Simanjuntak and Surantha
*Journal of Big Data* (2022) 9:108
https://doi.org/10.1186/s40537-022-00658-4

Journal of Big Data

RESEARCH Open Access

# Multiple time series database on microservice architecture for IoT-based sleep monitoring system

Eko Simanjuntak[1] and Nico Surantha[1,2*]

*Correspondence:
nico.surantha@binus.ac.id

[1] BINUS Graduate Program-Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia 11480
[2] Department of Electrical, Electronic and Communication Engineering, Faculty of Engineering, Tokyo City University, Setagaya-ku, Tokyo 158-8557, Japan

## Abstract

Monitoring health status requires collecting a large amount of data from the human body. The sensor can be used to collect data from the human body. The sensor transmits data for almost every second across the internet. The challenge of the health monitoring system is the massive amount of incoming data. Therefore, a system capable of sending, storing, analyzing, and visualizing vast amounts of data is required for health monitoring. A previous study proposed microservice and event-driven architecture. It also proposed a single database for all services and a relational database management system (RDBMS) for storing time series data, which might reduce the data transmission performance and reliability. This research intends to improve the monitoring system from the previous study to accommodate a greater throughput, faster database read and write operations, and a more reliable system design. The improvement consists of multiple changes in system architecture and technology. A multi-database is proposed in the system architecture to improve system reliability. Time series database and Message Queue Telemetry Protocol (MQTT) server are proposed as an upgrade on technology. As a result, the proposed system throughput is 2.43 times faster than the old system. On database performance, the new system's database write speed is 20.95 times faster and the database read speed is 1.64 times faster than the old system. The proposed system also achieves better scalability, resilience, and independence.

**Keywords:** IoT, Time series database, Health monitoring, Microservice, MQTT protocol

## Introduction

A Health monitoring system is a collective component used to receive, measure, analyze, and visualize health data. Modern health care systems consist of technical aspects such as wearable devices, cloud servers, and the internet of things (IoT) [1]. A Health monitoring system can use to monitor a few things, e.g., heartbeat, body temperature, and blood pressure.

One of the challenges of building a health monitoring system is a large amount of data [2]. Every day, medical data is collected from a large number of patients. Furthermore, each patient can wear multiple sensors simultaneously, increasing the data collected. Pereira et al. used the First-In First-Out (FIFO) approach to help the system

accept a large amount of data [2]. The FIFO approach creates a queue of data that comes to the system. The sensor keeps pushing the data to the queue, and another system picks up the data from the queue and then stores it in the database. This approach helps the system constantly receive data regardless of whether the data is already stored in the database.

The second challenge faced by Pereira et al. was the excessive amount of time spent storing received data in the database [2]. Fifty insert database queries require more than a second to complete, while 500 insert database queries require more than 10 s. They solve this issue by tuning database configuration to flush transaction logs to file once per second.

Another challenge faced by Pereira et al. is that the Hyper Text Transfer Protocol (HTTP) is too slow to deliver data at a high frequency. When new data arrive, HTTP must open a new connection and close it once the data has been acknowledged. To address this issue, they create a long-open connection so that the server and client do not have to reopen the connection for each data. Additionally, they add a delimiter to the request data, so the server knows when a request has concluded.

Surantha et al. proposed a design of high performance and resource-efficient IoT-based sleep monitoring system [3]. There are three main parts of the proposed design, i.e., data acquisition, cloud server, and monitoring application. Data acquisition uses sensors to detect and steam data to a cloud server. While cloud servers do data operations, i.e., classifying, quantifying, and storing data. On the other hand, the monitoring application is a web application that shows the monitoring result. The system design follows microservices and event-driven architecture. This design increases the response time by 55.85% and throughput by 34.76%.

The cloud server part of the monitoring system proposed by Surantha et al. [3] is designed following microservices architecture. Microservice architecture is an approach to system design that emphasizes creating smaller services that handle domain-specific operations. The services have intercommunication to exchange information. That is why on the cloud server part, there are five services, one database, and one message broker. Four services access the single database. This architecture makes the services not fully independent. Other services must adopt the change if one service requires a schema change. On the data acquisition part, the sensor publishes time series data. This data is received and stored in a database on the cloud server. The database type used in the proposed design of Surantha et al. is a relational database management system (RDBMS) [3]. It used PostgreSQL as the database.

The database is one of the critical components of an application [4]. Database performance can impact the application performance directly. Performance inefficiencies can be caused by poor physical database design, coding patterns, and the lack of caching [5]. Performance inefficiencies happen in all basic database operations like data creation, deletion, renewal, and retrieval.

Database performance issues can also happen because the database type is unsuitable for the application use case [6]. There are a few types of databases, e.g., relational databases, non-relational databases, key-value databases, graph databases, and time series databases. Each type of database has its purpose; for example, relational databases store data that have schema relations, and key-value databases store paired data.

Recently time series database (TSDB) was introduced as another option to store data. TSDB's target is to enable summary entries of data in a time interval rather than individual data. TSDB is also designed for high-frequency logging events [7]. TSDB also supports mathematical query functionality to help users do common time-based analysis [8].

There is room for improvement in the research done by Surantha et al. [3]. This research addresses the following weaknesses in the Surantha et al. study: (a) it uses HTTP, which has a longer response time than Message Queue Telemetry Protocol (MQTT); (b) it uses RDBMS, which has a longer query execution time than a time series database; and (c) it uses a single database for all services, which results in the loss of microservices' core properties, namely scalability, resilience, and independence. Improvement can help increase system throughput, speed up database read and write operation, and enhance system design. Based on improvements that can be made, the targets of this paper are to propose new system architecture, which is specified as follow:

1. Utilizing the MQTT protocol to achieve higher throughput. Implementing a MQTT server as a sensor gateway will increase the system throughput [3]. MQTT server handles the long-open connection to minimize the overhead of handling connection creation and deletion between client and server.
2. Utilizing the time series database to achieve faster database read and write performance.
3. Splitting a single database into multiple databases to improve scalability, resiliency, and independence.

The rest of the paper is arranged as follows: section "Related concepts" discusses all related terms and keywords used in this paper. Section "Related works" discusses all the previous studies in this field. Sect. "Proposed Methods" presents method and system architecture proposed in this study. In section "Results and discussion", the steps, results, and analysis are presented. Finally, Section "Conclusions" provides the conclusion of the paper.

## Related concepts

### MQTT

MQTT protocol runs over TCP/IP, which supports various data types like text, JSON, and XML [9]. This ability make MQTT have higher portability compared to other protocols. MQTT used the publish-subscribe pattern, meaning the client publishes a message to a particular topic, and each server that subscribes to the topic will receive the message [10].

