling irregular request arrival patterns on the backend service side. These fluctuating request arrival patterns have the potential to disrupt data processing consistency and subsequent processes such as medical signal analysis and visualization. In RPM, this interval variation can affect the continuity of monitoring and the user experience of medical data visualization, especially under high load [7]. Delay or transmission-interval control mechanisms have also been used as management options to maintain system stability. This approach focuses on regulating the data transmission rate to avoid sudden overload of the system. The main advantage of this mechanism is its ability to reduce traffic spikes without aggressively increasing infrastructure capacity. However, delay control also has limitations, particularly regarding potential data delays that must be carefully managed to avoid interfering with real-time monitoring requirements in healthcare applications [13]. Consequently, mechanisms that explicitly regulate backend data admission behavior to maintain service stability under bursty RPM workloads remain limited.

The concepts of delay control and transmission intervals have been widely applied across fields beyond healthcare, such as real-time multimedia streaming, wireless sensor networks, and real-time analytics systems, to maintain smooth data flow and mitigate traffic fluctuations. In the field of digital health, similar approaches typically take the form of buffering mechanisms, transmission rate settings, or asynchronous processing in patient monitoring systems and telemedicine services to maintain service continuity when physiological data is transmitted periodically. However, in Remote Patient Monitoring (RPM) systems, most research still focuses on optimizing communication protocols, leveraging edge computing, or increasing infrastructure capacity. At the same time, delay-control mechanisms explicitly designed as a separate layer of logic to regulate data transmission behavior on the backend remain relatively limited. In contrast to these approaches, which primarily address transmission efficiency, infrastructure capacity, or network-level optimization, this study focuses on regulating backend request admission behavior at the application layer to maintain stable service processing under high and irregular RPM workloads.

Based on this review, a gap exists in research on backend data-handling behavior control to maintain the operational stability of the RPM system under high load conditions. Most existing approaches focus on increasing capacity or communication efficiency, while mechanisms that specifically regulate the rhythm of data delivery to remain consistent under high loads are still rarely discussed. To address this gap, this study proposes implementing a Control Delay Layer (CDL) as a data traffic control mechanism that stabilizes the inter-packet interval. CDL is designed as a control layer on the backend side of the application that regulates data flow before it is processed and stored permanently, without changing the data retrieval mechanism on the patient device side.

This study proposes an approach by developing a Control Delay Layer (CDL) as a clever middleware mechanism in the Application Tier [14]. CDL actively manages data traffic by integrating Context-Aware Rate Limiting logic that prioritizes data based on patient clinical urgency, and an Asynchronous Buffering mechanism that separates the data reception load from

the storage load. In addition, a Batch Processing strategy is applied periodically in the background to aggregate multiple data writing operations, thereby reducing connection overhead to the central database server [16]. This intervention optimizes server load distribution by running resource-intensive processes on the backend to avoid user-side responsiveness issues.

This study aims to design, implement, and evaluate the performance of data transmission control mechanisms in a web-based Remote Patient Monitoring (RPM) ecosystem. An experimental method using controlled load testing was used to compare system stability between monolithic baseline conditions and modified systems. The analysis was conducted from the perspective of request handling stability at the application layer using service-level performance indicators, namely request failure rate, response time distribution and consistency, and computational resource utilization. Through this approach, the study provides an empirical overview of the role of backend interventions in maintaining system stability under high load conditions.

To understand the research process, this paper is organized systematically. The introduction outlines the background, challenges, and research gaps that underlie this study. Next, the research methods and proposed system design are discussed in detail. The results of the implementation and evaluation of system performance are presented in the following section, followed by conclusions and recommendations for future development of the RPM system.

2. RESEARCH METHOD

This paper presents an experimental approach to evaluate the efficiency of the Control Delay Layer in maintaining data transmission stability in RPM systems. The research flow is designed to capture the overall behavior of the system from the initial conditions before intervention to performance evaluation after the implementation of the CDL mechanism. That is, it considers how data flow is managed when the system encounters simultaneous spikes in data transmission. Each stage of the research is arranged sequentially so that any differences in performance can be directly attributed to the presence of CDL, distinct from changes in the environment or other configurations.

2.1. Research Workflow

The research workflow follows the stages illustrated in Figure 1. The process begins with a requirement analysis to examine the characteristics of the existing RPM system, focusing on how patient data is received, processed, and stored through a synchronous execution path. While this approach performs adequately under low load, it tends to create request queues as concurrency increases, indicating potential overload points.

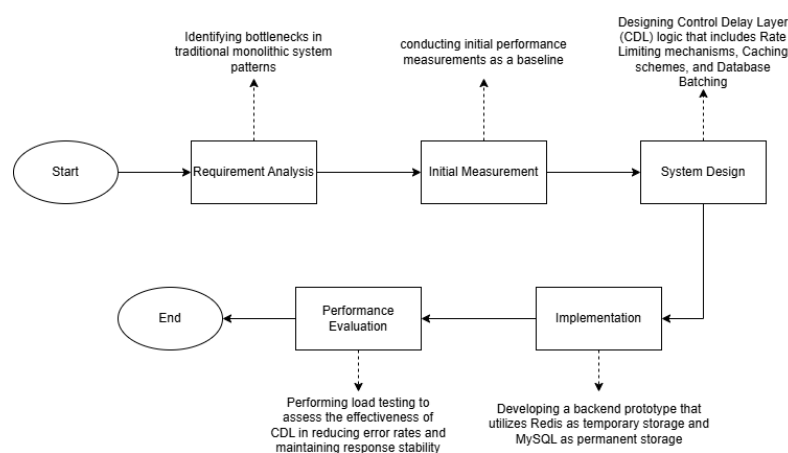


Figure 1. Research Workflow of the Proposed Study

The next stage is the Initial Measurement, which provides an overview of the system's initial performance before any intervention. Performance measurement is conducted through load

testing, gradually increasing the number of virtual users to simulate simultaneous data transmission from multiple devices. At this stage, several key performance metrics, such as request failure rates and response times, are collected as a baseline. The results of these initial measurements serve as a baseline for assessing the extent to which the proposed changes can impact system stability. Based on the results of the requirements analysis and initial measurements, the research proceeds to the System Design stage. Entering this phase, researchers design the Control Delay Layer (CDL) logic to control data flow on the application side. The design focuses on developing a rate-limiting scheme, a caching mechanism, and a gradual data processing strategy. This design is intended to regulate incoming data flow without changing the main structure of the existing system.

