Microservices In IoT-based Remote Patient Monitoring Systems: Redesign of a Monolith

AKMAL ALIKHUJAEV, University of Twente, The Netherlands

Since over half of the world population in remote areas does not have access to medical practitioners, Remote Patient Monitoring (RPM) systems have proved to be a viable alternative. However, because these systems are powered by many IoT devices, they require efficient software architectures. Therefore, an increasing number of applications of microservices architectures in the RPM systems with different models have been proposed. On the other hand, there is still an open question on how to decompose and apply microservice principles in IoT-based systems, while retaining benefits of microservices, like maintainability. Therefore, the goals of this study are to investigate how microservices should be applied in the RPM systems and whether they improve software maintainability. To achieve that we rebuilt the existing SBIoT-MPH system for monitoring hypertension using microservice principles and decomposition patterns. Moreover, to validate the findings we evaluate the end artefact and discuss improved maintainability and the sensor data ingestion throughput. With this study, we aim to provide both researchers and practitioners with a general and comprehensive solution to build microservice-based RPM systems.

Additional Key Words and Phrases: IoT, Microservices, MSA, architecture, design, implementation, Remote Patient Monitoring.

1 INTRODUCTION

According to the ILO[1], about half of the world's population that lives in remote areas, does not have access to healthcare. The reasons for that are many-fold with the major one being the lack of healthcare workers in those areas [16].

Therefore, e-Health passed the innovation stage to become a necessity. That includes Remote Patient Monitoring systems (RPM), which are designed to overcome the challenges of monitoring patients even in the most remote areas, providing equal opportunity to healthcare even for those who cannot commute to medical practitioners [15]. Moreover, RPM systems are frequently developed using Internet-of-Things (IoT) devices, which may contain several sensors on board that can monitor a patient's vital signs, such as blood pressure, heart rate, activity levels and others to help identify and prevent diseases [15]. Furthermore, patients' data is often collected and forwarded to the cloud, where it can be securely stored, queried, and analysed by medical professionals, who are presented with all the data at hand in a software application. The main benefit of storing data locally and then forwarding it to the cloud is that users have complete control over the data that is sampled, increasing the patient's autonomy and confidence [10]. Moreover, IoT systems have the advantage of being (near-) real-time, which can help identify health problems at an early stage, making the treatment more effective [15]. Lastly, early monitoring can help reveal potentially overlooked conditions and give a more holistic view of the patient's health [15].

However, current IoT-based systems, especially in the e-Health domain, generate large volumes of data and require relatively complex processing patterns. As mentioned in [10], the current IoT systems have become increasingly complex and hard to maintain. Furthermore, they require improved scalability and high availability in the context of mission-critical applications like healthcare. Therefore, both software industry practitioners and researchers have proposed the use of microservices-based architectures to overcome those challenges [10].

Consequently, based on the previous research done in this field, there is a need to evaluate the applicability of MSA-based architectures in the context of RPM systems, their decomposition patterns and whether microservices solve the maintainability challenges of the current monolithic IoT systems.

The goal of this study is twofold. Firstly, it involves an investigation into how microservices can be applied to IoT-based RPM systems. Secondly, it involves the redesign of an existing monolithic RPM system for monitoring hypertension proposed in [7] to a microservice-based system, with both implementation and validation of the end artefact.

The study follows the Design Science methodology, which is used to challenge the researcher's creativity and problem-solving as well as the application of IT solutions to a new domain, where a research gap exists. Therefore, with this research, we aim to provide a general solution to both software developers and researchers when applying microservice architecture for RPM systems.

The paper is divided into the following sections. Section 2 presents the research questions. Section 3 explains the methodology used in this research. Section 4 introduces the relevant background concepts that we use in this study. Section 5 presents the design and implementation. Section 6 demonstrates and explains the results of a final comparison. Section 7 presents related work with similar goals that links new findings with past research, and Section 8 provides final remarks and future directions.

2 RESEARCH QUESTIONS

Since the current study involves both a theoretical exploration of the microservice architectures in IoT-based systems as well as their practical application by rearchitecting the system proposed in [7], the following research questions have been defined:

TScIT 39, July 7, 2023, Enschede, The Netherlands

[1] International Labour Organisation is an UN's agency specialized in social injustices.

RQ1: How can microservice architecture be applied in an IoT-based RPM system?
RQ2: Does the microservice architecture improve maintainability of the newly implemented system when compared with the monolithic one?

The **RQ1** is an exploratory question that can be answered by assessing the relevant literature, whereas **RQ2** is a predictive question and it can be answered only upon completion of architectural design and implementation of the system in [7].

3 METHODOLOGY

To answer **RQ1** we found and studied the relevant literature on microservices, IoT systems and how they can be combined in an architecture. To accomplish that we have used the Google Scholar platform to search for articles based on keywords like "IoT", "architecture", "microservices" and "remote patient monitoring". The major drawback of this approach is that there is a lot of literature available and because of the limited time, only a relevant subset of articles to the problem domain was considered.

On the other hand, **RQ2** involves a practical exploration and for that purpose, we applied the Design Science methodology in our research.