MQTT has a small packet size which makes lower power consumption. MQTT is designed for IoT devices that have constrained resources. MQTT is lightweight and only requires a small bandwidth for data transmission [10].

### HTTP

HTTP is a primary protocol to transmit various data like text and images for everyday internet use. HTTP is widely used on web applications and mobile applications. HTTP

uses a request-response pattern, which means the client sends data to the server then the server sends back a response to the client [11]. HTTP has an acknowledge mechanism to ensure the server's response to the client is delivered well [12].

Other than web and mobile applications, HTTP is also used for IoT. HTTP can transmit sensor data to a server and from a server to other servers. HTTP is also beneficial when it comes to visualizing monitoring data.

#### MongoDB

MongoDB is a document-oriented database system. MongoDB is used to store unstructured data [13]. MongoDB is suitable for unstructured data because it supports schemeless, where we do not need to define the data structure first. MongoDB can handle the data structure changes on the fly. MongoDB will benefit IoT if the data structure is hard to define or there are possibilities to change.

#### Firebase

Firebase is a product offered by Google. Firebase offers some features like a database as a service, cloud messaging, hosting, and analytics [14]. By using Firebase, the user does not deploy and maintain their server. Firebase will handle the server maintenance.

Firebase database as a service called Cloud Firestore. Cloud Firestore is a NoSQL database that supports schemeless data [15]. Cloud Firestore supports real-time data sync between the server and the client. It is helpful to visualize IoT data for real-time data updates.

#### InfluxDB

InfluxDB is a time series database [16]. As a time series database, InfluxDB offers a bunch of features for managing time series data, like aggregating and grouping. Other than that, InfluxDB also supports mathematical queries to help analyze data. InfluxDB also offers a built-in monitoring dashboard to visualize the stored data [17]. InfluxDB uses a push-based approach to collect data, which means the data source actively transmits its data to InfluxDB.

## Related works

IoT in health monitoring systems has been widely used and studied. Researchers studied a few aspects of IoT as a health monitoring system. Atmoko et al. compared the MQTT and HTTP protocols for IoT data acquisition [9]. They designed a scenario to test the capability of each protocol to transfer data from hardware to server and store it in the MySQL database. The research results show that the MQTT protocol can send data up to 6 times faster than the HTTP protocol. This performance difference occurs because MQTT has a smaller header data size than HTTP.

Al-Joboury et al. experimented with analyzing the packet loss aspect between MQTT and HTTP [18]. Data loss is defined as the number of packets of data that fail to reach the destination when the data travel through the network. The result indicates that HTTP experienced approximately 49.7 times more data loss than MQTT. MQTT had a lower data loss because MQTT has a lower handshake and lower protocol data unit (PDU). PDU is a group of information added or removed by layers of the open-system

interconnection (OSI) model. PDU is used to communicate and exchange information between OSI layers.

Sasaki et al. experimented with analyzing the required network resource between MQTT and HTTP. The experiments found that MQTT requires fewer networks resource than HTTP [19]. This result is also related to the previous experiments that show MQTT had smaller header data, a lower handshake, and a lower PDU. These conditions make MQTT requires fewer network resource.

Musa et al. also accomplished an experiment to compare RDBMS and TSDB [20]. In the experiment, they compared PostgreSQL and InfluxDB. PostgreSQL is an RDBMS and InfluxDB is a TSDB. They analyze the performance of aggregation and grouping operations. InfluxDB had a lower execution time, around 46.87–77.62% for the aggregation operation compared to PostgreSQL. InfluxDB had a lower execution time, around 24.76–63.20% for the grouping operation compared to PostgreSQL. This result represents the main advantage of using a time series database for data that need to be aggregated and grouped.

Hou et al. designed and implemented an IoT system in the cloud server [21]. Data transmission from the IoT component to the cloud server can use HTTP and MQTT protocols. The HTTP protocol transmits data between the server and the end-user application, like web and mobile applications. On the other side, the MQTT protocol is used to transmit data between the IoT component to the cloud server. To store the data, they use the Redis cluster. Redis is an in-memory key-value datastore. The write and read speed will increase by storing the data in memory. To improve the Redis reliability, they used a Redis cluster. Redis cluster is a group of Redis servers that can perform data sharding and replication.

The health monitoring system proposed by Shashidar et al. used serverless technology [22]. Serverless technology provides simplicity where the user does not need to maintain the server and delegate to the serverless provider. They are using Firebase as the serverless provider. Firebase provides Firestore as the database service. Firestore is a NoSQL database that offers data synchronization between server and client regardless of the network latency. Using this serverless technology, they do not need to focus on the underlying hardware and system runtime. Scalability, reliability, and availability are the responsibility of the provider. The researcher can focus on what they are building.

Another research on IoT is the fall detection system proposed by Lacalle et al. [23]. They have created IoT wearable belts that can detect if someone is falling. Integrated IoT devices planted on the belt to collect and publish the data to a gateway. The gateway transmits the data to the cloud server using MQTT as a communication protocol between the wearable belt and the gateway. The gateway is deployed using Docker. The result of using Docker is that CPU usage does not increase so fast because the Docker container can share the underlying hardware resources with other processes. On the cloud server side, MongoDB is used as the data store. MongoDB is a NoSQL database that uses a document-oriented paradigm and stores data in JSON format.

Yang et al. also proposed a wearable monitoring system for smart healthcare [24]. There are three system parts, i.e., a monitoring node, a cloud server, and a monitoring dashboard. The monitoring node is used to collect data from the human body. It consists of a sensor, controller, Wi-Fi module, and power supply. Then the collected data is

transmitted to the cloud using MQTT. Besides the MQTT, the cloud also provides an HTTP server to provide data to the monitoring dashboard. The cloud server also has one storage server to keep all the data. They used Redis to store the data instead of RDBMS. RDBMS read and write speed can be a bottleneck for cloud performance. Using Redis, database read and write, and flexibility can be improved.

IoT is also utilized for traffic monitoring and vehicular accident prevention [25]. Researchers use the mobile sensor as a tracker device and install the tracker on the vehicle. The tracker is used to detect the vehicle position, speed, acceleration, and movement-related data. The tracker streams the data to the cloud server. The cloud server uses the data to analyze if there is any sudden traffic slowdown. The alert traffic slowdown alert sends to the driver's mobile application. The system uses microservice architecture and uses Docker to deploy the microservices. The system also uses two types of databases; those are SQL database and MongoDB database. SQL database is used to store OpenGTS data, an open-source project web-based vehicle tracking sent from the tracker. They also use OpenStreetMap, an open-source project, to visualize the traffic. OpenStreetMap requires GeoJSON data to visualize the traffic. Since the OpenGTS data is stored in the SQL database, they need to convert the data to JSON data and store it in MongoDB. The result of the research is that the system can parse and insert 100,000 OpenGTS data to MongoDB in 6 s. The system can also fetch 300,000 data from MongoDB in 30 ms to visualize the traffic.