The Implementation stage involved developing a backend prototype in accordance with the established design. At this stage, CDL was integrated into the system using Redis for temporary storage and MySQL for permanent storage. The implementation focused on separating the data reception and storage processes so the system could still respond quickly to requests even under high load. The final stage is Performance Evaluation, which assesses the effectiveness of CDL implementation on system stability. Load testing is performed again using the same scenario as the initial measurement. The test results are then compared to assess changes in failure rates, response time consistency, and resource utilization. This evaluation serves as the basis for assessing the role of CDL in maintaining data transmission stability in the RPM system.

2.2. Existing System Characteristics and Performance Limitations

The Remote Patient Monitoring (RPM) system that is the subject of this study was initially designed to receive and process patient health data directly via an API. Each piece of data sent from the user's device is treated as a synchronous request, where data validation, processing, and storage are carried out in a single execution sequence. This approach is relatively simple and quite reliable when the system operates on a small scale [17]. However, these characteristics also make the system highly dependent on the speed of data storage on the server side. This dependency becomes an important factor when the number of requests increases simultaneously.

The concept of the existing RPM system and its main components are shown in Figure 2. In this configuration, the application and database layers are in a single processing environment that is directly connected. Each incoming data request must wait until the database write process completes before a response can be sent to the sender. This processing pattern places database operations as a critical part of the main service path. As a result, a slight delay in the input/output process can directly affect the system's overall response time.

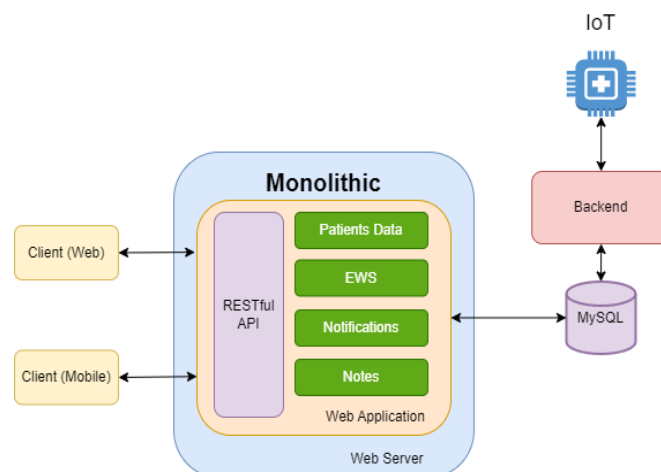


Figure 2. Conceptual architecture of the original RPM system [18]

This condition is even more noticeable when the system is running in a resource-constrained environment. Since the application server and database share the same computing resources, a spike in activity on one component can degrade the performance of the others. In high-load situations, writing data to the database slows application execution, causing new requests to

pile up on the server. This accumulation can lead to increased latency and request failures, especially when many devices send data almost simultaneously [19].

The limitations of the existing RPM system characteristics are also in line with the findings of several other RPM studies. A study by Fadilah et al. [20], highlights the need for a cloud-based backend architecture with high scalability to support the growth in the number of patients and devices, but does not discuss the mechanism for dynamic data traffic control at the application level. Meanwhile, Aniko et al. [21], through a Systematic Literature Review, focused on analyzing the needs of RPM system users, including functional and non-functional aspects, without formulating API integration and data flow management designs on the backend. Further research by Aniko et al. [22], emphasizes the importance of a user-centred design approach to improving the user experience for older adults on the RPM system, which still relies on a backend synchronization process that is not yet fully automated.

To better understand the impact of this processing pattern, initial performance testing was conducted on the existing system by gradually increasing the number of virtual users. This test referred to the performance evaluation results reported in previous research [18], the test scenario simulated simultaneous data transmission from multiple devices and observed the system's response. The test results showed that performance degradation began to appear at a relatively moderate load level. In the 50-user scenario, the request failure rate exceeded 50%, indicating the system could not handle the request queue efficiently under these conditions. A summary of the initial performance test results is presented in Table 1. The data shows a consistent pattern, where an increase in the number of virtual users is followed by a spike in the failure rate and a decrease in response time stability. The failures that occur are generally in the form of request rejections or response delays that exceed acceptable thresholds.

Table 1. Preliminary performance test results of the existing system [18]

Experiments	Virtual Users (VU)	Error rate	Request/second	Total request
1	10	24,94%	7,07	2,209
2	20	25,02%	13,84	4,320
3	30	24,56%	18,84	5,879
4	40	34,42%	19,75	6,138
5	50	52,66%	24,92	7,575

The characteristics of the existing system indicate limitations in the data transmission control mechanism. The synchronous processing pattern used makes the system vulnerable to traffic spikes, even though the number of users is not yet large [23]. This condition provides a strong basis for research to develop a more adaptive, high-load-resistant data management mechanism. The development of the Control Delay Layer aims to overcome these limitations without requiring significant changes to the existing system structure.

2.3. Testing Scenarios and Experiment Configuration

Performance testing in this study evaluated the behavior of backend services under simultaneous spikes in data requests. The approach used was controlled load testing in an isolated testing environment to observe backend service behavior under concurrent request pressure, instead of network-level transmission characteristics, so that the observed performance variations could be directly attributed to the system processing mechanism, particularly before and after the implementation of the Control Delay Layer (CDL). All experiments were run on a single local server with consistent hardware and software configurations. The test server used a laptop-class processor with 4 cores and 8 GB of main memory, without container-based resource restrictions. The operating system used was Windows-based, and all backend components, including API services and databases, were run on the same host to reflect a simple and commonly used implementation environment in the early stages of system development. The system backend is implemented using the Laravel framework, while the test load is generated using the k6 load testing

tool. This configuration is maintained across all test scenarios to ensure a fair comparison between the existing system and the system with CDL.