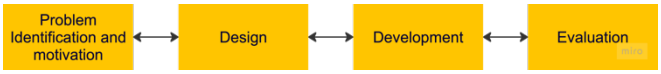

Fig. 1. Design Science Methodology process

Design science is a paradigm that challenges researchers to apply problem-solving and aims at devising creative solutions to a well-defined problem by practically implementing the artefact [5]. The whole process is shown in Figure 1 and consists of problem identification and problem motivation, following a design and development phase, and concluding with an evaluation. Evaluation in Design Science can be either through mathematical or empirical methods [5]. Moreover, each stage of the Design Science methodology (see Figure 1) is part of a feedback loop, which makes the whole process iterative. The goal of the methodology is to provide a practical contribution to the research space by providing a creative solution to a problem. Therefore, in the context of this paper, we explore the problems of the monolithic architecture proposed in [7], offer a design following MSA principles, implement a prototype and validate it. Moreover, we follow the Design and Engineering Cycle method [14] that suggests design validation techniques. We conduct the validation of the design in a laboratory setting by creating a simulated environment to demonstrate how the implemented artefact solves a particular use case, which in our case is an improvement of maintainability. To accomplish this, we propose a new functionality and evaluate the number of changes that are required in the monolithic and MSA-based systems.

4 BACKGROUND

In this section, we introduce relevant concepts presented in both academic and grey literature that we used during the design and implementation.

4.1 IoT Systems

IoT systems are often complex, and it is common to reason about them as a 3-tiered architecture that includes the Edge, Fog and Cloud computing layers. Firstly, Edge computing is the technique of processing the data on the physical devices deployed next to the source, enabling real-time data processing and fast feedback. Secondly, Fog computing acts as a layer in-between edge and cloud, relieving the cloud of the load, and supporting more complex processing than the edge computing layer. Lastly, with the emergence of cloud service providers like AWS, Azure and GCP, the IoT architectures began to leverage them and utilise the cloud for aggregating, storing and analysing massive amounts of data [3]. These three layers combined are key elements to develop scalable, real-time, and resilient IoT architectures.

4.2 Microservice Architecture

In [8], Sam Newman introduces the concept of microservices. In a nutshell, microservices are independently deployable software artefacts that are loosely coupled and are modelled around a particular aspect of the business domain. Consequently, to ensure a degree of isolation, microservices are deployed on different hosts and more frequently leverage virtualisation tools like, e.g., Docker. Microservice architecture has emerged as a specialisation of Service Oriented Architecture (SOA) to address its challenges and introduce novel solutions. While both revolve around the concept of loosely coupled services that encapsulate business functions, they largely diverge in governance patterns. For example, in traditional SOA, services are more commonly orchestrated by an Enterprise Service Bus (ESB) that centralises governance functions like routing, authentication and integration of services. Whereas, MSA promotes the decentralisation of the architecture, which allows the services to evolve in isolation [11]. Furthermore, traditional SOA favours the orchestration of business transactions, whereas MSA encourages choreography [11]. However, that does not mean that traditional SOA is inferior to MSA, since those are the two approaches that have valid use cases.

4.3 Microservices and IoT

The motivation behind the industry migration to MSA is outlined in [1], which claims that microservice architecture was created to address problems of maintainability and scalability in monolithic architectures. The same article also mentions the similarities between the requirements for the IoT and microservices-based systems. Furthermore, as IoT systems are composed of a large number of devices that generate large volumes of data, a simple monolithic system simply cannot handle the required load [1].

Moreover, the significance of microservice architectures in IoT-based RPM systems has been researched previously. The authors in [10] claim that efficient architectures that employ IoT and microservices are invaluable components of the modern e-Health domain. They also propose a reference architecture built around MSA principles for remotely monitoring patients, which provides real-time analysis on the patient's smartphone and then forwards that data to the cloud. In [4], a similar architecture is proposed for frailty status assessment. Both [4,10] pre-process the data at the fog layer (a smartphone) and then forward the summarised data to the microservice-based system (cloud), and justify that choice by claiming that it enables real-time processing and fast feedback.

4.4 Domain Driven Design

However, one of the mentioned research challenges in [10] is that the proposed MSA-based IoT architectures are lacking standardisation, are hardly replicable and vendor-specific. Moreover, most of the papers present the final reference architectures without presenting decomposition patterns. Therefore, more generally as stated in [1] a common approach to microservice decomposition is to apply Domain Driven Design (DDD). DDD was first introduced by Eric Evans in [2] as a pattern to manage software complexity. It revolves around modelling the software in line with the business domain. Furthermore, DDD makes a distinction between two components: the problem space and the solution space. The problem space describes the business capabilities, words, requirements and use cases. Furthermore, the problem space entails the domain, which is a set of patterns aimed at solving business needs, and the sub-domains which encapsulate business functions into manageable pieces. Consequently, when translating this into a software architecture, DDD requires engineers and domain experts to create domain models and separate them into bounded contexts. In some sense, bounded contexts act as a boundary within which the domain model has a meaning and decreases the software complexity by separating the concerns. Evans suggests one way of identifying the bounded contexts by looking at where the same domain model has a different meaning [2]. This modular thinking of the system is what makes DDD extremely useful and popular for microservice identification.