Silva et al. propose monitoring and controlling systems used on industrial devices [26]. The system is split into three parts: the sensor device, the gateway, and the cloud server. The sensor device works to gather the data. Then the collected data is transmitted to the gateway. The gateway transforms the data to the format required by the cloud server. Then the transformed data is transmitted to the cloud server. The cloud server consists of a message broker, database, frontend server, and event manager. The message broker streams data from the gateway and user devices. Message brokers also do data cleaning, transformation, and queuing the message to be processed later by the other parties. Then the data from the gateway is stored on Cloudant, a managed NoSQL database service from IBM. The frontend server is used for visualization and control. There are two types of frontend: web-based and mobile applications. Besides can visualize the monitored data, the application also can use to control the system. The event manager is a component used for analytics tasks and fire alerts if conditions are satisfied.

An automatic sleep monitoring system proposed by Jiang et al. used MySQL RDBMS [27]. There are two types of data tables: entity data tables and derivative data tables. Entity data tables are used to store user information, device information, and channel information. Derivative data tables are used to store analysis data that are streamed from sensors like heart rate, body movement, and respiration rate. There are ten tables placed in one database.

Table 1 shows the summary of the related study. Based on related studies above, no research uses a time series database. Moreover, no research implements a database per service (multiple databases). On the other hand, three pieces of research compare MQTT and HTTP based on a few aspects, like transmitting time, data loss, and required network resources [9, 18, 19]. The three papers insisted on the superiority of MQTT over HTTP in the three evaluated aspects. Therefore, the authors propose a multiple

**Table 1** Summary of related studies

| Authors | Protocol | Database |
|---|---|---|
| [3] | HTTP | SQL database (PostgreSQL) |
| [21] | Combination of MQTT and HTTP | Key-value database (Redis Cluster) |
| [22] | HTTP (Firebase) | NoSQL (Firebase) |
| [23] | HTTP | Document-based database (MongoDB) |
| [24] | Combination of MQTT and HTTP | Key-Value Database (Redis) |
| [25] | HTTP | Combination of SQL and document-oriented database (MongoDB) |
| [26] | Combination of MQTT and HTTP | NoSQL database |
| [27] | HTTP | SQL database (MySQL) |

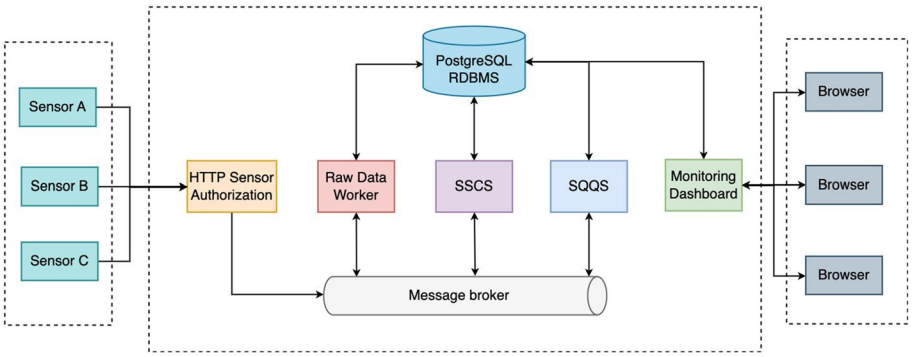


**Fig. 1** System architecture proposed by Surantha et al. [3]

time series database and use MQTT for an IoT-based sleep monitoring system. The aspect that is evaluated between MQTT and HTTP is the throughput. MQTT delivers lower transmitting time, data loss, and required network resources than HTTP, and we expect it to reflect on the throughput aspect. Database read and write performance are also evaluated. Finally, system design is evaluated using a few scenarios to evaluate scalability, resilience, and independence.

## Proposed method

Figure 1 shows the system architecture proposed by Surantha et al. [3]. The system comprises five services, one database, and one message broker. PostgreSQL is used as the database that is accessed by all services.

Three weaknesses are raised in traditional architecture, as explained below.

1. The first weakness is that it uses HTTP protocol on the sensor gateway. HTTP response times are 4.7 times higher than MQTT on cloud-based IoT systems [18]. HTTP uses a single connection for a single request which causes HTTP needs to open and close connections between requests. This mechanism causes response time degradation on HTTP due to the high overhead of maintaining connections for each request.
2. The second weakness is that it used PostgreSQL RDBMS to store time series data. PostgreSQL was found to have a longer execution time of aggregation with time

interval constrain, around 46.87–77.62% longer than InfluxDB [20]. This result represents the main advantage of a time series database in aggregating and grouping massive datasets.

3. The third weakness is using a single shared database to store data of five services. Using a single shared database on microservice architecture is considered an anti-pattern, where the microservice loses its core properties which are scalability, resilience, and independence [28]. Stec also explained that a single shared database could be a single point of failure. If a failure happened in a single database, it would impact all services connected to the database because there is no other database that could serve as a backup for the primary database. The failure can cause the service to stop working, not serve any requests, and cause data loss. Data loss happens because the services cannot receive and store the request in the database [28].

Figure 2 shows the proposed architecture. The system consists of five services, four databases, one MQTT server, and one message broker. Those services are sensor authorization, raw data worker, sleep stage classification service (SSCS), sleep quality quantification service (SQQS), and the monitoring dashboard. The database is separated for each service. PostgreSQL is used for the sensor database, and InfluxDB is used for the raw database, SSCS database, and SQQS database. The MQTT server is the main gateway of data coming into the system. Service-to-service communication will be asynchronous using a message broker.

Here is the flow of the proposed system. Incoming data arrives at the MQTT server first. Then the sensor authorization service fetches data from the MQTT server and checks whether the data came from a registered sensor. The authorized data is then delivered to the message broker. A row data worker fetches raw data from a message broker and then stores it in a raw database. SSCS also fetches data from the message broker and then classifies the data. The classified data is sent back to the message broker and fetched by the SQQS. SQQS quantifies the sleep quality and stores it in the SQQS database. Finally, the monitoring dashboard fetches data from the SSCS and the SQQS database and visualizes it on the web browser.

There are three benefits offered by this proposed architecture, as explained below.