The testing scenario was designed as a stress test with a step-load pattern, in which the number of virtual users (VU) was gradually increased from 0 to 200 VU. Each load phase was run for a particular duration to allow the system to reach a stable condition before proceeding to the next phase. This pattern is used to observe the system's response at each stage of load increase and to assess its ability to maintain service stability as load increases. Details of the testing phases are shown in Table 2.

Table 2. Stress Testing Scenario Configuration

Test Phase	VUs Range	Duration
Ramp-up	0-50 VU	30s
Middle Phase	50-100 VU	60s
Peak Load	100-200 VU	60s (increase) + 120s hold
Ramp-down	200-0 VU	30s

Each virtual user is modeled as a single independent data source representing a single device or RPM application. For each request, the VU sends a single HTTP POST payload containing blood pressure measurement data with a uniform structure throughout the test. The payload includes the patient's identity, systolic and diastolic values, heart rate, measurement status, device identity, and timestamp. The size and structure of the payload are kept constant to ensure comparability of results.

To represent a worst-case scenario, no artificial think time or client-side retry mechanism is applied between requests sent by each VU, so that all requests are issued immediately according to the defined load pattern. This configuration simulates conditions where many devices send data in close succession, resulting in the backend receiving a bursty request pattern. All test scenarios, including load scripts, payload structure, timeout settings, and execution patterns, are applied identically to the existing system and the system with CDL and each test is repeated ten times to reduce the influence of random variations and obtain a more stable performance picture. Request timeouts were configured consistently across all test runs, and no adaptive backoff or retry logic was enabled during load execution.

3. RESULTS AND DISCUSSION

3.1. Proposed Control Delay Layer Mechanism

After initial testing, a data transmission system was designed for this study, which became intelligent middleware placed between the API Gateway and the database layer. This approach is quite different from the previous architecture, which processed data synchronously, while CDL decouples the data-reception process. Architecturally, CDL changes the communication pattern to a hybrid one: it remains synchronous on the front end to ensure responsiveness to IoT devices while remaining asynchronous on the back end to ensure data stability. The principle in the design of CDL is the separation of the data reception and storage processes. Initially, both processes ran under the same execution path, so any delay in the storage stage could directly affect service responsiveness. CDL breaks this dependence by giving priority to data reception while relegating storage to a background thread. This allows the server to keep responding quickly to incoming requests without waiting for the completion of the relatively expensive I/O operations.

Figure 3 presents the sequence diagram that shows the flow of interaction between components after implementation of CDL. A request is sent by the user's device with health data. The application layer receives this request first and forwards it to CDL directly. The CDL now does some basic assessment of the coming data, determining whether to process it further. If it receives data too frequently or if the data itself isn't in a critical condition, it doesn't forward it immediately to the database but sends it to temporary memory-based storage. Once the data is on this medium, a confirmation response is sent back to the sender without any significant delay.

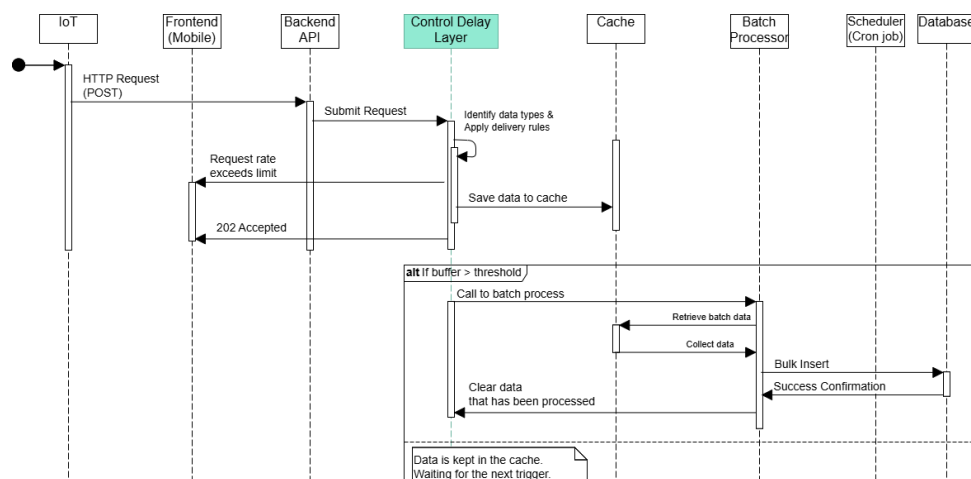


Figure 3. Sequence Diagram of Data Handling Process with Control Delay Layer

When patient health data is sent via the device, CDL acts as the initial gateway. Before the data is processed, the system applies a Context-Aware Rate Limiting algorithm. This logic works dynamically: if the data is detected as containing critical values, the system relaxes the transmission interval restrictions to prioritize that data. Conversely, under normal conditions, the interval is tightened to conserve resources. Data that passes validation is not written to the database immediately; it is temporarily stored in Redis to implement an In-Memory Data Store. Once the data is securely cached, the system immediately sends an HTTP 202 Accepted response to the client facing gateway, indicating that the backend service has successfully received the data for further processing. The 202 Accepted response indicates that the backend service has successfully received the data and is ready for further processing, even though the permanent storage process has not yet been performed. This technique keeps the data-reception path responsive without waiting for the completion of more intensive database I/O operations. The HTTP 202 response is treated as an acknowledgment of receipt at the application layer and is issued at the gateway level, where no automatic retry mechanism is triggered at the sensor device. As such, retry behavior is not part of the device-side data submission logic within the scope of this study.

In the background, the Batch Processor component works without interfering with the data reception process. This mechanism uses two triggers. The first is when the data stack in Redis reaches a certain threshold, and the second is when the Batch Processor works with a Scheduler (Cron Job) that runs periodically to sweep up any remaining unprocessed data. When one of the triggers is active, the Batch Processor retrieves all data packets from Redis and performs a Bulk Insert operation to the central database in a single transaction.

3.2. Implementation

The system was developed using a service-oriented architecture that separates data processing logic from the storage layer [24]. The backend was built using the Laravel Framework, which acts as an API Gateway and business logic orchestrator. To handle high transaction loads, this implementation integrates Redis as a high performance in-memory storage (cache) to hold temporary data queues, as well as MySQL as a relational database for long-term permanent storage [25].