4.5 Containers

Isolation is one of the core principles of microservices, presented in [8]. This means that there is a need to ensure the microservice application running on a physical host (server) does not interfere with other processes and vice versa. Traditionally, that has been achieved by using virtual machines (VM), which provide a completely isolated operating system, virtualised hardware, networking, kernel, and hypervisor. However, VMs underutilise the hardware and are rather heavy [8]. Container orchestration software, like Docker, on the other hand, is a technology designed specifically for cloud-native applications that provides complete isolation of software components and does not require to reserve hardware resources, making containers elastic. It leverages the Linux Containers feature, which renders hypervisor software obsolete. Moreover, containers are advantageous because their provisioning and start-up time is much faster compared to traditional VMs [17].

4.6 Message Broker

In our work, we decided to use a message broker for inter-service communication as opposed to the synchronous HTTP request-response pattern. In particular, we selected the popular, open-source Apache Kafka [2] distributed event-streaming software developed at LinkedIn. The reason for its widespread use is that Kafka provides a strong consistency model for message delivery and guarantees message ordering within a partition. Furthermore, it provides replication, which increases fault tolerance and reduces the chance of data being lost. Apache Kafka provides built-in partitioning of data that distributes it around the cluster and allows parallel processing by multiple partition leaders. Consequently, just 3 Kafka brokers deployed on 3 servers using commodity hardware provide a throughput of 2 million messages per second [18], which makes it an excellent tool for handling the data traffic normally present in an IoT-based system.

5 DESIGN AND IMPLEMENTATION

In this section, we first introduce the monolithic SBIoT-MPH RPM system for monitoring hypertension implemented in [7] and address the challenges that one of the developers of this system outlined in a collaborative session. Thereafter, we decompose the system by applying Domain Driven Design. Following the decomposition, we introduce the technologies used in the project and briefly explain the developed microservices.

5.1 SBIoT-MPH: System, Domain and Challenges

In [7], the authors report on the implementation of the IoT-based RPM system for monitoring hypertension called SBIoT-MPH. Hypertension is a cardiovascular disease with a yearly mortality rate of 9.8 million people [7]. It requires continuous monitoring of patients' vitals by health professionals, which justifies the use case for an IoT-based RPM system.

SBIoT-MPH has a modular architecture and is split into 3 layers, namely, the Sensor, Fog and Cloud layers. The Sensor Layer consists of sensors embedded into a wristwatch-like device and is worn by a patient. The first sensor is the MKB0805 module for blood pressure and pulse measurements. The second sensor is the MPU6050 module for measuring acceleration on 3 axes (X, Y, Z), which is required for the MKB0805 readings. Lastly, the third sensor is the DS18B20 module used to capture the patient's body temperature. The three sensors constitute a Wireless Body Sensor Network (WBSN), whose data is captured and processed by the TTGO T7 V1.3 MINI 32 module, which is a common hardware component in the current IoT system. Consequently, collected and processed data is forwarded to the patient's mobile device via Bluetooth Low Energy (BLE) protocol employing the ESP32 microcontroller [7].

The Fog layer consists of an Android application developed in Java and runs on the patient's mobile device. The application provides authentication, sensor management, data visualisation, collection and analysis. Moreover, the collected sensor data is stored both locally in an encrypted database and forwarded to the cloud through a background process. The application provides an easy-to-use interface that displays alerts, and messages from health professionals and provides an overview of the collected sensor data.

The Cloud Layer consists of an API server and a client-side Web application. The API server acts as a central governing authority that collects the data, authenticates administrators, patients and medical practitioners. Furthermore, it provides patient and health professional management to the administrator to invite new users to the system. Moreover, it exposes an interface for patient-to-health-professional assignment, sensor data analysis, message exchange and alert monitoring. The API server is developed in JavaScript and Node.js, which exposes a GraphQL communication endpoint.

[2] Apache Kafka https://kafka.apache.org

Although SBIoT-MPH works well, its developers have faced several challenges after deploying the software artefact. During a collaborative session, we identified two main problems: (1) as the number of requirements grows, so does the complexity of the system. It was mentioned that the architecture requires a new approach to manage complexity; (2) we identified that the sensor data read/write speeds (throughput) were quite low and did not deliver a comfortable user experience (UX). Therefore, we decided to redesign the system using a microservice architecture to increase maintainability and improve sensor data throughput.

5.2 Microservice Decomposition

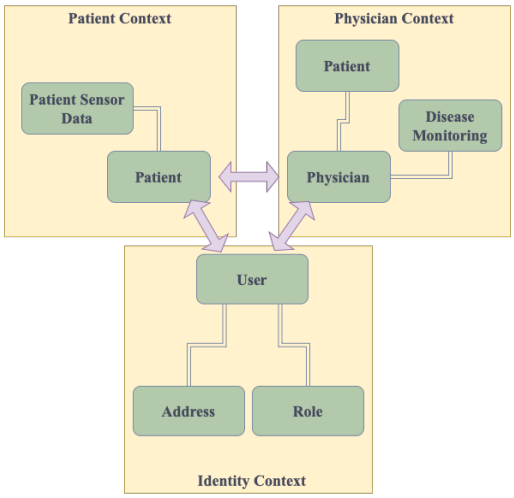

Fig. 2. Business sub-domains and contexts

We applied the Domain Driven Design decomposition methodology proposed by Eric Evans in [2], to create domain models and identify the bounded contexts.