1. MQTT servers have a different way of handling connections compared to HTTP. HTTP uses a single connection for one request. After the request is made, the con-

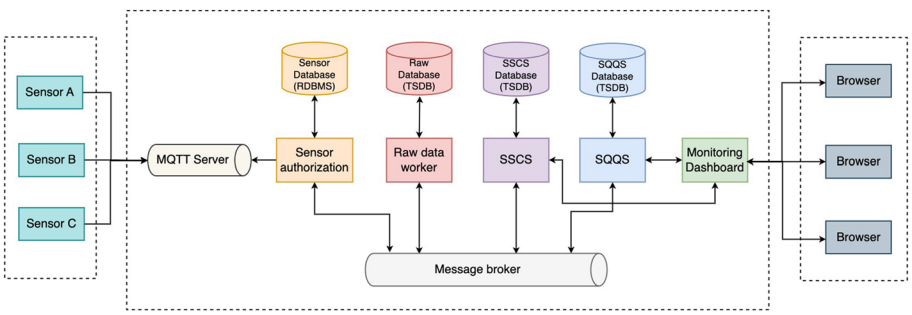


**Fig. 2** Proposed architecture

nection will be closed. On the other hand, MQTT uses a single connection for multiple requests. The MQTT client and server only need to establish a connection once and will use the connection to transmit the request. This mechanism is beneficial for transmitting frequent data.

2. InfluxDB offers a bunch of built-in features suitable for handling large amounts of data for reading and writing operations. InfluxDB can write data in batches that minimize network overhead for a write operation. A single connection can be used to write multiple data. Another feature of InfluxDB is that it has data compression to speed up writing and reduce network bandwidth. InfluxDB is a columnar database for the read operation, where data is stored on a column basis. Storing data per column helps increase read speed because when data is grouped in columns, it will not retrieve data that is not needed.
3. Using a multi-database helps to reduce coupling between components, which leads to better scalability, resilience, and independence. Having a database per service enables the developer to scale the database that needs to be scaled in the future. It also improves resilience because failure in one database will not impact other services. Another benefit of having a single database per service is the database independence of others. Changes in a database will be isolated and do not impact other databases.

## Results and discussion

### Evaluation scenario

#### *Throughput evaluation*

Evaluating system throughput means counting how many requests can be received in a time range. In this evaluation, throughput is calculated by counting how many requests that sensor gateway can receive and acknowledge in 1 s. Evaluation is conducted by doing load testing using a simulated sensor. Figure 3 shows the system topology for throughput evaluation with a simulated sensor and a sensor gateway. The testing scenario is applied to both the traditional and proposed systems. The evaluation scenario is detailed below.

1. The simulated sensor uses Apache JMeter.
2. The simulated sensor sends 28,800 requests and 57,600 requests. This number is calculated by assuming one sensor sensing ECG data every second for 8 h. The

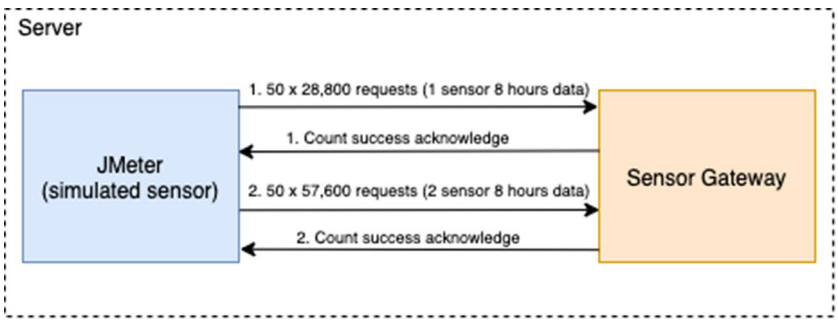


**Fig. 3** System throughput evaluation topology

average sleep time for adults is 8 h [29]. Total ECG data of 8 h running a sensor is $8 \times 60 \times 60 = 28{,}800$ and for two running sensors is 57,600.

3. A successful load test's expected condition is that all simulated sensor requests should be successful.
4. The load testing is done 50 times for each amount of data.

#### *Database read evaluation*

The main data for database reading are sleep stage data and sleep quality data. Sleep stage data are generated by SSQS using raw ECG data from the message broker. Sleep quality data are data generated by SQQS by using sleep stage data.

There is a difference in the amount of sleep stage data used for the database write test. The ratio of total data is 1:960. It means 1 row of sleep stage data on the proposed system equals 960 rows of sleep stage data on the traditional system. The algorithm that classifies the sleep stage will chunk all the given data by 30. Each chunk is used to define a single sleep stage. For example, if there are 300 ECG data, the algorithm chunks it into ten groups of ECG data, generating ten sleep stage data. If a sensor runs for 8 h, it will generate 28.800 ECG data, generating 960 sleep stage data ($28{,}800/30 = 960$). In the traditional system, each sleep stage data is stored in one row to have 960 rows for 28,000 ECG data. The proposed system's sleep stage data is stored in one row for 28,000 ECG data. Figures 4 and 5 show the system topology for database read evaluation on the traditional and proposed systems, respectively. The evaluation scenario is detailed below.

1. Fill the SSQS database with 9600 messages and SQQS database with 1000 messages.
2. Start the service that provides the REST API that the simulated client will access. In the traditional system, a web application gateway is started, as well as the SQQS and SCCS in the proposed system.
3. Simulate ten clients to send 100 requests each.
4. The expected condition of a successful read test is that all the requests got HTTP response code 200.
5. The load testing is done 50 times. Note how long the time is needed to finish each iteration.