The Control Delay Layer (CDL) implementation is translated into two main algorithms. Figure 4 handles the synchronous data reception logic on the controller side, validating clinical urgency and deciding whether to process the data immediately or store it in a buffer.

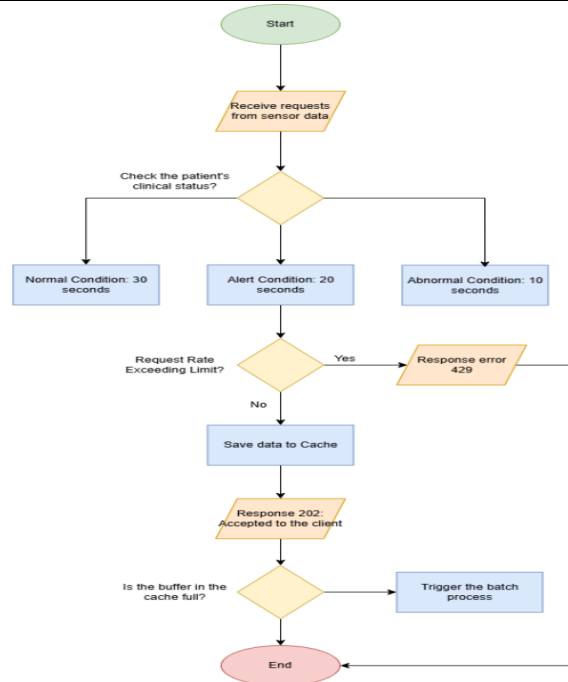


Figure 4. Dynamic data prioritization in the Control Delay Layer based on patient clinical status

In Figure 4, the system implements a dynamic prioritization mechanism. The AllowedInterval variable is determined automatically based on the patient's ClinicalStatus. If data arrives faster than the allowed interval, such as when there is data with a normal status, the system does not reject it, but instead transfers it to the buffering mechanism (Cache.Push) and returns an HTTP 202 Accepted response to the client without having to wait for the time consuming process of writing to the database to complete. This mechanism ensures that clients do not experience blocking or timeouts while the server is processing a long queue. Next is Figure 5, which represents the asynchronous batch processing mechanism. The scheduler triggers this process to move data from Redis to MySQL using the bulk insert technique to minimize database connection overhead.

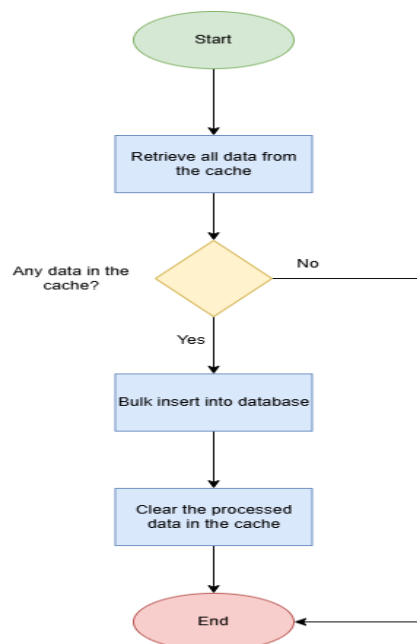


Figure 5. Batch Processing Strategy for Deferred Data Persistence

Figure 5 is designed to address the challenges of storage efficiency and data consistency in a distributed environment. This process implements a Batch Processing strategy to convert single

write operations into bulk insert operations. Flushing in Figure 5 implements two data security strategies. First, AcquireLock prevents race conditions in which two worker processes attempt to retrieve the same data simultaneously. The second security measure is a database Transaction wrapper that ensures atomicity. If a failure occurs during a bulk insert, the system can roll back and return the data to the cache queue (Cache.Push), so no patient data is lost.

3.3. System Implementation Result

The Control Delay Layer (CDL) has been successfully integrated into the backend system. Functional validation was performed by sending simulation data from IoT devices through the API endpoint. This test was used to ensure that the incoming data was processed according to the designed mechanism, a summary of the information returned by the system after data processing is shown in Table 3. The table presents the interpretation of the system response received by the client during blood pressure data transmission. Based on these results, the system confirms that the data has been received and placed in the buffer mechanism, accompanied by the clinical status of the evaluation results and the amount of data currently in the buffer queue.

Table 3. Example of HTTP API Response After Control Delay Layer Processing

Parameter	Description	Value
Response Message	Message status from the system after data is received	Health Data Saved & Buffered
Clinical Status	Category of clinical condition based on data evaluation results	High Hypertension
Buffered Data Count	Amount of data currently in the buffer	6
Device Identifier	Data sending device	HEM-7142T1
Data Handling Action	System handling of incoming data	Buffered (Deferred Storage)
Response Code	HTTP code returned	202 Accepted

Table 3 shows confirmation from the system that health data has been received and transferred directly to the temporary storage mechanism. The data represents transmissions that occurred within a close time interval, so it is still in the buffer queue. This response indicates that the data is not stored in the central database immediately but is first held for further processing. This mechanism proves that an asynchronous communication path has been established, allowing the system to send a success response to the client without having to wait for the database input/output (I/O) operation to complete. To complete the validation, examples of health data stored in the temporary storage mechanism are shown in Table 4. This table shows the format of the data held by the system before it is transferred to the central database via the batch processing mechanism.

Table 4. Sample Health Data Records Temporarily Stored in The Buffer

Patient ID	Systolic (mmHg)	Heart Rate (BPM)	Clinical Status	Timestamp
499	112	66	High Hypertension	2025-11-30 09:52
499	110	89	High Hypertension	2025-11-30 09:55
499	118	76	High Hypertension	2025-11-30 09:58

Table 4 shows that the system can store several blood pressure measurement records sent sequentially within a relatively short period. Several entries with the same patient identity are consistently recorded, both in measurement values and recording timestamps. This condition states that the data reception process does not overwrite itself even when the transmission is continuous. The incoming data is not immediately processed permanently but is stored in a temporary storage mechanism in a maintained order.