The domain models have been identified in a two-step process. Firstly, the domain was introduced by the expert in a collaborative session. Secondly, we analysed the source code of the existing application, drew diagrams and dependency graphs. The latter helped us identify constraints and define an understanding of different use cases.

Figure 2 shows the domain diagram, which contains the domain concepts (green boxes) and bounded contexts (yellow surrounding boxes). The core domain concepts that have been identified are patient, physician, user and patient's sensor data, whereas the core bounded contexts are patient, physician and identity.

The patient context encapsulates the patient's information, medical records and sensor data. The physician context abstracts physician information, disease monitoring and patient-to-health-professional assignment. The patient concept appears again in this context, however, the understanding of the patient model in the physician context differs from the one in the patient context. For instance, when discussing the patient in the physician context we often do not require any contact or private information that should be visible to the patient only. Therefore, the patient model has only a subset of information that is relevant to the physician. Lastly, the identity context is where general user management takes place. Consequently, it is not a core sub-domain, but rather a supporting one because without it there is no way to manage user authentication and roles.

In MSA, each bounded context is a good candidate for a microservice because it is an isolated set of capabilities that interacts with other contexts through a well-defined set of constraints and models [8]. We have identified the bounded contexts by looking at where the meaning of a user changes throughout the system. In the original implementation of SBIoT-MPH, a user can be either an administrator, patient or physician. This distinction makes a clear separation of the context and the meaning of the user throughout the system.

Another advantage of splitting the domain into bounded contexts during development is cohesion. As a general rule in microservices, code that changes together must stay together [8]. For instance, when working with a patient's sensor data, it is often required to correlate that information and perhaps perform some modifications in the context of a patient because it is the only place where we have a complete view of the patient's data. Therefore, if new requirements surface, the changes to the sensor data processing will not span multiple bounded contexts, thereby reducing the chance of violating the independent deployability principle [8].

5.3 System Implementation

The system was developed using Java version 17 along with Spring Boot 3 and Spring Cloud frameworks. Spring Framework is a mature library for building server-side applications, with extensive tooling and an active developer community. Furthermore, Spring Cloud is a sub-project of the Spring ecosystem that provides the tools to build cloud-native applications. Spring Cloud provides out-of-box implementations for security, API gateway, service discovery, caching, message broker integration and others, which makes it a great tool for building microservice-based applications.

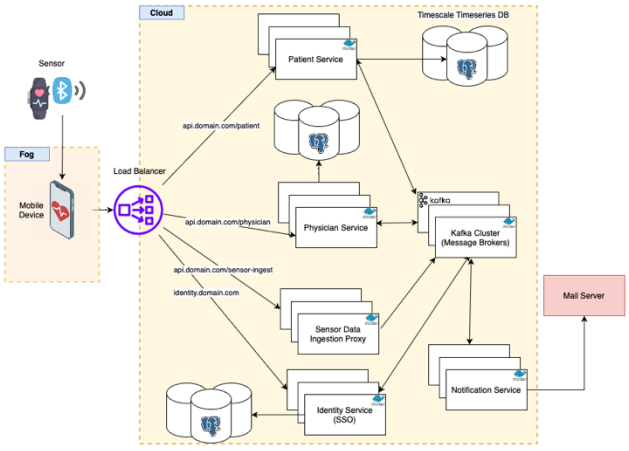

Fig. 3. SBIoT-MPH Microservices Architecture

The proposed architecture (see Figure 3) features 3 layers, namely, the Sensor, Fog and Cloud layers. The sensor and the fog layers remain the same as in [7], where the blood pressure, pulse and body temperature are sampled and forwarded via BLE protocol to the fog layer for pre-processing and real-time analysis. The main difference begins in-between the fog and cloud layers, where previously the mobile device forwarded the data to the API server using GraphQL protocol, in the new architecture, however, the

device uses Representation State Transfer (REST) API and transfers the data via HTTP in JSON format to the Sensor Data Ingestion (SDI) Proxy. All calls to the services are routed by the load balancer, which can select the least congested service instance through service discovery. Another advantage of having a load balancer is that internal topology can change, with new instances being added or removed, making them available to the clients instantly without waiting for the DNS changes to propagate [8].

Moreover, the architecture follows the Database-Per-Service pattern introduced in [8], which means that every service has its set of database instances that no other service can access. This solution mitigates potential consistency issues and increases the degree of information hiding, forcing other services to use well-defined API contracts.

As stated in Section 4.6, the system utilises asynchronous communication patterns between microservices through a message broker, making the services loosely coupled and more resilient to cascading failures in case of an unhealthy microservice instance [13]. Moreover, the developed microservices are unaware of each other's existence and only publish the events that interested parties (other services) can consume and react to, thereby, making this an Event-Driven-Architecture (EDA). Events in this case are statements about an action that has happened. This kind of collaboration pattern models the real-world more closely since everything that happens around us is a series of events that we react to. To enable this communication pattern, we deployed a Kafka cluster consisting of 3 brokers in the KRAFT mode, which utilises the RAFT consensus algorithm that performs leader elections in the cluster without centralised cluster management software.