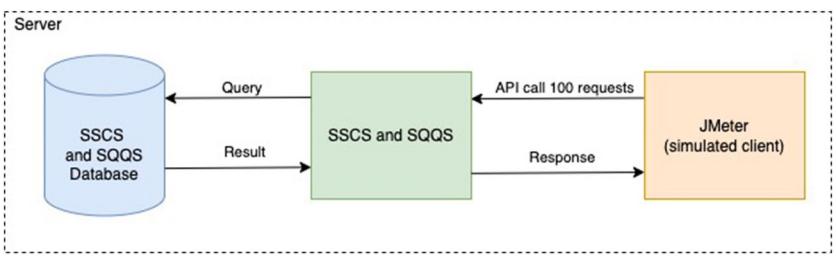


**Fig. 4** Database read evaluation topology on the proposed system

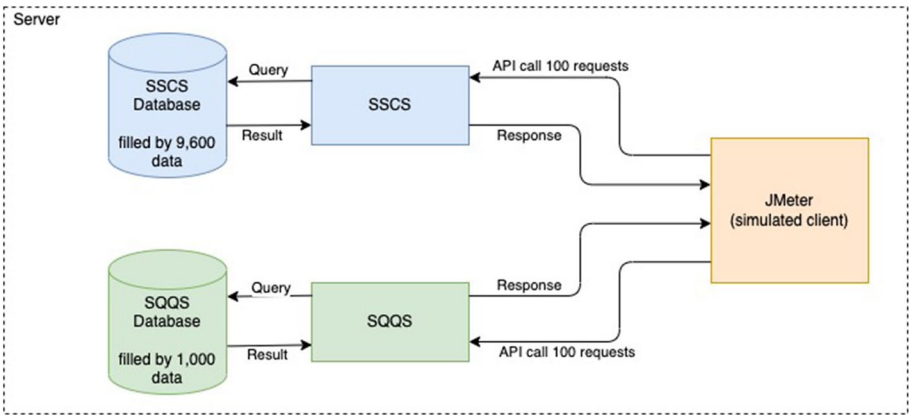


**Fig. 5** Database read evaluation topology on the traditional system

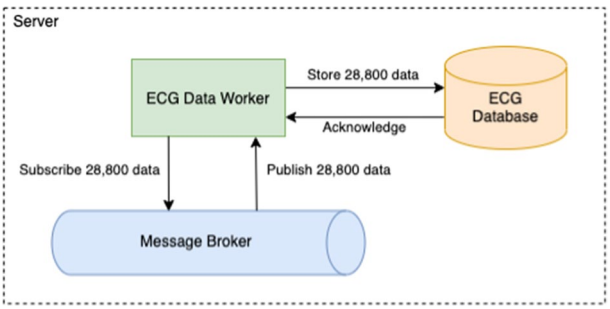


**Fig. 6** Database write evaluation topology

***Database write evaluation***

The data that is used for database write is raw ECG data. The raw ECG data is pulled by the raw data worker and stored in the raw data database. The total time to pull and store the raw data is counted as a write performance metric. Figure 6 shows the system topology for database write evaluation. The evaluation scenario is detailed below.

1. Fill the message broker with 28,800 ECG data.
2. Make sure the amount of data on the database is 0.
3. Start the consumer to consume and store the data in the database.
4. The expected condition of a successful write test is that all the data fetched from the message broker should be stored in the database.
5. The load testing is done 50 times. Note how long the time is needed to finish each iteration.

## Evaluation result

***Throughput evaluation***

Evaluating system throughput means counting how many requests can be processed in a second. Throughput evaluation is done by doing load testing and using the simulated sensor. The testing scenario is applied to both the proposed and traditional systems. Figures 7, 8, and Table 2 show the result of throughput evaluation.

The higher number of requests per second means the system achieves better throughput performance. The proposed system throughput evaluation shows that the average number of requests per second that the system can handle is 2652.34 requests/s for 28,800 requests and 2692.01 requests/s for 57,600 requests. In another side, traditional system average throughput is 1049.25 requests/s for 28,800 requests and 1106.99 request/s for 57,600 requests. It means the proposed system can handle 1.43 times more requests per second compared to the traditional system.

The proposed system throughput increased because MQTT has a lower overhead on managing connections. MQTT creates a new connection when a client is connected. MQTT reuses the single connection during data transmission. On the other side, HTTP creates connections for each request. HTTP manages connections as much as the requests and causes a higher overhead. The higher overhead on managing connection gives HTTP lower throughput than MQTT.

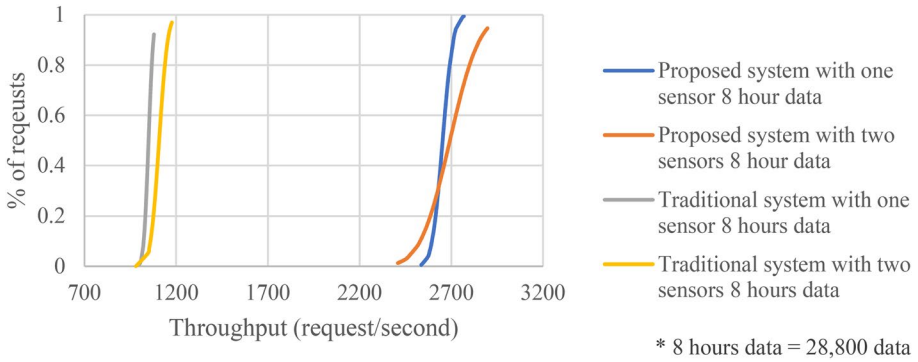


**Fig. 7** Box plot chart of throughput evaluation

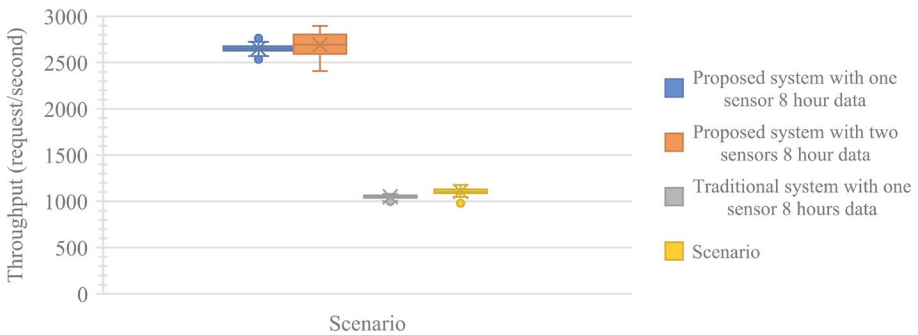


**Fig. 8** Cumulative distribution chart of throughput evaluation

**Table 2** Throughput evaluation scenario

| | Proposed system with one sensor 8 h data | Proposed system with two sensors 8 h data | Traditional system with one sensor 8 h data | Traditional system with two sensors 8 h data |
|---|---|---|---|---|
| Mean | 2652.34 | 2692.01 | 1049.25 | 1106.99 |
| Std dev | 42.43 | 126.81 | 20.78 | 37.03 |
| Min | 2535.88 | 2408.53 | 999.27 | 980.79 |
| Max | 2768.43 | 2896.8 | 1078.77 | 1,176.42 |