3.4. Performance Comparison and Analysis

System performance was tested to compare the backend service's behavior before and after implementing the Control Delay Layer (CDL) under identical load scenarios. The evaluation

focused on how the system handled bursty request patterns at the application layer when the number of simultaneous data transmissions increased. The stress testing scenario was conducted by gradually increasing the number of virtual users (VU) from 0 to 200, and repeating the experiment 10 times to obtain a consistent picture of service behavior at the service level. This approach enabled direct analysis of differences in system characteristics across two operating conditions, unaffected by variations in test configuration. In systems without CDL, performance declined clearly as load increased. The large gap between the median and 95th percentile response times indicates irregularities in request processing, where some requests experience very long wait times due to synchronous database operations.

After applying CDL, the system exhibited more consistent service-level behavior under load. Although response time variations persisted, they no longer escalated into extreme delays or request failures, indicating that burst-induced pressure was effectively absorbed at the backend service layer. The absence of request failures across all test iterations indicates that asynchronous buffering and deferred persistence effectively prevent overload conditions from propagating into service-level failures. Instead of seeking to eliminate variability in request arrivals, CDL ensures that it does not result in request rejection or timeouts. Response times also demonstrated a more controlled distribution, with lower average and p95 values than in the previous system. These changes mean the system can maintain service responsiveness even under high load.

Table 5. Summary of Service-Level Performance Metrics Averaged Over Ten Stress Test Runs

Metric	Monolithic System (Mean)	CDL System (Mean)
Failure Rate (%)	13,97	0,00
Average Response Time (ms)	975,11	618,90
Median Response Time (ms)	425,62	369,61
Response Time p (95) (ms)	4.086,39	1.732,39
Average CPU Usage (%)	13,93	17,11
Average RAM Usage (%)	66,59	61,94
VUs when Error Rate > 10%	66,90	0,00

A summary of the performance comparison between the monolithic system and the CDL-enabled system is presented in Table 5. The table shows direct differences in key metrics, including failure rate, response time, and resource utilization. In the system without CDL, the average failure rate was 13.97%, while in the system with CDL, it dropped to 0.00%. A decrease in latency is also observed in the p95 response time metric, which decreased significantly after CDL implementation. In terms of resource utilization, CPU usage in systems with CDL tends to increase, while memory usage decreases, indicating a shift in workload patterns driven by asynchronous processing.

To address variability across repeated experiments, performance stability was examined across ten independent stress test runs using distribution-based analysis in lieu of single point averages. Figure 6 presents boxplot visualizations of key service-level metrics, including failure rate, p95 response time, and the virtual user threshold at which the error rate exceeded 10%. This representation highlights the spread, median, and outliers observed across runs. In the monolithic system, the distributions show considerable dispersion, particularly in p95 response time and failure onset, indicating inconsistent behavior under identical load conditions. In contrast, the CDL enabled system exhibits tightly clustered distributions with no observed request failures and limited response time variability across all runs. This consistency across repeated experiments indicates that the observed performance improvements are not incidental to a single execution but reflect stable backend behavior under high concurrency.

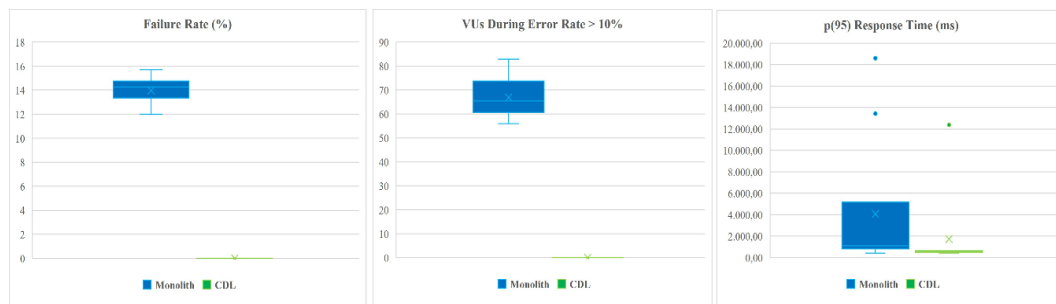


Figure 6. Distribution of key performance metrics across ten independent runs: (a) failure rate, (b) virtual user threshold at error rate >10%, (c) p95 response time

Figure 7 illustrates the system's behavior under load pressure without CDL implementation. When the average load reaches approximately 66.9 VU, the error rate begins to increase as the number of virtual users increases. This turning point reflects not only throughput saturation but also the system's resilience limit, at which tolerance to uneven demand spikes begins to collapse, triggering chain failures.

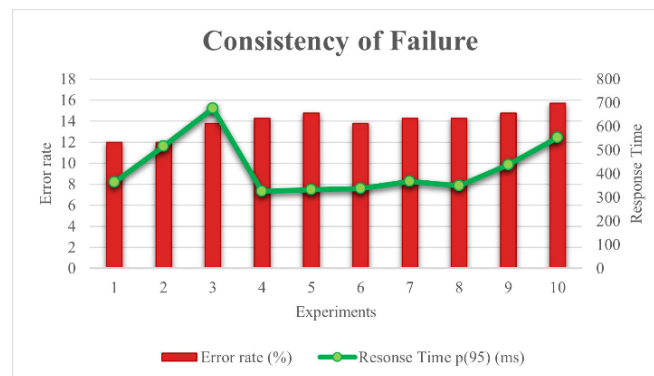


Figure 7. Stress testing results of the monolithic backend under increasing load

In contrast, the consistency of system performance with CDL is shown in Figure 8. The absence of an increase in error rate throughout the testing indicates that the system remains capable of handling incoming requests without experiencing service failures. This pattern contrasts sharply with the previous conditions and shows how the separation between data reception and permanent storage affects system stability. By using memory based buffering and background batch processing, traffic spikes no longer directly burden database I/O operations, allowing request queues to be managed more effectively.