As shown in Figure 3, each microservice and infrastructure-related software runs in a Docker container to support the requirement of isolation outlined in [1,8]. Another advantage of using containerisation for microservices is the replicable environment since containers abstract the underlying operating system (OS). That ensures that if an image is built for a desired computer architecture, it can be deployed to any cloud instance without fear that the application does not work.

The overall system consists of patient, physician, identity and notification services as well as an SDI proxy, where each one of them has its database, maintains a connection to the central Kafka cluster and runs in an isolated Docker container.

5.4 Identity Service

Authentication and authorisation are crosscutting concerns that must be tackled with care not only to achieve compliance but also to ensure data confidentiality, integrity and availability (CIA). Authentication is rather a trivial task when implemented using 3rd party tools in a monolithic application since all the logic and interactions reside in one application. However, in MSA if each microservice authenticates its users, this would lead not only to code duplication but also to poor UX because the user will have to present the credentials to each service [8]. Therefore, both [8,9] stress the importance of the Single-Sign-On (SSO) in a microservices architecture. SSO features a seamless authentication experience where the user presents the credentials only once and can interact with a variety of services afterwards. The core component of SSO is an identity provider, which can be either managed like Okta, Auth0 or AWS Cognito [8], or can be self-hosted.

Since our MSA-based SBIoT-MPH system is event-driven, we decided to build our own identity service using Spring Boot, Spring Security and Spring Authorization Server frameworks. In particular, Spring Authorization Server provides a robust implementation of OAuth 2.0 and OpenID Connect (OIDC) 1.0 standards, which are commonly used to achieve SSO. OIDC 1.0 protocol is used for authentication, whereas OAuth 2.0 is used for authorisation.

Furthermore, to address the challenge of propagating the user's authentication state, we have selected JSON Web Tokens (JWT), which are issued by the service upon successful authentication. Issued JWTs are cryptographically signed using RSA private key and SHA512 hashing algorithm. JWTs were selected because they provide a lightweight approach to transmitting an authentication state that makes it unnecessary to call the identity service to validate whether the user is authenticated and/or authorised [8].

Moreover, the identity service was extended to support signing key rotation and utilises PKCS12 key store type to store private RSA 2048-bit signing keys. In contrast, public keys are collected into a JSON Web Key Set (JWKS) and are exposed via RESTful API so that other microservices can load the keys at start-up and validate each incoming request, without the need to call the identity service.

The identity service also provides user management capabilities, such as patient and physician registration, and as shown in Figure 3, it stores the user data in a PostgreSQL database. Each update to user information generates events that are forwarded by the Kafka message broker so that other services that duplicate user information can react to those events and update the local user view.

Lastly, physician and patient registration can be achieved by emitting the respective events from the physician and patient microservices, to which the identity service reacts by creating the user record, generating the activation token and sending the User Created event, which is consumed by the notification service to generate a welcome email.

5.5 Patient Service

The patient context (see Figure 2) is represented by a microservice that encapsulates patient-related functions. This microservice was developed using Spring Boot and has its own time-series Timescale database instance to store both patient information and health sensor readings. Time-series database is a preferred data store for the IoT sensor data because it provides high write throughput and low read latencies [9,12,13]. We have selected the Timescale database because it extends an ordinary PostgreSQL instance with time-series capabilities, where the data is partitioned based on its timestamp.

Furthermore, it eliminates the need to learn an additional query language since the data can be queried using ordinary SQL syntax. Timescale database also allows for both Online Transaction Processing (OLTP) and Online Analytics Processing (OLAP) workloads, which eliminates the need to have two separate databases for patient data and sensor readings.

Moreover, the service exposes a RESTful API for administrators to create, update and delete patients, leading to the emission of respective domain events. For example, when the patient is created,

the service emits Patient Created event, to which the identity service reacts and provisions the user account. On the other hand, when the deletion of a patient is requested, all the services remove the associated patient data and invalidate the sessions. Lastly, it provides the interface for patients and medical professionals to query the sensor readings for a given time range, sensor type and patient.

5.6 Sensor Data Ingestion (SDI) Proxy

Another microservice we built following the findings in [9,13] is called Sensor Data Ingestion (SDI) Proxy. It has two primary roles: (1) analogous to Technology Integration Adapter in [9] the service abstracts the modalities of IoT sensors and provides an interface for sensor integration. However, in the context of the SBIoT-MPH system, the only integration that was implemented is a RESTful API that accepts sensor readings from a mobile device, but it has limitless capabilities to be extended and make it work with various IoT sensors and communication protocols without changing other system components [9]; (2) the service acts as Inbound Pipeline in [9] and as a Sensor Gateway in [13], which works as a buffer between the fog layer and the patient data service. Its advantages of it are twofold: (1) as shown in [13], it increases the throughput and reduces the write latency; (2) it allows the development of loosely coupled event-based systems [9].

The SDI Proxy works as follows, when the fog layer pre-processes the sensor data, it sends it to the SDI Proxy service. Thereafter, SDI Proxy performs an integrity check of the data and publishes it as an event to the Kafka cluster. The sensor readings are then consumed by the patient service and are inserted into a time-series database. That means that in case of high load spikes the number of instances of SDI Proxy can be increased, whereas the patient service can remain unchanged and process the data at its own pace without affecting the load and write times.