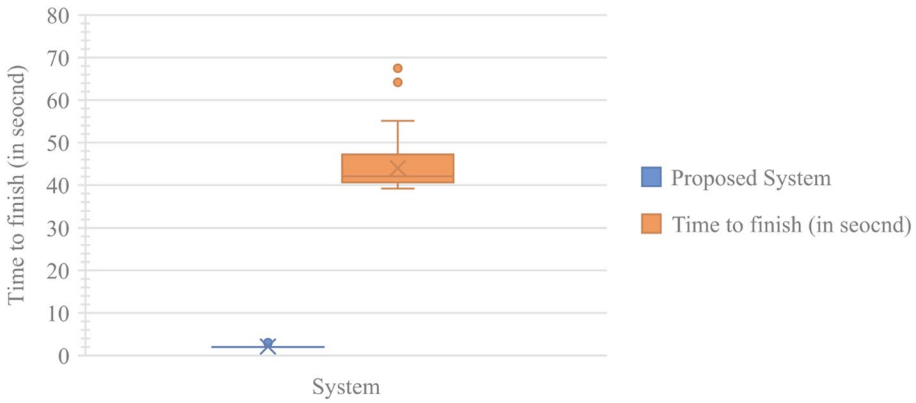


**Fig. 9** Box plot chart for database write evaluation

**Table 3** Database write evaluation summary

| | Proposed system | Traditional system |
|---|---|---|
| Mean | 2.1 | 43.97 |
| Std dev | 0.30 | 5.66 |
| Min | 2 | 39 |
| Max | 3 | 67 |
| Speed | 13,714.29 request/s | 654.99 request/s |

#### *Database write evaluation*

Database performance evaluation is done by running the service that writes data to the database. Figure 9 and Table 3 show the database write performance evaluation result. The lower number of times need to finish the test means the system achieves better database write performance. The proposed system database write evaluation shows the average time to finish the test is 2.1 s; the lowest is 2 s, and the highest is 3 s. On the other hand, the traditional system's average time to finish the test is 43.97 s, the lowest is 39 s, and the highest is 67 s. It means the proposed system is 20.95 times faster in writing data to the database.

The proposed system's write speed increases because InfluxDB has a feature to write data in batches. Writing data in batches helps to minimize network overhead. A single connection can use to write multiple data. Another feature of InfluxDB is it has data compression to speed up writing and reduce network bandwidth.

#### *Database read evaluation*

Another database performance that was evaluated was dataset read performance. Both the proposed system and the traditional system will be evaluated. Figure 10 and Table 4 show the result of the database read evaluation. The lower time to finish means the system achieves better database read performance. The proposed system database read evaluation shows the average time to finish is 21,023 ms, the lowest is 19,929 ms, and the highest is 22,266 ms. On another side, the traditional

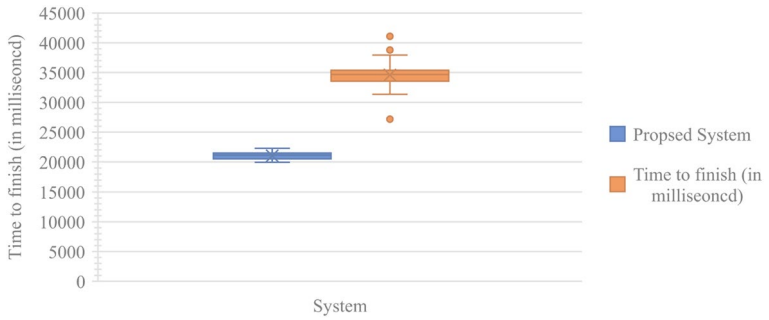


**Fig. 10** Box plot chart of database read evaluation

**Table 4** Database read evaluation summary

| | Proposed system | Traditional system |
|---|---|---|
| Mean | 21,023 | 34,605 |
| Std dev | 618.15 | 2259.62 |
| Min | 19,929 | 27,180 |
| Max | 22,266 | 41,082 |
| Speed | 504.21 request/s | 306.31 request/s |

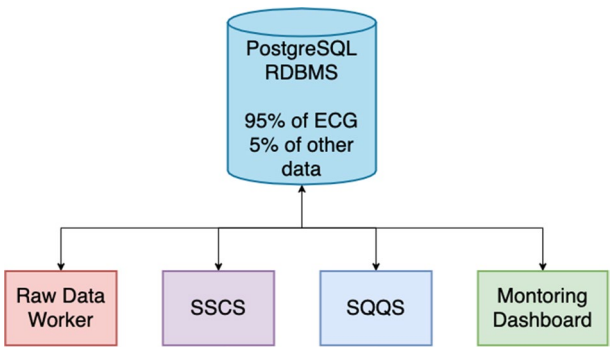


**Fig. 11** The traditional system with a full database

system's average time to finish is 34,605 ms, the lowest is 27,180 ms, and the highest is 41,082 ms. It means the proposed system is 1.64 times faster.

The proposed system database read speed increases because InfluxDB is a columnar database, where data is stored per column basis. Storing data per column helps increase read speed because when data is grouped in columns, it will not retrieve data that is not needed.

#### *System design scalability evaluation*

Another evaluation done on both systems is analyzing the scalability of the system. In both systems, around 28,800 ECG data, assuming patients sleep for 8 h long, were used to generate one sleep quality data. In other words, the ECG table will grow 28,880 times faster than sleep quality data.

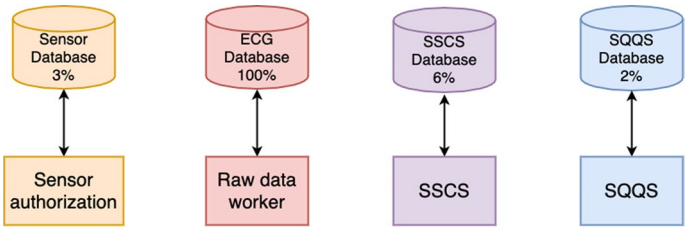


**Fig. 12** The proposed system with a full ECG database

Here is the scenario that was used to analyze the system's scalability. Here we assume to have a PostgreSQL database with 100 GB capacity. Figures 11 and 12 show the condition when the database is full of ECG data on the traditional and proposed systems, respectively. In the traditional system, a database is thoroughly dominated by ECG data, and other data cannot be inserted into the database. In the proposed system, other data can still be inserted into the database. Even though the ECG database is fully occupied, other services have their database.

Adding one more database with the same capacity will give some space for new upcoming data. Figures 13 and 14 illustrate how the database is split into two for the

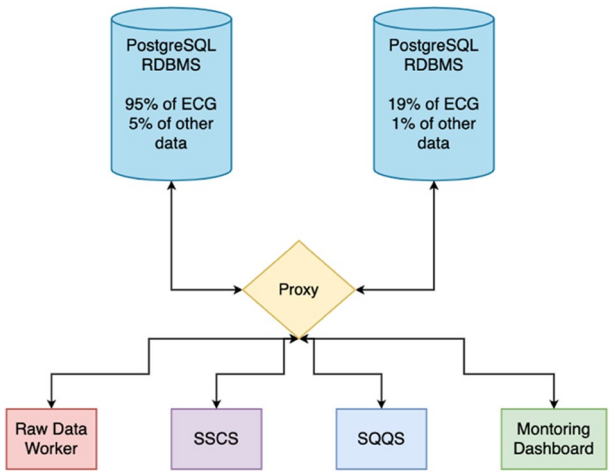


**Fig. 13** The proposed system with two split databases

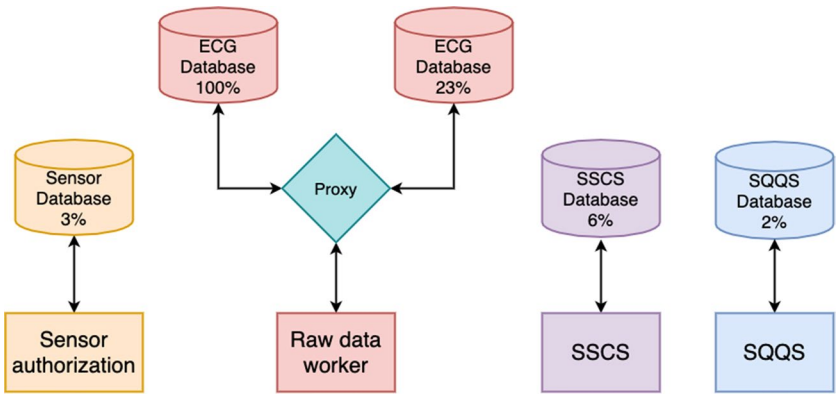


**Fig. 14** Traditional system with two split databases

conventional and proposed systems, respectively. In the traditional system, splitting a single database will force other data that do not need to split become split. In this case, sleep stage and sleep quality data are forced to be split into multiple databases because it coupled with ECG data on the same database. Here we can see that the system scalability is low because there are components that are not needed to scale, forced to scale because of other components. In the proposed system, since it uses multiple databases, we can only split the database that needs to be split. In this case, we can split ECG data into multiple databases without splitting another database. It shows that the proposed system has higher scalability because it allows scaling component that needs to be scaled. Table 5 compares the traditional and proposed system reactions in the full database and splitting database scenario.

#### *System design resilience evaluation*

Another evaluation done of both systems is analyzing the system resilience. A single database can be a single point of failure in the traditional system. A failure that happens in the database will affect multiple services. Raw data workers, SSCS, SCCS, and monitoring dashboards will not be able to process data. This show that the traditional system lack resilience. Figure 15 shows how the traditional system when the database failed.

**Table 5** System scalability summary

| Scenario | Traditional system | Proposed system |
|---|---|---|
| Database full | All services cannot write data to a database | Only raw data workers cannot write data to the database; other services are able |
| Database split | All data is split into multiple databases, even though the data does not need to be split | Only the ECG database is split. Other data is not split |

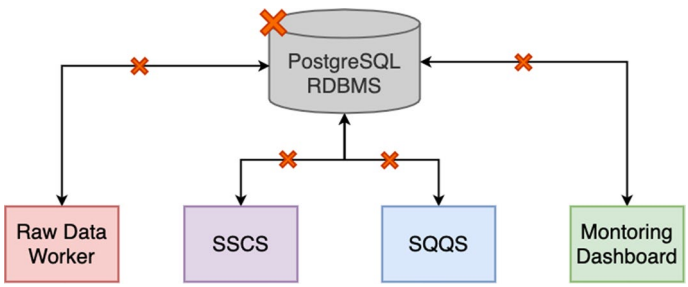


**Fig. 15** Proposed system when database failed

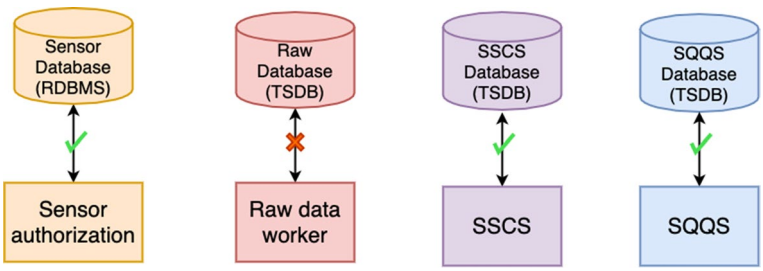


**Fig. 16** Proposed system when database failed

**Table 6** System resilience summary

| Scenario | Traditional system | Proposed system |
|---|---|---|
| Single database failed | All services are not able to access the database | Only a particular service got impacted. Other services are still working fine |

**Table 7** System independence summary

| Scenario | Traditional system | Proposed system |
|---|---|---|
| There are changes in the database | All services must follow the database changes | Only particular services need to follow the changes. Other services do not need to |

Figure 16 shows one database failure and three databases working as expected in the proposed system. If there is a failure on a particular database, other services that are not connected to that database will not be impacted. Therefore, it will decrease the radius of error in the systems. For example, if a failure happens on the raw database, other services like sensor authorization, SSCS, and SQQS are still able to process data. Table 6 shows the summary of comparison between the traditional system and the proposed system with the scenario of a single database failure.

#### *System design independence evaluation*

Another attribute of microservice that is analyzed is system independence. Since the traditional system uses a single database, any changes on the database schema level will force multiple services to follow the changes. At the database level, if there is a plan to change the database type, it will impact all services to follow the changes, even those not required by all services. It shows that the traditional system lacks independence. In the proposed system, since it uses a dedicated database for each service, any changes to a particular database will not affect services that do not interact directly with the database. Table 7 shows the comparison between the traditional system and the proposed system with the scenario that there are changes in the database.

## Conclusion

In this paper, we have presented the improved architecture of the sleep monitoring system. There are three major aspects changed in the proposed method. The first aspect is to propose the MQTT protocol on the sensor gateway. MQTT reuses a single connection to communicate to a client for all data transfers. Therefore, MQTT has a lower overhead for maintaining connections. The MQTT implementation increased the system throughput to 2.43 times faster than the traditional system. The second aspect is the implementation of time series databases. InfluxDB has a write-in batch feature that reduces the number of connections needed to write a bunch of data. This feature increases database write performance by about 20.95 times faster. InfluxDB is a columnar database. A columnar database is preferred for analytic applications because it allows fast retrieval of columns of data. This condition increases database read performance by about 1.64 times faster. The third aspect is the implementation of multiple databases in which one database is only used to store specific data. A database only communicates

with one service. This architecture gives the overall system better scalability, resilience, and independence.

The limitation of the research is that it uses a simulated sensor. The system throughput evaluation results might be slightly different using actual sensors because of network latency and data transmission time. In future research, we might evaluate the results using actual sensors. In future research, we might also evaluate the impact of batch size when storing data in a time series database. It will be beneficial to explore more on time series database write performance.