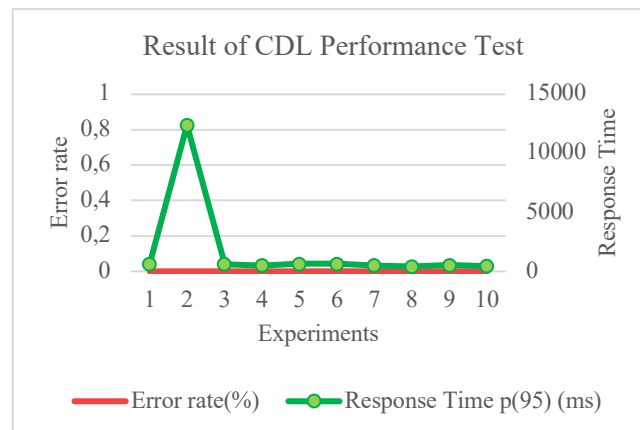


Figure 8. Service-level performance consistency of the CDL-enabled system under increasing load

Based on Figure 8, the relationship between the error rate and average response time shows a relatively consistent pattern in most test iterations. In the early iterations, there are noticeable fluctuations in the average response time, while the error rate remains very low. This condition indicates that an increase in response time is not directly followed by service failure, but rather reflects the system's initial dynamics in handling the given test load [26]. As iterations progress, the average response time tends to decrease and stabilize, without an increase in error rate.

This pattern shows that variations in response time and error rate do not always move in a linear manner [27]. In this context, the system still experiences changes in request processing time, but not to the extent of triggering service failure. Compared to a system without additional control mechanisms, response delays in a system with CDL do not escalate into error conditions but remain at a manageable level. These results demonstrate that increased response time does not necessarily compromise overall service robustness when appropriate backend control mechanisms are in place. By decoupling data reception from permanent storage, CDL prevents transient load spikes from cascading into systemic failures, thereby maintaining stable backend request processing under high concurrency.

This evaluation is intentionally framed at the backend application layer, as the proposed CDL is designed to regulate request handling behavior within the backend service rather than to smooth packet transmission at the network layer. Accordingly, the experimental analysis does not include direct measurements of packet-level jitter, transport-layer latency variance, or device-to-database end-to-end delay. These aspects were excluded from the scope of this study, as the proposed CDL operates exclusively within the backend without requiring modifications to patient devices, communication protocols, or network infrastructure. Instead, system stability is assessed using service-level indicators such as request failure rates, response time distribution, and resource utilization, which reflect the system's ability to absorb bursty workloads without propagating overload into service-level failures. Future studies may complement this work by incorporating network-level or end-to-end latency measurements to explore further the interaction between backend control mechanisms and transmission dynamics in Remote Patient Monitoring systems.

3.5. Resource Utilization and Efficiency

CPU and memory usage observations in this section were conducted in a single-node testing environment, where API and database services were run on the same host with a laptop-class processor (4 CPU cores and 8 GB RAM), using a consistent hardware configuration as described in Section 2.3. CPU and RAM percentage values are used to illustrate changes in system workload patterns before and after the implementation of the Control Delay Layer (CDL), instead of serving as indicators of absolute hardware efficiency across different environments. The evaluation of server resource utilization was conducted to verify that the integration of CDL maintains operational efficiency without overburdening the infrastructure. The analysis focused on comparing computational processing (CPU) and memory management (RAM) during peak load periods. A comparison of average resource usage between the conventional and proposed mechanisms is shown in Figure 9.

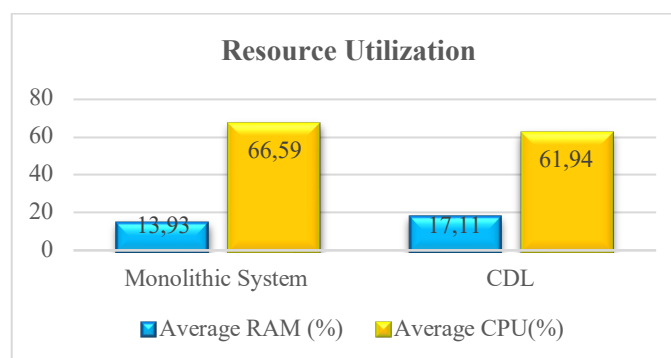


Figure 9. Average CPU and memory utilization during peak load conditions

Based on Figure 9, the CDL implementation recorded an average CPU usage of 17.11%, a slight increase over the conventional synchronization mechanism, which was 13.93%. In-depth analysis indicates that this increase is a sign of a shift in the system's processing load rather than inefficiency. The increase in processor usage correlates linearly with the high throughput ratio, with the system successfully processing 200 VUs without failure. From a cloud infrastructure management perspective, this phenomenon reflects resource optimization, in which the system maximizes hardware utilization to ensure service continuity. This observation is consistent with the findings of Shakil et al [28], which emphasizes that efficiency in cloud-based healthcare systems is measured by how effectively the CPU is utilized for valid medical data processing, as opposed to how low its utilization is when the system is idle or fails.

In contrast, memory management shows a different efficiency pattern. The average RAM consumption of the CDL implementation dropped to 61.94% compared to the initial mechanism, which reached an average of 66.59%. The efficiency shows that the superiority of the asynchronous model in managing minimal resources is proven. The asynchronous buffering strategy successfully clears the memory burden that generally occurs due to passive thread buildup and open database connections in synchronous models. The results confirm the proposition of Bhattacharya & Wood [29], that the control of active loads at the application layer level is much more efficient in saving memory compared to increasing hardware capacity. Moreover, as confirmed by Laigner et al [30], queue-based data management reduces microservice memory consumption that processes large volumes of data, in line with the efficiency obtained by Redis in this study.

However, this increase in efficiency has logical operational consequences, namely the transition from strong consistency to eventual consistency. The decoupling process introduces a momentary latency between data reception in the cache and its permanent persistence in the database, creating a window of time during which the data is not fully synchronized. Newman [31], in his discussion of distributed systems, asserts that in high-load scenarios, maintaining instant consistency often comes at the expense of service availability. In the context of RPM, El-Rashidy et al. [9], justify this approach, arguing that the top priority of critical monitoring systems is to ensure zero data loss and high availability, even if it means enduring synchronization delays of a few seconds.