In contrast with other services, this microservice was built around Reactive Programming Paradigm, which favours non-blocking IO operations. Traditionally, web servers were implemented using a thread-per-request pattern and that approach worked well if the operations did not require frequent IO interactions. However, in the case of the SDI proxy, all interactions except for data integrity checks are IO related, such as serialising/de-serialising HTTP requests and pushing the data to the Kafka broker. Consequently, that means that most of the write requests will have to be blocked waiting for the operating system to complete the data transfer, which precludes those threads from servicing the requests of other users. Therefore, Reactive Paradigm is a suitable implementation for the SDI proxy, and it is expected to increase the overall throughput under high load. Moreover, the web capabilities are implemented using Spring Webflux, which bases itself on Project Reactor that is built according to the Reactive Streams specification for non-blocking operations.

5.7 Physician Service

Physician management was one of the original functionalities of the SBIoT system. Therefore, we implemented the physician service that resides in the physician domain context (see Figure 2). This microservice is built as the other core services using Spring Boot and has its own PostgreSQL instance (see Figure 3). It provides a RESTful API for administrators to create, update and delete physicians, and analogous to patient service, it emits events that are consumed by the identity service for user account provisioning. Furthermore, it exposes the endpoints to query all physicians in the system as well as physicians by patient and all patients for a given physician. Additionally, the server exposes an API to assign a medical professional to a patient, which creates a many-to-many connection. Lastly, as in the original SBIoT system, the service provides functionality to send a message from a physician to an assigned patient. The physician can also select the message priority and upon validation of patient assignment, the message event is emitted to the notification service, which can be then queried by the patient.

5.8 Notification Service

The notification service falls under the supporting domain and as shown in Figure 3 is connected to the external mail server outside the private network through SMTP protocol. Similarly, to other services, it is built using the Spring Boot framework and utilises the PostgreSQL database to store user emails and physician messages. This microservice consumes the notification events from Kafka brokers such as User Activation Required and Message events, to which it reacts and sends the notifications. Furthermore, it follows the de-normalised data approach and stores partial records of the users with their respective emails. Lastly, the microservice provides a RESTful API to query all messages by physicians and patients.

6 RESULTS

In the last stage of the Engineering Cycle methodology, it is expected to evaluate the end software artefact. For that purpose, we have deployed both the monolith and the microservices-based SBIoT-MPH system on a physical machine with a 10-core CPU and 32GB of RAM. Along with that, we provisioned another 10-core CPU machine with 16GB of RAM for the k6[3] load testing tool that helped us simulate concurrent users and generate large volumes of synthetic sensor data. Furthermore, in the microservices-based implementation, each service was deployed as a single instance to provide a better comparison with the existing system.

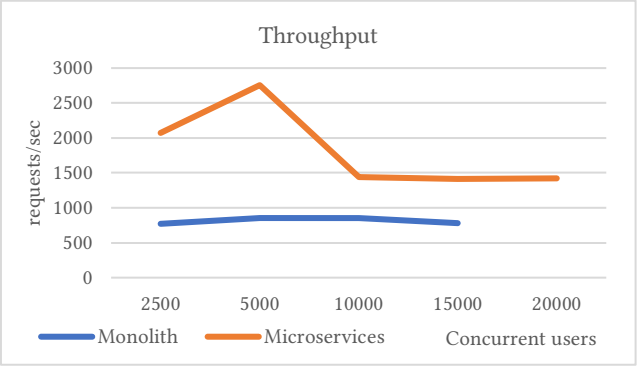

Fig. 4. Throughput comparison graph

Figure 4 shows the throughput graph with the number of concurrent users on the x-axis and the number of requests per

[3] k6 load testing tool https://k6.io

second on the y-axis. The data was sampled by running both microservices and monolithic systems in isolation with their own infrastructure to ensure that other processes do not cause the congestion of computing resources. The throughput data suggests that the microservices-based implementation has a 28% increase in sensor data ingestion throughput via the SDI Proxy service. These results can possibly be attributed to the introduction of the SDI proxy service that publishes ingested sensor data to the Kafka cluster, thereby, buffering the data. These findings closely align with the results obtained with a system that has a similar event-based architecture in [13].

Moreover, Figure 4 shows that the microservices-based implementation, in particular SDI proxy, can withstand the load of 20000 concurrent users with an average response time of 4.53s and a throughput of 1420 requests/sec. On the other hand, the monolith's results for the same load parameters were excluded because the error rate was as high as 73% and, therefore, the system was assumed to be unavailable. One of the reasons for the monolithic SBIoT-MPH system failing under the load was the database writes, where many user requests were idle for a long time waiting for the database writes to complete. In contrast, the microservices-based implementation utilised both a message broker to buffer the requests as well as a time-series database to ingest sensor data, which are probably the reason for the results above.

7 RELATED WORK

Architectures for MSA-based IoT RPM systems have already been studied by many researchers.

In [12], the authors propose a general architecture for a multi-purpose MSA-based RPM IoT system. The system includes 5 microservices that leverage a variety of communication protocols like HTTP, MQTT and Web sockets. It has a general service built using the Django framework that handles user authentication and authorisation, enables chat capabilities with medical practitioners and provides patient data over HTTP protocol, whereas other services include MQTT broker, patient data, reporting and time-series data ingestion services. Moreover, the authors show the importance of using a time-series database for IoT sensor data because it provides greater scalability and data partitioning.