**Acknowledgements**
Not applicable.

**Author contributions**
ES is the main author of this article. ES worked on the research under the supervision of NS. NS wrote and edited the manuscript. Both authors read and approved the final manuscript.

**Authors' information**
Eko Simanjuntak received his bachelor's degree in computer science from the Del Institute of Technology. Eko Simanjuntak is currently a student in the Master of Computer Science. Bina Nusantara University.
Nico Surantha is a lecturer at the Department of Electrical, Electronics, and Communication, Faculty of Science and Engineering at Tokyo City University. He also teaches at the BINUS Graduate Program-Master of Computer Science, Bina Nusantara (BINUS) University, Indonesia. Nico Surantha received his B. Eng (2007) and M. Eng. (2009) from Institut Teknologi Bandung, Indonesia. He received his Ph.D. degree from Kyushu Institute of Technology, Japan, in 2013.

**Funding**
This work was supported by the Directorate General of Higher Education, Ministry of Education, Culture, Research and Technology, Republic of Indonesia as a part of Penelitian Dasar Unggulan Perguruan Tinggi Research Grant to Binus University entitled "Sistem dan Aplikasi Portabel Pendeteksi Gangguan Irama Jantung Menggunakan Sinyal ECG Berbasis Machine Learning dan Cloud Computing" or "Portable ECG-Based Heart Arrhythmia Prediction System and Application Using Machine Learning and Cloud Computing" with contract number: 163/E4.1/AK.04.PT/2021.

**Availability of data and materials**
Not applicable.

## Declarations

**Competing interests**
The authors declare that they have no competing interests.

Received: 16 February 2022 Accepted: 24 October 2022
Published online: 17 November 2022

**References**

1. Soni VD. An IoT based patient health monitoring system. Int J Integ Educ. 2018;1:43–8.
2. Pereira D, Moreira A, Simões R. Challenges on real-time monitoring of patients through the Internet. Lisbon: Institute of Electrical and Electronics Engineers; 2010.
3. Surantha N, Utomo OK, Isa SM, Lionel EM, Gozali ID. Intelligent sleep monitoring system based on microservices and event-driven architecture. IEEE Access. 2022;10:42069–80.
4. Hellerstein JM, Stonebraker M, Hamilton J. Architecture of a database system. Found Trends in Databases. 2007;1(2):141–259.
5. Yan C, Cheung A, Yang J, Lu S. Understanding database performance inefficiencies in real-world web applications. Proceedings of the 2017 ACM on Conference on Information and Knowledge. 2017; pp. 1299–1308
6. Zhang J, Research on database application performance optimization method. 6th International Conference on Machinery, Materials, Environment, Biotechnology and Computer, 2016.
7. Sekerinski E, Yao S, Fadhel M. A comparison of time series databases for storing water quality data. Mobile Technologies and Applications for the Internet of Things. 2018; pp. 302–313.
8. Bader A, Kopp O, Falkenthal M. Survey and comparison of open source time series databases. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, 2017.
9. Atmoko RA, Riantini R, Hasin MK. IoT real time data acquisition using MQTT protocol. International Conference on Physical Instrumentation and Advanced Materials, 2017.
10. HiveMQ, MQTT & MQTT 5 Essentials, Landshut: HiveMQ, 2020.
11. Pollard B. HTTP/2 in Action, Shelter Island: Manning, 2019.
12. Gourley D, Totty B. HTTP: the definitive guide. Sebastopol: O'Reilly; 2002.
13. Chauhan D, Bansal KL. Using the advantages of NOSQL: a case study on MongoDB. Int J Recent Innov Trends Comput Commun. 2017;5(2):90–3.

14. Khawas C, Shah P. Application of firebase in android app development—a study. Int J Comput Appl. 2018;179(46):49–53.
15. Firebase. Cloud Firestore, Firebase, 29 08 2022. https://firebase.google.com/docs/firestore. Accessed 30 Aug 2022.
16. Vasile M-E, Avolio G, Soloviev I. Evaluating InfluxDB and ClickHouse database technologies for improvements of the ATLAS operational monitoring data archiving. Saas-Fee: IOP Publishing; 2019.
17. Nasar M, Kausar MA. Suitability of Influxdb database for Iot applications. Int J Innov Technol Explor Eng. 2019;8(10):1850–7.
18. Al-Joboury IM, Al-Hemiary EH. performance analysis of internet of things protocols based Fog/Cloud over high traffic. J Fundam Appl Sci. 2018;10(6S):176–81.
19. Sasaki Y, Yokotani T. Performance evaluation of MQTT as a communication protocol for IoT and prototyping. Adv Technol Innov. 2019;4(1):21–9.
20. Musa E, Delač G, Šilić M, Vladimir K. Comparison of relational and time-series databases for real-time massive datasets. MIPRO Computers in Technical Systems. 2019; pp. 1065–1070.
21. Hou L, Zhao S, Xiong X, Zheng K, Chatzimisios P, Hossain MS, Xiang W. Internet of things cloud: architecture and implementation. IEEE Commun Mag. 2016;54(12):32–9.
22. Shashidhar R, Abhilash S, Sahana V, Alok NA, Roopa M. Iot Cloud: in health monitoring system. Int J Sci Technol Res. 2020;9(1):227.
23. Lacalle I, Sarabia-J´acome D, Palau CE, Esteve M. Efficient deployment of predictive analytics inEdge gateways: fall detection scenario. IEEE 5th World Forum on Internet of Things, 2019.
24. Yang Z, Zhou Q, Lei L, Zheng K, Xiang W. An IoT-cloud based wearable ECG monitoring system for smart healthcare. J Med Syst. 2016;40:286.
25. Celesti A, Galletta A, Carnevale L, Fazio M, Ekuakille A, Villari M. An IoT cloud system for traffic monitoring and vehicular accidents prevention based on mobile sensor data processing. IEEE Sensors J. 2017;8(12):4795–802.
26. Silva AF, Ohta RL, Santos MN, Binotto APD. A cloud-based architecture for the internet of things targeting industrial devices remote monitoring and control. Int Fed Autom Control J. 2016;49(30):108–13.
27. Jiang L, Tang Z, Liu Z, Chen W, Kitamura K, Nemoto T. Automatic sleep monitoring system for home healthcare. Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics, 2–7 January 2012.
28. Stec A, Database design in a microservices architecture. 2021. https://www.baeldung.com/cs/microservices-db-design.
29. Singh DA. How much sleep do we really need? 2021. https://www.sleepfoundation.org/how-sleep-works/how-much-sleep-do-we-really-need.


## Publisher's Note