3.6. Discussion

The Control Delay Layer (CDL) approach in this study is placed at the application layer as a mechanism for regulating data-reception flow, which distinguishes it from distributed computing approaches commonly discussed in the literature. Previous research by Baucas and Spachos [32] and El-Rashidy et al. [9] suggested using Fog and Edge Computing to reduce server load by processing data locally before sending it to the center. These approaches have proven effective in specific contexts, particularly in systems with mature distributed infrastructure, but typically require additional hardware deployment and more complex architectural management.

CDL is designed to complement existing edge-based solutions or infrastructure scaling strategies as a backend-level control mechanism that operates within existing architectures. By separating data reception from permanent storage, the system can regulate request admission behavior under high load, thereby reducing direct pressure on the database without introducing additional physical infrastructure layers. This positioning emphasizes CDL as an initial or intermediate optimization step, particularly in deployment contexts where architectural changes are constrained.

In the context of data priority management, CDL operates in a different control domain compared to network-level Quality of Service (QoS) mechanisms. Akbar et al [33] used a network-layer protocol to prioritize data for certain high-risk patients. While effective for managing traffic between nodes, such approaches lack visibility into the internal state of backend services, such as storage queues or database load levels. The findings of this study indicate that application-layer control can support prioritization decisions that consider both clinical urgency and backend processing capacity, a capability not directly addressed by network-layer QoS methods.

The implications of these findings relate to managing the scalability of RPM systems during both initial implementation and incremental development. Standard practice for handling increasing user numbers often relies on vertical scaling by increasing server CPU and memory capacity [34]. However, this study indicates that performance bottlenecks in RPM systems do not always stem from limited computing resources, but rather from poorly controlled data management patterns at the application layer. In this context, CDL can be viewed as an initial optimization step that helps the system utilize available capacity before deciding whether to scale the infrastructure. This view aligns with the findings of ZargarAzad & Ashtiani [35], who found that backend software optimization is part of a medium-term scalability strategy.

Although this study's results show an increase in service stability on the backend, several limitations remain. The evaluation focused on the backend application layer and did not include network-level measurements, such as packet jitter or end-to-end latency from the device to the database. All testing was conducted in a single-node environment with consistent configurations, so the results reflect the system's behavior in that context and do not represent a fully distributed scenario. This study also does not present a direct experimental comparison with other approaches, such as edge computing, fog computing, or automatic scaling mechanisms. These limitations do not detract from the study's main findings but indicate areas for further exploration to understand the interaction between backend control mechanisms and RPM system dynamics at a broader scale and across diverse environments.

4. CONCLUSION

This study aimed to design, implement, and evaluate a Control Delay Layer (CDL) as an application-layer mechanism for improving data transmission stability in Remote Patient Monitoring (RPM) systems under high concurrency. Experimental results show that the baseline monolithic system experienced request failure rates ranging from approximately 13% to 15% and began to degrade at an average load of around 66.9 virtual users. In contrast, the CDL-enabled system maintained stable request handling with zero observed request failures across ten test runs, even when the load was increased to 200 virtual users. This improvement was achieved by decoupling data reception from permanent storage through asynchronous buffering and deferred batch processing.

From a resource utilization perspective, the introduction of CDL resulted in a moderate increase in average CPU usage from 13.93% to 17.11%. In comparison, average memory consumption decreased from 66.59% to 61.94%, indicating a more controlled backend processing pattern that does not reflect excessive resource consumption. These findings suggest that backend-level request-handling controls can serve as an effective initial optimization strategy to improve RPM system stability without requiring immediate infrastructure scaling or device-level modifications. Future work may extend this evaluation to distributed environments, diverse health data types, and adaptive control mechanisms that consider both clinical urgency and network conditions.

REFERENCES

- [1] M. P. Carello, A. M. Spaccamela, L. Querzoni, and M. Angelini, "A Systematization of Cybersecurity Regulations, Standards and Guidelines for the Healthcare Sector," Apr. 2023, doi: 10.1109/ISI58743.2023.10297246.
- [2] P. Mishra and G. Singh, "Internet of Medical Things Healthcare for Sustainable Smart Cities: Current Status and Future Prospects," *Applied Sciences (Switzerland)*, vol. 13, no. 15, Aug. 2023, doi: 10.3390/app13158869.
- [3] M. Cao *et al.*, "Developing remote patient monitoring infrastructure using commercially available cloud platforms," *Front Digit Health*, vol. 6, 2024, doi: 10.3389/fdgh.2024.1399461.
- [4] I. G. M. N. Desnanjaya and I. M. A. Nugraha, "Real-time monitoring system for blood pressure monitoring based on internet of things," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 35, no. 1, pp. 62–69, Jul. 2024, doi: 10.11591/ijeecs.v35.i1.pp62-69.