In contrast, [9] proposes a layered architecture that enables the development of a near-real-time, multi-tenant RPM system for monitoring chronic metabolic disorders. One of the main advantages of the system is that it uses an abstraction layer called Technology Integration Adapters (TIA), which allow the integration of heterogeneous IoT devices without changes that span the whole system. The developed artefact leverages a microservice architecture and is designed to be deployed in the cloud. Furthermore, to address scalability and maintainability, the authors created an Inbound Pipeline microservice that receives the IoT sensor data and publishes it to a message broker. The advantages of this solution are twofold. Firstly, it creates a buffer space and lets other microservices process the data at their own pace, thereby avoiding load spikes. Secondly, as stated in [9], it makes services loosely coupled, which improves maintainability and opens the possibility to add new services without changing other microservices, therefore, satisfying the set of requirements for MSA-based systems introduced in [1] and [8]. Other microservices presented include a user service that manages authentication, a patient service that stores patient's data and a patient monitoring service, which enables medical professionals to access and monitor the subject. Moreover, as in [12], the authors address the importance of the use of time-series data stores for sensor data to reduce latency and increase throughput [9]. However, this solution differs from our system largely because we further decomposed the system components and made the data ingestion layer follow the Single Responsibility pattern, omitting the introduction of business logic at the ingestion layer.

Similarly, [6] proposes domain-specific Internet of Health Things (IoHT) architecture for non-invasive remote monitoring of the elderly called RO-SmartAgeing. The system is cloud native and utilises a microservices architecture to achieve a high degree of customisation, flexibility, scalability and extensibility [6]. The authors employ a Raspberry PI 4 micro-computer to simulate IoHT devices generating pulse, body temperature, blood oxygen level, and blood pressure data as well as mimicking environmental, motion and wearable sensors. Analogous to [10] and [4], the data is summarised and pre-processed at the fog layer, enabling real-time decision-making and data buffering [6]. Furthermore, the system is composed of different microservices, such as advanced analytics, behavioural monitoring, triggers and alerts, and patient data services. However, it is unfortunate that the work in [6] does not provide any results and implementation. Nevertheless, it provides a sound theoretical framework and argumentation.

In contrast, [13] reports on one of the most detailed, well-architected and modern MSA-based IoT RPM systems for sleep monitoring. The authors introduce in detail the concept of event-driven architectures (EDA) and show how loosely coupled an IoT system can be. All the participating microservices are unaware of the existence of other services and only consume/publish events. Events in this case are statements about the world depicting an action or notifying about data changes. Similarly to [9], a sensor gateway (analogous to Inbound Pipeline) is introduced that buffers the sensor data and publishes it to the message broker in the form of events. Thereafter, the authors implement the Data Persistence service that consumes the events and stores the data in a time-series database to achieve high throughput and low read latencies. Moreover, the same sensor events are captured by the sleep monitoring and sleep classification services that rate the quality of sleep based on the ECG readings. To validate the architecture, the authors simulated large streams of data at the fog layer to the cloud. The study also features empirical analysis of metrics comparing monolithic and MSA-based systems, which includes response time, throughput, and RAM usage. The microservices-based system excelled in all 3 benchmarks, showing that the throughput was increased by 92%, response time decreased by 75% and lower RAM usage that was adjusted per physical node.

8 FINAL REMARKS

This study explores the ways microservices can be applied in IoT-based RPM systems and whether it improves long-term maintainability to battle software complexity. We introduce relevant technologies that enable microservice communication such as Kafka as well as Docker to ensure the microservices run in isolation. Furthermore, we introduce and explain the concept of Domain Driven Design and its benefits when identifying microservice boundaries.

Moreover, to evaluate our findings and contribute to the research landscape we conclude the study with a comprehensive system design and implementation. Following that we introduce the findings about the improved throughput of the event-based microservice architecture by 28% and compare it with an existing implementation. Going further, given the final reference architecture and the relevant literature studied, it is now possible to answer RQ1 and RQ2.

RQ1 addressed the ways of how microservices can be applied in the IoT-based RPM systems, which was demonstrated in studies [1,4,6,9,12,13] and the concepts learned were practically applied in the implementation of the SBIoT-MPH system in [7] using a microservices-based approach.

To conclude whether the microservices-based approach improves maintainability, we proposed a change in both systems and evaluated the changes required and the instances that must be redeployed. The proposed new requirement is the transfer of the alert processing business logic from the fog layer to the cloud, following the same thresholds for readings as in [7]. In the monolithic system, we must update the database schema to store the alerts and introduce the code to process the sensor readings and classify them. The changes span all 3 layers of the application, namely, the API, service and persistence layers. That means we must update the existing database schema and redeploy the whole application. However, in the microservice-based implementation alerts do not fall under any of the existing business contexts and can be implemented as a separate service. Furthermore, the sensor readings with all the required patient details are already available in the Kafka topics. Therefore, after the development of a new service, it is a matter of adding one more consumer to the Kafka cluster that can start to classify the sensor readings immediately. No other component has to be redeployed or changed and the new functionality can be released independently, thereby, satisfying the requirements in [8]. That demonstrates that event-based microservice systems, through decoupling consumers and producers, can indeed improve maintainability and reduce complexity. These findings are closely aligned with the results introduced in [9,13].