- [5] R. Uddin and I. Koo, "Real-Time Remote Patient Monitoring: A Review of Biosensors Integrated with Multi-Hop IoT Systems via Cloud Connectivity," *Applied Sciences (Switzerland)*, vol. 14, no. 5, Mar. 2024, doi: 10.3390/app14051876.
- [6] S. Rajapaksha, W. Abhayarathne, S. Kumari, M. De Silva, and W. Wijesuriya, *A Mobile Application to Predict and Manage High Blood Pressure and Personalized Recommendations*. 2019.
- [7] M. Waleed, T. Kamal, T. W. Um, A. Hafeez, B. Habib, and K. E. Skouby, "Unlocking Insights in IoT-Based Patient Monitoring: Methods for Encompassing Large-Data Challenges," *Sensors*, vol. 23, no. 15, Aug. 2023, doi: 10.3390/s23156760.
- [8] S. D. Mamdiwar, A. R. Z. Shakruwala, U. Chadha, K. Srinivasan, and C.-Y. Chang, "Recent Advances on IoT-Assisted Wearable Sensor Systems for Healthcare Monitoring," 2021, doi: 10.3390/bios.
- [9] N. El-Rashidy, S. El-Sappagh, S. M. Riazul Islam, H. M. El-Bakry, and S. Abdelrazek, "Mobile health in remote patient monitoring for chronic diseases: Principles, trends, and challenges," Apr. 01, 2021, *Multidisciplinary Digital Publishing Institute (MDPI)*. doi: 10.3390/diagnostics11040607.
- [10] A. Rashid, F. A. Lone, and A. K. Chaturvedi, "Horizontal Dynamic Resource Scaling on Scheduling Interval in multi-tenant Cloud Computing," *International Journal of Advanced Science and Technology*, vol. 29, no. 11s, pp. 359–372, 2020, doi: 10.13140/RG.2.2.23020.80001.
- [11] A. Al Hanif and M. Ilyas, "Effective Feature Engineering Framework for Securing MQTT Protocol in IoT Environments," *Sensors*, vol. 24, no. 6, Mar. 2024, doi: 10.3390/s24061782.
- [12] G. Kousiouris, "A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments," *IEEE*, Jan. 2021.
- [13] M. Kumar *et al.*, "Healthcare Internet of Things (H-IoT): Current Trends, Future Prospects, Applications, Challenges, and Security Issues," May 01, 2023, *MDPI*. doi: 10.3390/electronics12092050.
- [14] S. Iranpak, A. Shahbahrani, and H. Shakeri, "Remote patient monitoring and classifying using the internet of things platform combined with cloud computing," *J Big Data*, vol. 8, no. 1, Dec. 2021, doi: 10.1186/s40537-021-00507-w.
- [15] S. N. Sahu, M. Moharana, P. C. Prusti, S. Chakrabarty, F. Khan, and S. K. Pattanayak, "Real-time data analytics in healthcare using the Internet of Things," in *Real-Time Data Analytics for Large Scale Sensor Data*, Elsevier, 2019, pp. 37–50. doi: 10.1016/B978-0-12-818014-3.00002-4.
- [16] O. Khattach, O. Moussaoui, and M. Hassine, "End-to-End Architecture for Real-Time IoT Analytics and Predictive Maintenance Using Stream Processing and ML Pipelines," *Sensors*, vol. 25, no. 9, May 2025, doi: 10.3390/s25092945.
- [17] F. A. Alaba, A. Rocha, H. A. Sulaimon, and O. Najeem, "IoT Applications and Challenges in Global Healthcare Systems: A Comprehensive Review," *Future Internet*, vol. 17, no. 12, p. 549, Nov. 2025, doi: 10.3390/fi17120549.
- [18] A. S. Rofiudin, T. F. Kusumasari, and S. Suakanto, "Backend of Heart Rate Detection System with Early Warning for Patients with Heart Disease," 2024.
- [19] M. Abbasi, M. V. Bernardo, P. Váz, J. Silva, and P. Martins, "Optimizing Database Performance in Complex Event Processing through Indexing Strategies," *IEEE*, vol. 9, no. 8, Aug. 2024, doi: 10.3390/data9080093.
- [20] S. Muhamad, I. Fadilah, and T. F. Kusumasari, "Backend Architectures for IoT Integration in Remote Patient Monitoring Systems : A Systematic Literature Review," 2024.
- [21] A. A. Rasendriya, T. F. Kusumasari, and S. Suakanto, "Application of User Experience Method to Determine User Requirements in Remote Patient Monitoring Systems: Systematic Literature Review," 2024.

- [22] A. R. Aniko, T. F. Kusumasari, S. Suakanto, and M. I. Fadilah, "User Experience Development in Elderly Heart Patient Monitoring System," *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, vol. 14, Nov. 2025, doi: 10.22146/jnteti.v14i4.18783.
- [23] J. H. Lee and S. H. Chung, "Adaptive Cell Scheduling and Negotiation Techniques for 6TiSCH Networks Under Bursty Traffic," *Sensors*, vol. 25, no. 5, Mar. 2025, doi: 10.3390/s25051418.
- [24] L. Chamari, E. Petrova, and P. Pauwels, "An End-to-End Implementation of a Service-Oriented Architecture for Data-Driven Smart Buildings," *IEEE Access*, vol. 11, pp. 117261–117281, 2023, doi: 10.1109/ACCESS.2023.3325767.
- [25] M. V. Privalov and M. V. Stupina, "Improving web-oriented information systems efficiency using Redis caching mechanisms," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 33, no. 3, pp. 1667–1675, Mar. 2024, doi: 10.11591/ijeecs.v33.i3.pp1667-1675.
- [26] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the Planet of Serverless Computing: A Systematic Review," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, Jul. 2023, doi: 10.1145/3579643.
- [27] H. Moni Prakash and A. F. Heckler, "Response time on math skills and its association with math skills accuracy and physics course grades," *Phys Rev Phys Educ Res*, vol. 21, no. 1, Jan. 2025, doi: 10.1103/PhysRevPhysEducRes.21.010144.
- [28] K. A. Shakil, F. J. Zareen, M. Alam, and S. Jabin, "BAMHealthCloud: A biometric authentication and data management system for healthcare data in cloud," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 1, pp. 57–64, Jan. 2020, doi: 10.1016/j.jksuci.2017.07.001.
- [29] R. Bhattacharya and T. Wood, "BLOC: Balancing Load with Overload Control In the Microservices Architecture," 2022.
- [30] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2021, p. 3348. doi: 10.14778/3484224.3484232.
- [31] Sam. Newman, *Building microservices : designing fine-grained systems*. O'Reilly Media, 2021.
- [32] M. J. Baucas and P. Spachos, *Fog and IoT-based Remote Patient Monitoring Architecture Using Speech Recognition*. IEEE, 2020.
- [33] M. S. Akbar, Z. Hussain, M. Sheng, and R. Shankaran, "Wireless Body Area Sensor Networks: Survey of MAC and Routing Protocols for Patient Monitoring under IEEE 802.15.4 and IEEE 802.15.6," Nov. 01, 2022, *MDPI*. doi: 10.3390/s22218279.
- [34] M. A. Razzaq, "Smart campus system using internet of things: simulation and assessment of vertical scalability," *Indian J Sci Technol*, vol. 13, no. 28, pp. 2902–2910, Jul. 2020, doi: 10.17485/IJST/v13i28.1034.
- [35] M. ZargarAzad and M. Ashtiani, "An Auto-Scaling Approach for Microservices in Cloud Computing Environments," *J Grid Comput*, vol. 21, no. 4, Dec. 2023, doi: 10.1007/s10723-023-09713-7.