Due to the limited timeframe of the research project, certain aspects were not explored. Therefore, we encourage researchers to look for possible extensions of the proposed microservice-based architecture. One possible extension could be the addition of IoT sensors to extend patient monitoring capabilities. Consequently, another direction that could be pursued is the complete deployment and evaluation of the system on the public cloud provider like AWS, GCP or Azure and possibly improvements to make the architecture completely cloud native. Furthermore, we advise researchers to try different message broker implementations and protocols to identify the best event streaming software platform for the IoT-based RPM systems.

9 REFERENCES

[1] Björn Butzin, Frank Golatowski, and Dirk Timmermann. 2016. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1–6. DOI:https://doi.org/10.1109/ETFA.2016.7733707

[2] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[3] Farshad Firouzi, Bahar Farahani, and Alexander Marinšek. 2022. The convergence and interplay of edge, fog, and cloud in the AI-driven Internet of Things (IoT). *Inf. Syst.* 107, (July 2022), 101840. DOI:https://doi.org/10.1016/j.is.2021.101840

[4] Francisco M. Garcia-Moreno, Maria Bermudez-Edo, José Luis Garrido, Estefanía Rodríguez-García, José Manuel Pérez-Mármol, and María José Rodríguez-Fórtiz. 2020. A Microservices e-Health System for Ecological Frailty Assessment Using Wearables. *Sensors* 20, 12 (January 2020), 3427. DOI:https://doi.org/10.3390/s20123427

[5] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. Design Science in Information Systems Research. *MIS Q.* 28, 1 (2004), 75–105. DOI:https://doi.org/10.2307/25148625

[6] Marilena Ianculescu, Adriana Alexandru, Gabriel Neagu, and Florin Pop. 2019. Microservice-Based Approach to Enforce an IoHT Oriented Architecture. In *2019 E-Health and Bioengineering Conference (EHB)*, 1–4. DOI:https://doi.org/10.1109/EHB47216.2019.8970059

[7] Pedro Lopes de Souza, Wanderley Lopes de Souza, Luis Ferreira Pires, João Luiz Rebelo Moreira, Ronitti Juner da Silva Rodrigue, and Ricardo Rodrigues Ciferri. 2023. Ontology-Driven IoT System for Monitoring Hypertension. *ICEIS* 169, 2 (2023).

[8] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.

[9] Edoardo Patti, Maria Donatelli, Enrico Macii, and Andrea Acquaviva. 2018. IoT Software Infrastructure for Remote Monitoring of Patients with Chronic Metabolic Disorders. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, 311–317. DOI:https://doi.org/10.1109/FiCloud.2018.00052

[10] Euripides G. M. Petrakis, Stelios Sotiriadis, Theodoros Soultanopoulos, Pelagia Tsiachri Renta, Rajkumar Buyya, and Nik Bessis. 2018. Internet of Things as a Service (iTaaS): Challenges and solutions for management of sensor data on the cloud and the fog. *Internet Things* 3–4, (October 2018), 156–174. DOI:https://doi.org/10.1016/j.iot.2018.09.009

[11] Dharmendra Shadija, Mo Rezai, and Richard Hill. 2017. Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, 1–6. DOI:https://doi.org/10.23919/IConAC.2017.8082018

[12] B. A. Sujatha Kumari, K. S. Shreyas, M. S. Skanda, Manoj Kumar, and C. D. Prajwal. 2022. IOT-Based Remote Patient Monitoring System Using Microservices Architecture. In *Soft Computing for Security Applications* (Advances in Intelligent Systems and Computing), Springer, Singapore, 365–381. DOI:https://doi.org/10.1007/978-981-16-5301-8_28

[13] Nico Surantha, Oei K. Utomo, Earlicha M. Lionel, Isabella D. Gozali, and Sani M. Isa. 2022. Intelligent Sleep Monitoring System Based on Microservices and Event-Driven Architecture. *IEEE Access* 10, (2022), 42069–42080. DOI:https://doi.org/10.1109/ACCESS.2022.3167637

[14] Roel J. Wieringa. 2014. The Design Cycle. In *Design Science Methodology for Information Systems and Software Engineering*, Roel J. Wieringa (ed.). Springer, Berlin, Heidelberg, 27–34. DOI:https://doi.org/10.1007/978-3-662-43839-8_3

[15] Hoe Tung Yew, Ming Fung Ng, Soh Zhi Ping, Seng Kheau Chung, Ali Chekima, and Jamal A. Dargham. 2020. IoT Based Real-Time Remote Patient Monitoring System. In *2020 16th IEEE International Colloquium on Signal Processing & Its Applications (CSPA)*, 176–179. DOI:https://doi.org/10.1109/CSPA48992.2020.9068699

[16] 2015. More than half of the global rural population excluded from health care. Retrieved May 4, 2023 from http://www.ilo.org/global/about-the-ilo/newsroom/news/WCMS_362525/lang--en/index.htm

[17] 2023. Docker overview. *Docker Documentation*. Retrieved May 18, 2023 from https://docs.docker.com/get-started/overview/

[18] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). Retrieved May 18, 2023 from https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines