Article

An Approach to Build e-Health IoT Reactive Multi-Services Based on Technologies around Cloud Computing for Elderly Care in Smart City Homes [†]

Luis Jurado Pérez * and Joaquín Salvachúa

Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain; joaquin.salvachua@upm.es

* Correspondence: luisalberto.jurado.perez@alumnos.upm.es

[†] This paper is an extended version of our paper published in the Proceedings of the 2018 International Conference on Parallel and Distributed Processing Techniques & Applications (PDPTA), Las Vegas, NV, USA, 31 July–2 August 2018.

Citation: Jurado Pérez, L.; Salvachúa, J. An Approach to Build e-Health IoT Reactive Multi-Services Based on Technologies around Cloud Computing for Elderly Care in Smart City Homes. *Appl. Sci.* **2021**, *11*, 5172. <https://doi.org/10.3390/app11115172>

Academic Editor: Ki-Hyun Jung

Received: 4 April 2021

Accepted: 25 May 2021

Published: 2 June 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Abstract: Although there are e-health systems for the care of elderly people, the reactive characteristics to enhance scalability and extensibility, and the use of this type of system in smart cities, have been little explored. To date, some studies have presented healthcare systems for specific purposes without an explicit approach for the development of health services. Moreover, software engineering is hindered by agile management challenges regarding development and deployment processes of new applications. This paper presents an approach to develop health Internet of Things (IoT) reactive applications that can be widely used in smart cities for the care of elderly individuals. The proposed approach is based on the Rozanski and Woods's iterative architectural design process, the use of architectural patterns, and the Reactive Manifesto Principles. Furthermore, domain-driven design and the characteristics of the emerging fast data architecture are used to adapt the functionalities of services around the IoT, big data, and cloud computing paradigms. In addition, development and deployment processes are proposed as a set of tasks through DevOps techniques. The approach validation was carried out through the implementation of several e-health services, and various workload experiments were performed to measure scalability and performance in certain parts of the architecture. The system obtained is flexible, scalable, and capable of handling the data flow in near real time. Such features are useful for users who work collaboratively in the care of elderly people. With the accomplishment of these results, one can envision using this approach for building other e-health services.

Keywords: cloud computing; container as a service; DevOps; e-health; emerging fast data architecture; Internet of Things (IoT); reactive system

1. Introduction

The World Health Organization (WHO) has estimated the growth in the number of people aged 60 and over will reach 2 billion by 2050 [1]. This gradual growth requires management and control strategies for this population group. The continuous growth of population aging is increasingly important since, for example, it forces medical centers to look for ways to reduce medical appointments to avoid possible saturation. Likewise, the increase in life expectancy requires new strategies to alleviate the concerns of family members who focus on the quality of life that elderly people could have inside the home. One of the fields required to manage these facts is e-health, which can facilitate the creation of health services, for example, care of vital signs, care of food or medication intake, and sleep care. Information and communication technologies (ICTs) can help ensure a certain level of well-being for elderly people while they are at home, leaving in the background the idea of moving elderly people to a care home. Some research on systems for the care

of elderly people has been proposed (see Section 2), but does not exploit the benefits of using reactive properties to facilitate the implementation of architectures for distributed pervasive systems. To date, several studies have presented healthcare systems for specific purposes without a common or explicit approach for the development of health services. Other studies do not take advantage of the use of the cloud computing to develop services in scenarios such as a smart city. In addition, the solutions of these studies often have scalability, interoperability, and extensibility issues.

On the other hand, software engineering has new challenges to address for the creation of innovative solutions based on new paradigms or technologies. Thus, some lines of research are related to the software process, software methodologies, design pattern development, privacy and security, and big data for software engineering [2]. Thus, software creation processes need appropriate strategies to manage the increasing complexity of the systems. One of the techniques used to deal with the design of applications is domain-driven design (DDD). DDD is used as a structured technique that helps manage the design of application complexities and establish delimited contexts to establish microservices.

Currently, the use of the Internet of Things (IoT), which is a form of implementation of pervasive systems, can provide novel functionalities through innovative breakthrough technologies. Some of the areas of application of the IoT are smart homes, smart cities, smart factories, agriculture, and healthcare [3]. Although some solutions of IoT systems have been carried out in the healthcare field, there are some challenges and open problems that need to be addressed, such as standardization, IoT healthcare platforms, the app development process, scalability, continuous monitoring, and data protection [4]. In recent years, mobile crowd sensing (or simply crowd sensing) has appeared as an IoT solution that relies on smartphone sensors to implement large-scale detection services [5], which requires incremental data storage to save sensed data in big data repositories [6]. These types of solutions are inexpensive compared to wireless sensor and actuator networks (WSANs).

The IoT enables the use of sensorization technologies to collect massive amounts of data from urban areas and offer community services to citizens, which makes it possible to offer intelligent city systems and services. The installation of different types of sensors in various environments and the emergence of new communication technologies, such as 4G/5G, allow the creation of systems with innovative functionalities to monitor and manage services such as electrical energy, water, household waste, public transport and traffic, the health of citizens, and public libraries. However, real-time analytics in connection with scalability and massive data transmission are research challenges [7].

The implementation of these distributed applications for a smart city requires high computing power provided by highly scalable hardware and software resources. In addition, distributed applications must concurrently provide interactive features for a wide variety of users and devices, provide batch and near-real-time processing and big data analytics and must be rapidly designed and developed. The principles of the Reactive Manifesto can be used to address these challenges and address the design and implementation of the system architecture [8]. The Reactive Manifesto describes a set of principles (responsive, resilient, elastic, and message driven) for designing reactive architectures and achieving computer systems that work in distributed, dynamic, and highly scalable environments.

It is vital to take into account that specific e-health applications require handling critical scenarios such as in the case of emergency management and obtaining medical reports in near real time, especially in chronic diseases. Due to these considerations, there is a need to use streaming data through near-real-time applications; therefore, this article promotes the use of emerging fast data architectures. This type of architecture allows ingesting, and continuous processing, and provides components for the design of a big data subsystem [9]. An architectural base of this type makes it possible to create data analytics services for batch and near-real-time processing.

Additionally, this paper also promotes cloud computing as an infrastructure as a service (IaaS) and encourages automation tools for deployments, such as Kubernetes (K8s) which helps to manage containers under the concept of “container as a service”

(CaaS). K8s is a portable and extensible platform for managing workloads and services. K8s automatically orchestrates useful computation, network, and storage infrastructure in DevOps processes [10]. Thus, it is possible to create and manage flexible DevOps processes with short development cycles and rapid deployment of distributed applications (microservices) that have high computational requirements.

The main idea behind the approach of this paper is to propose a set of methods, principles, and architectural decisions to enable the design and implementation of e-health reactive services that can be used for elderly people in smart city homes (Figure 1). The software components of the services were based on microservices, and some of the services incorporated crowd sensing as an IoT sensing solution to meet the pervasive system requirements. Moreover, a set of architectural decisions was proposed to have continuous integration of the services code in version repositories and continuous deployment of services in a public cloud. Furthermore, the use of a fast data architecture and the incorporation of a big data subsystem for supporting batch and near-real-time data analytics were considered.

Figure 1. Information and communications technology for health services.

This article makes the following main contributions:

1. An approach to build cloud-based e-health IoT reactive multi-services as a feasible solution;
2. The implementation of the system built on an emerging fast data architecture is proposed with open-source components. This architecture incorporates a big data subsystem for data analytics in batch and near-real-time modes;
3. Based on the proposed architecture, an e-health system prototype is deployed in a public cloud incorporating a continuous integration and continuous deployment (CI/CD) pipeline, and several experiments are conducted to evaluate the performance.

The results show that, through the proposed approach, it is possible to build reactive e-health multi-services on a flexible architecture deployed in a public cloud, allowing scalability and low latency in the critical management of near-real-time data flow. In light of these results, one can envision using this approach for the design, development, integration, and continuous deployment of other types of reactive services in the field of e-health, allowing applications for the various activities inherent in the lives of elderly people.

Organization. The remainder of this article is organized as follows. Section 2 provides a brief overview of the related work with the design, implementation, and deployment

of some e-health applications. The approach used for the design, implementation, and deployment of the system is presented and discussed in Section 3. In Section 4, the authors introduce the system architecture considering services as an integral part. The implementation of the system prototype is presented in Section 5. In Section 6, an evaluation of some parts of the system is carried out. Finally, conclusions and future work are shown in Section 7.

Extended part. The current paper extends our previous work [11] considering the extension of the approach to achieve an agile development of the system software services and a flexible deployment of the system components as containers in the environment of a public cloud. Major extended parts are as follows. In Section 2, we update the list of related works and provide an extended set of relevant technological features in healthcare systems. In Section 3, we expand our approach to propose a flow of integration and deployment of software versions on a public cloud through an approach to CI/CD of DevOps practices. Additionally, a set of Kubernetes patterns was applied for container orchestration. A new and more complex use case is introduced to show the flexibility of the architecture to handle multi-services in Section 4. This last use case is related to services for the management of diets. In Section 5, the implementation of the system is extended around CaaS and K8s. The new results obtained from the study of the scalability of the system services deployed in a public cloud computing environment are shown and discussed in Section 6. Additionally, we present an extended test of the emergency service related to the response time to communicate a situation of emergency.

2. Related Works

In recent years, research in the field of healthcare systems has provided independent solutions that present a variety of issues. Particularly, these different healthcare system initiatives present little or no interoperability, scalability, and extensibility. Our work is motivated by some previous research and complements it in many ways.

A healthcare monitoring system based on an architecture focused on grouping and providing interoperability between healthcare sensing devices was proposed in [12]. Data on health conditions are reported through a web application and mobile application. However, the architecture lacks the flexibility to adapt and scale to the demands of the number of users of a smart city.

A framework for the design of a smart health monitoring system was presented in [13]. The main objective of this framework was to enable ubiquitous monitoring of various population groups, including elderly people. The framework includes basic components such as real-time data extraction, wireless communication controllers, and wireless communication controllers. However, the framework is not validated by any prototype, and strategies for managing scalability or a big data subsystem are not proposed.

A specific solution to monitor the heart rate for hypertensive patients combining the benefits of some technologies such as ZigBee, Wi-Fi, and a web application was proposed in [14]. Although the system had some reactive features through the use of messaging subsystems, the high scalability and availability of the system were not taken into consideration.

A system that captures data from wearable sensors at home and sends them to a cloud-based web server was proposed in [15]. The applications were based on REST web services. However, considerations for real-time monitoring in a smart city were not considered. Additionally, the architecture did not cover the aspects related to the management of a large amount of data.

A system for continuous monitoring of patient respiration was developed in [16]. An Arduino controller transmitted the data to a web server that stored the data in a MySQL database, and a web page was used to view the data from a monolithic web server. However, the system did not incorporate reactive capabilities, such as messaging systems, component replication, or the use of microservices as opposed to the use of monolithic web servers.

A fusion between the IoT and cloud computing was used to implement a patient health monitoring system in [17]. This framework provided continuous on-demand monitoring. Although mechanisms were proposed for the analytics of the data collected, the architecture did not incorporate specific solutions for the treatment of big data and real-time analytics. The architecture of the software applications was not explicitly discussed.

The authors in [18] proposed a specific model for the creation of sense electronic health records. This solution used Bluetooth to connect to sensors, and a smartphone sent the data to the EHR server using a RESTful API. However, the coupling between smartphones and the EHR server affects the scalability of the system.

A smart health system assisted by cloud computing and big data was presented in [19]. The system included a data collection layer with a unified standard, a data management layer for distributed storage, and a data-oriented service layer. On the other hand, it presented a weak reuse, and the system did not guarantee the integrity and interoperability of the data in the environment of its operation.

The authors in [20] proposed a prototype design of a health monitoring system for patients in healthcare. Some vital signs were retrieved by low-cost hardware, transferred to the cloud computing environment, and processed with big data technologies. In the prototype, the mechanisms and products used to transfer the data from the sensors to the cloud computing environment were not fully explained, especially the interaction between the messaging brokers. Although the system used some reactive components, the strategies or mechanisms to achieve scalability and availability of the system in an environment with a large number of users were not explained.

An end-to-end framework for big data storage and analysis for batch and real-time processing in the context of an ECG monitoring application was presented in [21]. The system was designed in the context of Amazon Web Services with products owned by Amazon. However, the implementation lacks the flexibility to handle containers with open-source products. On the other hand, the architecture did not incorporate DevOps practices for CI/CD of the services that are of interest to the developers of the system.

Furthermore, in relation to systems for the control of food intake, some models of computer systems for the prediction and control of diets have been developed. These systems assign a diet to a person with a particular health condition such as obesity or diabetes [22,23]. An expert system based on an ontology for the care of the nutrition process for elderly people was presented in [22]. The system tests were carried out inside the laboratory to validate an inference engine to assess the nutrition problems and the comparison of the results provided with those given by the nutritionist. However, the architecture does not present characteristics that allow the scalability of the system, or the real remote monitoring of food intake. The purpose of the research in [23] was only characterized through a zone class of blood potassium levels; monitoring and control were not addressed.

Moreover, there are studies related to the control of feeding people for various purposes, mostly for weight control, and some relevant technologies are used in some of them [24,25]. The work in [24] proposed a mobile app to help parents and doctors monitor children who suffer from high obesity rates. The IoT application allows tracking of food intake, remote capture, and constant monitoring of children's data. Although the children send information about the intake through a mobile application to an application server, the intake control is in the nutritionist's hands. Additionally, the mechanisms to provide high scalability and high availability to the system are not addressed. The work reported in [25] presented a model that collects data from sensors and social networks to provide monitoring and to prevent obesity, including tracking food intake, lifestyle, exercise activities, generating warnings, and triggering interventions whenever needed. The real controlled planning of food intake does not arise. Although there are data mining processes, the system does not provide solutions for handling the big data produced by this type of system. Additionally, the components of the monitoring process are highly coupled, and the system does not show evidence of scalability.

The objective of our work is to propose an approach to develop cloud-based e-health IoT reactive services as a distributed system for users who work collaboratively in the care of elderly people. Our approach includes the use of architecture with scalability and interoperability between its components. In addition, the use of certain design patterns allows the system to grow flexibly to provide new subsystems for healthcare.

Unlike some of the studies discussed above, we promote the use of crowd sensing [5,26] for collecting data as opposed to using a dedicated infrastructure such as WSANs that are too expensive. The smartphone can act as a gateway that takes data from a wireless body area network (WBAN) through light protocols such as Bluetooth. Additionally, we promote the use of the MQTT protocol incorporated into smartphones for the exchange of messages with other parts of the system.

The use of a certain type of fast data architecture has been used minimally without enhancing its usefulness. As part of our system architecture, we built a fast pipeline to obtain real-time monitoring and additional strategies and mechanisms for the interoperability and scalability of its components. Thus, we use an emerging fast data architecture that supports big data for batch and near-real-time data analytics. This incorporates reactive characteristics through the use of messaging systems providing load management, elasticity, flow control, and back pressure control.

Regarding the implementation of services, some systems used a certain type of service as a monolithic application, and other investigations used some groups of RESTful web services for specific requirements. In our research, we incorporated the architectural pattern of RESTful microservices using DDD as a basic design principle. Thus, we promote the construction of reusable components per the domain of service groups through low coupling and modularity.

None of the healthcare systems analyzed incorporate DevOps practices. Our approach promotes the incorporation of CI/CD pipelines to provide an architecture that makes the integration of the software development and deployment phases more flexible. Most of the analyzed studies use the cloud computing environment only to locate the final application and its components, such as the database. The strategies and mechanisms to provide interoperability, scalability, and availability are not addressed. In our architecture, we incorporate the flexible use of containers through cloud services around the use of “CaaS” and Kubernetes. The architecture of our e-health system in the cloud computing environment provides interoperability, scalability, and availability and is loosely coupled, distributed, and elastic.

Currently, the adoption of computer technology can help the development of systems in the healthcare field in many ways, such as software management or cost reduction. However, the use of cloud computing is in an initial stage in the field of e-health. In Table 1, we compared our system with the related studies addressed in this section using different categories.

Table 1. Related work.

Reference Number	Monitoring (Sensors)	Emergency Situations	Monitoring: WSN/Wireless/3G/4G/Bluetooth or Others	Near Real-Time Communication	Scale for Smart City	Characteristics Related to Big Data	Near-Real-Time Analytics	Architecture Implementation	Services Implementation	CI/CD of DevOps Practices	Container as a Service
[12]	Y ¹	Y	Wearable sensors, 3G	Y	N ²	N	Y	Y	Web services (JSP, servlet, EJB, MDB, and JDBC)	N	N
[13]	Y	Y	ZigBee-based WSN, Bluetooth, Wi-Fi	Y	N	N	N	N	N/A	N	N
[14]	Y	Y	IEEE 802.11 Wi-Fi and IEEE 802.15.4 Zigbee	Y	N	N	N	Y	JAVA web application	N	N
[15]	Y	Y	Bluetooth, CoAP, 3G	N	N	N	N	Y	RESTful web services	N	N
[16]	Y	Y	Ethernet Shield, Wi-Fi, IEEE 802.15	Y	N	N	Y	Y	Web server (PHP, Apache), HTML Code	N	N
[17]	Y	Y	BSN/Wi-Fi/LTE, Bluetooth	Y	N	N	N	Y	N/A	N	N
[18]	Y	Y	Bluetooth, 3G	Y	N	Y	Y	Y	RESTful API, JSON, HTML5	N	N
[19]	Y	Y	3G, Wi-Fi	Y	N	Y	Y	Y	N/A	N	N
[20]	Y	N	Wi-Fi 802.11 b/g/n	Y	N	Y	Y	Y	Simple RESTful web service	N	N
[21]	Y	Y	IEEE 802.11.b/g/n/ac	Y	Y	Y	Y	Y	Amazon Web Services (AWS)	N	N
[22]	N	Y	N/A	N	N	N	N	Y	Protégé, RDF API, D2RQ	N	N
[23]	N	N	N/A	N	N	N	N	N	N/A	N	N
[24]	Y	Y	Bluetooth, Wi-Fi, 3G	Y	N	N	N	Y	REST Web Services, PHP, Java script, HTML5, CSS	N	N
[25]	Y	N	Wi-Fi, GPRS, 3G, 4G, Bluetooth	Y	N	N	N	Y	RESTful web services and JSON	N	N
Our work	Y	Y	Bluetooth, 4G	Y	Y	Y	Y	Y	RESTful Microservices	Y	Y

Y means YES and N means NO.

3. Methodology

The proposed approach of this paper for the construction of the e-health system follows a methodology based on the use of the software architecture process by Rozanski and Woods [27], taking into consideration the identification and use of a set of fundamental architectural patterns. Additionally, the use of the “Reactive Manifesto principles” (responsive, resilient, elastic, and message driven) was taken into consideration to provide the system with high reliability and scaling features [8]. In addition, the characteristics of the emerging fast data architectures are integrated into the system design [9]. In this way, the use of a basic configuration of a subsystem of big data is incorporated for the treatment of the data collected by the IoT system, which extends the possibility of creating services related to data analytics in batch and real-time modes. Moreover, the DDD is used to divide the various domains of the system into microservices [28]. Furthermore, some activities such as the development and control of microservice versions, the use of containerized microservices, and microservice deployment were carried out by integrating workflows through DevOps practices. In this way, it was possible to facilitate continuous integration and continuous deployment [29]. Finally, the location of each of the system components as containers in the cloud computing environment requires effective orchestration. This orchestration was done with the cluster manager for containers called Kubernetes. Thus, a set of cloud-native patterns for configuring and placing containers on the cloud computing environment was applied [30].

3.1. IEEE 1471

The iterative process of architectural design by Rozanski and Woods is based on the IEEE 1471 standard. Through this process, the fundamental elements of the system software architecture were developed using a series of architectural views to define the architectural description [27].

3.2. Architecture Patterns

To address some of the fundamental characteristics of architecture, it is possible to apply some popular architectural patterns.

3.2.1. Layered Architecture Pattern

The layered architecture pattern was used to break down the tasks of the system into a series of interrelated subtasks. Each of these tasks was represented by a layer that had two communication interfaces, the upper interface providing services to its top layer, and the lower interface providing services to its lower layer [31,32].

3.2.2. Message-Oriented Broker Pattern

This asynchronous communication pattern permits concurrency and high scalability and can be used as a data distribution intermediary for other components called publishers and subscribers. The data that come mainly from the sensors are sent to the distribution intermediary (the broker), thus achieving a decoupling of the readings and writes of the data [31,32]. In general, system components such as mobile devices could take advantage of publishing data in these messaging brokers that can be consumed by other parts of the system.

3.2.3. Microservice Architecture Pattern

The microservice architecture pattern is a distributed architecture based on decoupled components that provide resilience, scalability, and ease of deployment. System applications were based on the design principles of the microservice architecture pattern based on RESTful web services. The use of the microservice architecture pattern made it possible to have a manageable set of small pieces of software. In this way, software development was carried out through a microservice-based programming infrastructure to implement the services of system applications, facilitating agile integration into the system. In gen-

eral, the microservice architecture pattern allowed decoupled components and facilitated understandable independent tasks to deploy, scale, and test each of the services [33].

3.2.4. Model-View-Controller Architecture Pattern

As part of the approach, the MVC pattern was used in the modeling of web applications to design user requests (multiple tasks and interactions) to the system core or to specialized applications through its electronic devices such as personal computers, laptops, and smartphones, which support graphical interfaces. The interactive software system was divided into three fundamental components: (1) a model component that handles the data of the system, (2) a visual element that plays a primordial role in the human-machine interactions presenting the data, and (3) a control part that handles the inputs of the users, and is the intermediary between the model part and the visual part [31].

3.2.5. Cloud Computing Paradigm

In general, the cloud computing paradigm offers a solution to fulfill computational economics, web-scale data collection, system reliability, and scalable performance [34]. The cloud deployment model called “public cloud” was used as a valid environment for the deployment of the system components. The “public cloud” service is provided by cloud computing providers such as Amazon, Google, Microsoft, IBM, and Oracle [35], which offer a broad spectrum of services to meet the architectural requirements for systems implementation.

Although the use of typical services such as infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) is available, other types of additional services located between IaaS and PaaS have emerged and can be used to facilitate the implementation of container-based systems. In this work, the use of the public cloud was minimized due to its costs. However, it encourages the use of concepts related to “container as a service” (CaaS) that help manage containers through Kubernetes.

3.3. Reactive Principles

These principles help to make a high-level abstraction and to provide reactive characteristics to the system [8]. These four principles are:

1. Responsive: Response times must be consistent with the needs of the applications with a certain quality of service;
2. Resilient: Some features, such as replication, containment, isolation, and delegation, should be system objectives to maintain response even in the case of any failure;
3. Elastic: Requests to the system must be addressed with replicated components to prevent bottlenecks or containment points;
4. Message Driven: The use of asynchronous messaging queue systems and the design of some form of back pressure facilitate load management, elasticity, and flow control, which provides low coupling, isolation, and location transparency.

3.4. Fast Data Architecture and Big Data Architecture

The architecture of the IoT system was based on an emerging fast data architecture for the intake and continuous processing of the data. This type of architecture provides more stream-oriented features and is desirable to perform near-real-time data analytics [9]. A big data subsystem was also incorporated into the system architecture to store huge amounts of data from the IoT subsystem. The big data subsystem enables the creation of near-real-time data analytics services or batch data analytics services.

3.5. Domain-Driven Design (DDD)

DDD is an exciting way to structure a microservice. Thus, DDD was used in the design of the system’s e-health applications since, from the logical point of view, it allowed decision making to determine the various functional domains of the system. Additionally, DDD facilitated the distribution of system domains and the coordination of their parts.

Through bounded contexts, domains were segregated, and each of these domains was modeled through microservices [28].

3.6. DevOps and Containerization Ecosystem

An automatic CI/CD pipeline for system software was proposed to provide a dynamic workflow to software development and deployment processes. Thus, to reduce workflow times between the processes used in the development environment and the operations environment, some additional tools were used. These tools were a distributed software version control system with integration and continuous deployment [36] and a system for building container images [37,38]. All application components, such as core services, specialized services, and other system elements developed by the authors of this paper, were deployed as containers in a public cloud.

3.7. Kubernetes Patterns

A set of patterns and practices for container orchestration is useful to take advantage of common cloud services (deployment, scaling, load balancing, logging, monitoring, etc.). K8s provides the highly available cluster startup and enables management of certain resources such as pod resources and service resources. These resources can be used to form design patterns that help make architectural decisions to cover architectural aspects of systems on the cloud infrastructure [30]. In this paper, some of these patterns have been used for the deployment and configuration of the system components.

4. System Architecture

4.1. Scope and Context

One of the techniques suggested by the iterative process of architectural design by Rozanski and Woods [27] is the use of scenarios, which was used for the establishment of requirements, the definition, and validation of the scope of the system. These requirements were used for the development of the architecture description, the construction of a prototype, and the proofs of concept. In order to show the possibility of building multiple applications and services, and restricted by time, only two scenarios are presented that were used for the realization of the architecture description. However, additional applications and services in the health field could be conceived and built through our approach.

4.1.1. Scenario 1: Basic Services for the Management of the System, Monitoring Services, and Services for Alerts and Emergency Management

The first scenario tries to establish a series of fundamental concerns related to the monitoring of vital signs. An elderly person lives in a house of predetermined dimensions, and the spaces of the house can be expressed in a two-dimensional plane. Elderly people can move freely in their homes, and at the same time, their vital signs are collected through a series of wearable devices and sent to a smartphone (gateway). Additionally, the room temperature is collected by a sensor on a smartphone. The smartphone acts as the final transmitting element of all sensed data.

On the other hand, medical staff or family members can know the status of vital signs or the room temperature in near real time through remote devices, for instance, computers, smartphones, and tablets. Permanent control of changes in vital signs by the system is carried out to activate medical alerts. These medical alerts are vital to initiate emergency medical protocols if necessary. Authorized users can access the system to: consult patient data, consult the characteristics of the house, check the temperature status in the house, and monitor the vital signs of elderly people in real time.

4.1.2. Scenario 2: Services for the Management of Diets

Scenario 2 is related to the care of food intake within the home. The primary objective of these services is to help people to maintain healthy nutrition. Special considerations for the treatment of diets related to specific diseases are beyond the scope of this work.

The main actors of the system are the nutritionist and elderly people who follow the diet. Moreover, a third actor could exist, a housekeeper who helps with household chores. Hence, a minimum set of requirements was considered for the system components related to the prediction, planning, control, and monitoring processes of diets:

1. Digitized Diet Catalog. The art of designing diets, types of food, and nutritional components is beyond the scope of this article, so the diet catalog is based on the design of nutritional intakes provided by the healthy Mediterranean-style eating pattern and healthy US-style eating pattern. These patterns are described in the dietary guidelines for Americans 2015–2020 [39]. The healthy US-style pattern includes 12 calorie levels to meet the needs of individuals across the lifespan. The pattern is used for the creation of a digitized catalog of diets by building schedules of daily intakes during each week, following the distribution of the recommended daily calories for a person;
2. Prediction of the diet. The diet prediction uses the dietary reference intakes (DRIs) that are published by the Food and Nutrition Board of the Institute of Medicine, and they are intended for healthy people in the United States and Canada. Thus, the prediction of diets is based on the estimated energy requirement (EER). The EER is defined as the average dietary intake that is predicted to maintain energy balance in a healthy adult of a defined age, gender, weight, height, and physical activity level (PAL) consistent with health [40];
3. Diet planning. The catalog and its basic plans help the nutritionist to plan a diet for a person by making minimal changes in the proportions of food. Through the system, the nutritionist can assign a weekly plan and meal schedules to dieters;
4. Monitoring and control of the diet. The monitoring and control cycle is as follows: the housekeeper chooses the diet and prepares it every day. The system provides services such as consultation of recipes and videos for the preparation of meals. Once the person completes the food intake, the housekeeper confirms the completion of the intake to the system by sending two quick response codes (QR codes), ingested food and elderly person ID, and the value of the amount ingested through the mobile crowd sensing components. Finally, automatic alerts are sent to users if dietary plans are not carried out.

4.1.3. Definition of System Requirements

Table 2 shows a summary of the set of requirements to be addressed to satisfy the needs of the proposed scenarios. It does not attempt to provide a complete list of requirements.

Table 2. System requirements.

#Req.	Requirements
R1	Provide a wireless body area network (WBAN) to enable the collection of medical data through monitoring devices that the elderly person wears comfortably. This network must be part of a data collection component within the home and must be customizable, transparent, and non-intrusive.
R2	Make it easier for monitoring devices to send data to a device that acts as a gateway, which should be part of the data collection components within the home. This gateway should allow constant communication with the elderly person's medical data collection sensors. This gateway must be customizable, transparent, and non-intrusive.
R3	Provide the communication components to transport the data from the monitoring process of the elderly person to the secure data repositories.
R4	There must be a flexible integration and communication of the different components of the system.
R5	Provide the distributed software services with facilities for maintaining the data of the different data entities of the system (elderly people, sensors and actuators, catalog of diets, family members, etc.) linked to the various applications for care of the health of the elderly person.
R6	The system must have the mechanisms for real-time control of the medical conditions of the elderly person and the generation of alerts that communicate emergencies. The emergency notifications can be used by the people who take care of the elderly person to execute the action protocols for immediate medical assistance.
R7	The system must have a repository for the safe storage of data: personal data of the elderly person, data of family members, medical data from monitoring, basic data of sensors or actuators, etc. This data repository must be scalable, adaptable, flexible, and support optimized data queries.
R8	Allow consultation of the medical data or medical condition of the elderly person remotely from anywhere/at any time, through software applications through electronic devices such as computers, tablets, or smartphones. The graphical interfaces of these software applications must be flexible and friendly.
R9	Provide the system with the elements for the treatment of big data. These components should have the ability to facilitate data analytics in real time and in batch mode.
R10	Apply security mechanisms to the entire system.
R11	Facilitate the development of new services and incorporate a flow of integration and deployment of software versions on the cloud computing infrastructure.
R12	Make the deployment of system components more flexible on a public cloud. The public cloud should facilitate the deployment and execution of containers, the management of clusters, and the automation of scaling.
R13	Provide reactive characteristics to obtain a flexible, low-coupling, and scalable system. A reactive system is easy to develop and upgrade, fault tolerant, and highly responsive.

4.2. Architecture Description

4.2.1. Context View

The system context view describes the relationships, dependencies, and interactions between the system and its environment (people, systems, and external entities). Figure 2 shows the context view. Stakeholders want to know the utilities and benefits of the system to enhance their daily activities. Some examples are indicated below:

- A doctor could immediately carry out medical follow-up and report the vital signs of a specific patient who is at home;

- Pharmacies may know if certain products should be provided if there are people in the area who use specific medications;
- The food stores could offer products oriented to the needs of the diets followed by elderly people;
- Medical research institutes can use the data collected by the system to develop plans that contribute to research results in the science and medicine field.

Figure 2. Context view.

4.2.2. Functional View

Figure 3 shows the system as a set of interrelated layers. The layers represent the various components that the system needs to connect to achieve a valid interaction between them and thus provide the implemented health services.

Figure 3. Layer view.

These layers must have a secure access configuration, and the design of each of them considers the system services as a set of end-to-end elements. Additionally, there is a set of microservices that can be used by the users of the system. The layers considered in the architecture are perception, communication, infrastructure, orchestration and containerization, middleware, and application. The perception layer includes the sensors, the QR code reader, and other devices that could be included in future applications such as video sensors or radio frequency identification (RFID) tags. The infrastructure layer reflects the use of the services offered by the public cloud, and the orchestration and containerization layer covers the use of Docker container images to be installed on the Kubernetes cluster.

Finally, another relevant aspect is the inclusion of fast data features in the middleware layer for real-time analytics and real-time ingestion, which are useful in capturing big data.

Figure 4 shows the functional components and their relationships. System functionality has been divided into the following set of components:

1. The Core System: It consists mainly of:
 - Elderly People Manager: Allows managing the personal data of elderly people;
 - User and User Groups Manager: Its main functions are to create profiles and security levels, create user-type profiles, and create users or groups of users of the system. It is only managed by the system administrator;
 - Home Manager: Allows creating data related to the elderly person's home such as location, number of rooms, types of rooms, free zones, and dimensions;
 - Sensors Manager: Permits recording of the data about features of the sensors.
2. Monitor and Control Manager: It consists mainly of:
 - The sensor monitoring and control subsystem to collect the vital signs and the temperature status of the room inside the home;
 - The input/output transporter whose function is to transport the sensing data in near real time (real-time ingestion);
 - The functional elements of the system that allow the parameterization and configuration of the input/output transporter, such as the creation of topics in distributed publish/subscribe systems;
 - The system elements for near real-time data analytics (real-time analytics).
3. Data Storage Manager: The functional component that stores the data used by the system. It is a conventional database management system. Its goal is to keep the data in a secure repository.
4. Diet Manager: It consists of the following components:
 - Diet Catalog Manager: Used to manage all the data in the digitized catalog;
 - Diet Prediction Service: A set of operations for updating the physical data of an individual. These data are age, weight, height, and physical activity level (personal physical data). It also contains the functional element that performs the predictor role based on the EER of elderly people;
 - Diet Prescription Service: Helps the nutritionist to assign a diet label to a person based on the diet proposed by the EER predictor and establishes the periods of the diet;
 - Diet Preparation Service: The group of operations that are specially designed for the housekeeper, who is the person in charge of preparing meals in the elderly person's house. It also provides videos and the steps for preparing menus through recipes to help prepare meals of the day;
 - Service for Food Intake Monitoring: A set of operations used to record in the system the start and end of food intake. These data are sent from a mobile application to a system using the IoT messaging system. The start of food intake is recorded by sending the QR code of the diet menu stored in the catalog. Finally, the end of food intake is indicated by sending the elderly individual ID and the approximate percentage value of the food ingested. During the assignment and

follow-up of a diet, the services execute functionalities that help keep records of the actions performed, as summarized in Figure 5.

5. **Alert Manager:** Helps create and maintain all kinds of system alerts related to the status of vital signs or the incidents that could occur in the follow-up of the diet plan. It consists of a component (alert engine) in continuous operation with the ability to trigger alerts within the system based on the analysis of the various data received from the home.

4.2.3. Information View

The information view shows the data structure in the system. The analysis of the characteristics of the data entities was carried out to obtain a diagram of their relationships. The diagram obtained reflects the ad hoc needs of the system requirements. The data model, with its entities and their relationships, is shown in Figure 6.

The data model has two groups of entities which are related to the subapplications of the system. The common entity of these subapplications is the elderly people entity which is the main entity of the system. The description of the use of each of these entities is shown in Appendix A.

Figure 4. Functional view.

(a)

Health Care Home - UPM Project

Identification Document Required

Illness Required

Age Numeric

HeightCm Numeric

WeightKg Numeric

Registration Date Date ('yyyy-MM-dd')

Update Date Date ('yyyy-MM-dd')

System Status Required

[Create this Patient Primary Data](#) or [Cancel](#)

Technical University of Madrid - DIT

(b)

Figure 5. Cont.

Figure 5. (a) Diet prediction, prescription, and monitoring process. (b) Graphical user interface to collect the personal physical data of the elderly person. (c) Graphical user interface for calculating the EER. (d) Graphical user interface that helps to prepare meals.

Figure 6. Data structure.

4.2.4. Development View

This view shows the local development environment where programmers can write code for software services and perform preliminary tests. Figure 7 shows how the local computer of the members of the development team becomes a reduced test environment that includes the installation of the parts of the system as containerized components from the Docker Hub. Additionally, third-party software products can be installed locally and interact and verify the functionality of the software components.

Figure 7. Development view.

The developer workflow is complemented by operations performed to maintain version control of software objects created for the system. The workflow of the development team members can be updated and integrated collaboratively. Here, one of GitLab's features is leveraged, allowing project management and modern source code management through issue tracking and continuous integration. For additional details of the development view, see Appendix B.

4.2.5. Deployment View

The deployment view represents the preproduction environment of the system components located in the cloud infrastructure. In this paper, we use Google Cloud Platform services to define a basic K8s cluster with minimal technical characteristics. The characteristics of the K8s cluster and cluster nodes are shown in Appendix C. The e-health system component deployments were carried out with the main K8s elements: pod resources and service resources. The pods are part of an internal communications network of the cluster. The K8s service resources provide the mechanisms for the communication of the pods inside or outside of the cluster. The advantage of placing components that work closely with each other in the same K8s cluster is because the network and subnets are defined at the cluster level. Although it is possible to use additional K8s clusters, the placement of a component of a part of the system in another independent cluster implies the introduction of unnecessary latencies in the system.

We created two groups of nodes within the cluster to organize the e-health system in two parts. Each group of nodes was dedicated to providing sufficient exclusive resources for each of the parts of the system. Additionally, this organization facilitated the independent analysis of scalability and performance. Figure 8 shows an abstraction of the deployment view. The e-Health-System-K8s-cluster nodepool-1 includes the microservices of the subapplications and the different components for its control of continuous integration and continuous deployment. The e-Health-System-K8s-cluster nodepool-2 includes the software used by the system as third-party software, which was downloaded from the Docker Hub. The e-Health-System-K8s-cluster nodepool-2 mainly contains the messaging subsystem.

Figure 8. Deployment view.

Some K8s organization and configuration patterns were used for the deployment of software components, such as multiple availability zone design, single containers, automated placement, stateful service, service discovery, environment variable configuration, etc. For a detailed analysis on the use of the most relevant patterns used, see Appendix D. For additional details of the deployment view, see Appendix E.

5. Implementation Details

For the implementation of the system prototype, two environments were taken into consideration: the development environment and the preproduction environment. The development environment represents the local machines of system developers and integrators, and the preproduction environment represents the prototype of the fully deployed system using the underlying infrastructure provided by Google Cloud infrastructure. These environments are related through the sharing of versioned software in GitLab. Moreover, both environments use some system components obtained from the Docker Hub. In general, the components of the system architecture are a set of open-source technologies used to facilitate the construction of emerging fast data architectures and satisfy the functional requirements of the system. To deploy the architecture, we have chosen the closest data center (London) to our research point (the city of Madrid). The following section describes the prototype components that were used in the preproduction environment (Figure 9).

Figure 9. System architecture.

5.1. Google Cloud Platform and Kubernetes

In this work, GKE was used as a management service for running Kubernetes clusters on the Google Cloud [41]. Although K8s is not a full conventional PaaS, K8s was used as a flexible platform to manage system components as containers. K8s functionalities aid in the deployment, logging, monitoring, scaling, and load balancing processes [42]. Following the deployment view, the system's preproduction environment was implemented with a K8s cluster version 1.16.13-gke.401 which we have called the e-Health-System-K8s cluster.

5.2. Health Applications of the System

The system is a set of subapplications, each of which was considered as a set of microservices. These subapplications are web applications (REST web services) and were developed with Play Framework v2.4.8 [43] and the Scala programming language v2.11.8. Play Framework is an MVC framework and is based on a lightweight, stateless, web-friendly architecture for highly scalable reactive applications. The user interfaces (MVC views) were developed using HTML5, CSS3, JavaScript, Bootstrap, and JSON. Figure 10 shows the microservices of the e-health system.

Figure 10. The microservices of the e-health system are designed with an MVC design pattern. The graphical user interfaces provide access to core services and diet management services.

The building of the images of the microservices of the subapplications is part of the CI/CD pipeline that includes the storage of the images in the GitLab container registry. With these facilities, the process of deploying the CI/CD pipeline is performed automatically from the GitLab container registry to the e-Health-System-K8s-cluster nodepool-1. In Kubernetes, the pods of the subapplications were configured to provide scalability based on the increase in requests to the system. External exposure to the services of e-health applications was made through an ingress resource that allows access to multiple services through a single IP address. For additional details of the graphical user interfaces for access to services, see Appendix F.

5.3. Data Storage Infrastructure

MongoDB is an open-source distributed and document-oriented database [44]. It ingests and stores data in near real time and in an operational capacity. MongoDB is a distributed database with high availability and horizontal scaling, and is easy to use. MongoDB is fully scalable compared to traditional relational databases like MySQL [45].

In the prototype, MongoDB v3.4.5 was used, and a MongoDB cluster was deployed on e-Health-System-K8s-cluster nodepool-1 to facilitate access to data from subapplications. The deployed containers used images from the Docker Hub and were deployed using properly configured YAML Ain't Markup Language (YAML) files.

5.4. EMQ Cluster

MQTT is a lightweight publish/subscribe message protocol, and its implementation is useful for the collection of data coming from devices with limited resources. In this work, the message broker EMQ v4.0.0 was used to build a cluster of EMQ brokers. EMQ broker is a distributed, massively scalable, highly extensible MQTT message broker [46]. The EMQ cluster is used as the first point of entry for data that come from the homes of elderly people.

For the architecture implementation, we deployed the EMQ cluster with three nodes in e-Health-System-K8s-cluster nodepool-2 (see Section 4.2.5) with images from Docker Hub.

5.5. Confluent Platform

For the system to offer services to a large population group in a city, it is necessary to incorporate a high-capacity messaging communication channel into its architecture. Due to the medium scalability of the EMQ (millions of messages), the Confluent platform has been incorporated into the communications channel [47]. In that way, it is also possible to reduce the back pressure that can be exerted by the EMQ cluster. Confluent improves Apache Kafka and supports trillions of messages, which is useful for highly scalable application in a smart city. For the prototype, an installation of Confluent v5.1.0 with an Apache Kafka cluster with three nodes and an Apache Zookeeper cluster with three nodes was deployed on the e-Health-System-K8s-cluster nodepool-2. Confluent deployment was done using Helm charts by downloading images from the Docker Hub [48].

5.6. Apache Spark

The characteristics of Apache Spark v2.4, such as streaming processing, high speed, SQL language, and using in-memory distributed computing, together satisfy the requirements of the system. Although Apache Spark allows data processing in batch mode and near-real-time processing, in the prototype, an Apache Spark cluster was only used as the near-real-time processing component to support the alert manager (Section 4.2.2). Another design alternative with similar characteristics but of almost equal capacity is possible using Apache Flink. On the other hand, unlike Hadoop, Apache Spark could be used in batch and near-real-time modes [49]. The alert manager runs as a Spark application deployed on the e-Health-System-K8s-cluster nodepool-2 through YAML files and images from the Docker Hub [50].

5.7. Connectors

A set of additional elements was used to connect some components of the system:

5.7.1. MQTT-Kafka Connector

This is a reactive connector that has been specially created by the authors of this paper to connect the data flow between the EMQ cluster and the Apache Kafka cluster. It was implemented using Scala programming language, Akka Streams-Kafka library [51], and Paho-Akka library [52]. An image of the container with the connector was pushed to Docker Hub. The connector was deployed through YAML files on the e-Health-System-K8s-cluster nodepool-2.

5.7.2. Kafka-MongoDB Connector

This writes events from Kafka to MongoDB. It is a Kafka Connect Mongo Sink which helps to quickly and safely store the data. Kafka Connect (in distributed mode) and the Mongo Sink connector are used to carry out this data transfer from the Apache Kafka cluster to the MongoDB cluster. This connector is part of Lenses.io v4.0. The Mongo Sink was installed on the e-Health-System-K8s-cluster nodepool-2 through Helm charts provided by its vendor [53].

5.7.3. Spark-Kafka Connector

This is a Spark-streaming-Kafka package that connects Apache Kafka and Apache Spark [54]. In this paper, we use the connector version v0.10 and it is incorporated into the alert manager to perform data analytics in near real time. Additionally, the data can be analyzed via SQL, and the analysis results can be stored in MongoDB or sent to a dashboard.

5.7.4. MongoDB Connector for Spark

This connector facilitates the implementation of services that can analyze data extracted from MongoDB. Furthermore, it allows the management of resilient distributed datasets (RDDs) to minimize data extraction and reduce latency [55]. In this paper, version 2.2.0 of this connector was used. Additionally, other applications can be conceived, such as those that could use this connector together with the connector of Section 5.7.3 to process data in batch mode and obtain machine learning models.

5.8. Mobile Technologies

Our e-health system considered the use of smartphones and mobile applications developed following the layering model of the Android Architecture Platform [56]. The system considers two types of smartphones: those used by elderly peoples and those used by paramedics. The smartphone used by an elderly individual contains two apps. The first app supports the collection of data on the patient's vital signs and room temperature. The second app is used to record the intake of meals. The apps were implemented with the lightweight MQTT protocol to send the data to the IoT messaging system. The first mobile app is capable of collecting data from the elderly person's wearable subsystem via Bluetooth and collects room temperature data through one of its integrated sensors. Additionally, the logic of a lightweight alert manager was added to produce alert messages for paramedics. The second mobile app enables the housekeeper to record the intake of meals. This app can read the QR code of the foods found in the GUI of the diet preparation services and the QR code of the identification of the elderly person on a wristwatch. Figure 11 shows some examples of the QR codes sent by the mobile app and received by the EMQ cluster.

Figure 11. (a) Messages from food intake monitoring and sent to the system. (b) Example of QR codes related to the elderly person's identification. (c) Example of QR codes related to food identification. (d) Smartphone interface for capturing food intake.

Finally, each paramedic uses a smartphone that contains an app to collect medical alerts from the system. This app uses the lightweight MQTT protocol to collect alert messages from the EMQ cluster of the IoT messaging system.

5.9. Technological Resources for Software Construction and Version Control

For the development of the system software, the authors used mainly: a MacBook Pro with macOS Sierra version 10.12.3 (2 GHz Intel Core i5, RAM 8G), IntelliJ IDE 2019.1.3, Android Studio version 2.3.1, JRE: 1.8.0_112-release-408-b6 x86_64, JVM: Open JDK 64-Bit Server VM, Play Framework 2.4.8, Scala programming language version 2.11.8, HTML5, CCS3, Ajax, JSON, and JavaScript.

In the development machines, the tests were performed using Docker container images of the system components, for which Docker Machine 0.13.0 and the Docker Hub images repository were used. For the software version control, Git version 2.19.1 and GitLab version 12.1 were used. GitLab was used as a SaaS with the minimum free features, which allowed the control of distributed versions and the use of the CI/CD pipeline to build, test, and deploy the software in the preproduction environment.

5.10. Security

Data security in the processes of data transfer and storage is essential in distributed processes, which are related to the selected protocols and technologies. First, the EMQ broker supports authenticating MQTT clients with client ID, username/password, IP address, and even HTTP cookies [57]. Confluent also provides security through transport layer security (TLS) or Kerberos authentication, encryption of network traffic via TLS, and authorization via access control lists (ACLs) [58]. Furthermore, web application security was implemented with Silhouette, which supports several authentication methods, including OAuth, OpenID, CAS, credentials, and basic authentication [59].

Moreover, the authors consider the use of anonymization to prevent inferences in the data. Additionally, MongoDB provides security mechanisms, such as authentication, control, and encryption, to secure MongoDB deployments: role-based access control and transport layer security/secure sockets layer (TLS/SSL) [60]. Another important component of the architecture with security mechanisms is Apache Spark, which uses authentication via a shared secret in all the master/workers configurations and in the Spark applications [61].

On the other hand, the entire system is supported on the Google Cloud infrastructure, which provides a series of intrinsic security mechanisms such as physical access to its facilities through biometric identification [62]. For applications, there are guarantees of a secure boot stack, machine identity, service identity, service integration, service isolation, denial of service (DoS) protection, encryption of interservice communication, and intrusion detection. K8s also provides several possibilities to configure the security of nodes, containers, and pods [63].

Due to the goals of this work, only some of the K8s security options were used. Access to K8s was achieved through a Google account. Therefore, an authentication and authorization scheme for using the K8s cluster API was required. The granularity of access to K8s resources was achieved through security policies applied to users through RBAC's ClusterRoles and ClusterRoleBindings to provide access to cluster namespaces. An advantage of using K8s clusters is that the K8s nodes and the set of deployed containers represent a configurable communication network. A K8s cluster inherently provides IP filtering rules, routing tables, and firewall rules on each node. In addition, it is possible to configure additional firewall rules for the ports of system components.

The information security of the services discussed in this paper is a critical issue that must be considered for data during movement or at rest. Although we have pointed out some security mechanisms, a full study of security mechanisms is beyond the scope of this article. Some challenges that must be addressed to increase the security of the data of e-health systems are indicated below.

The collection of sensitive medical data from homes poses complex security and privacy challenges due to the open nature of patient data that is susceptible to eavesdropping. In [64], some issues related to the privacy and security of WBANs are mentioned, such as accountability, which is related to the need for a person who possesses patient information to have the responsibility of safeguarding such information. Moreover, there are several concerns about the data collected by crowdsensing applications since users' personal devices could be connected to insecure access points or be contaminated with malicious code, which opens up several challenges regarding detecting the veracity of the data collected [65]. On the other hand, in many countries, the treatment of medical data must follow strict medical regulations, such as the Health Insurance Portability and Accountability Act (HIPAA) in the United States. In recent years, there has been great interest from cloud computing platform providers such as Google and Amazon in offering HIPAA-compliant services for hosting sensitive data [66,67]. However, the implementation of HIPAA-compliant systems that, for example, support a failure in the infrastructure of the cloud platform, together with the lack of control by the owners over these infrastructures, or the lack of direct interpretation of the property of the data and the use of this information, are some of the challenges that still have to be faced [68].

6. Experiment and Result

The main objective of the experiments was to carry out a quantitative evaluation of the performance of the relevant components of the system. We performed a set of tests and an experimental analysis to meet the system requirements in scenarios compatible with a public cloud. Apache JMeter v3.3 [69,70] was used as a distributed testing tool to simulate a large number of concurrent users with artificial data. The results were established as estimates for the improvement and refinement of the architecture. They were also considered as a guide for estimating progressive changes to the system and managing workloads. The characteristics of the JMeter-K8s cluster and cluster nodes are shown in Appendix C.

The scenarios of the experiments were prepared for the preproduction test environment and were configured on K8s clusters: the e-Health-System-K8s cluster (entire e-health system) and JMeter-K8s cluster (Apache JMeter in a distributed testing mode). The tests were run on artificial data to simulate user requests for microservices and sensor data from the homes of elderly people. Additionally, the average utilization of all vCPUs in a node pool was set at 80% as the limit value for the safeguarding of vCPU use. The scenarios of the experiments were divided as follows: core services and specialized services, monitoring service, and emergency service.

6.1. Data

For testing, we used artificial data that were related to the attributes of the entities defined in the data model. These data were divided into two groups. The first group was created manually and included data such as the patient identification, the name of a patient, home address, and other attributes of medical interest such as the date of birth of a patient. These attributes have static behavior. Although they are important in the system, they are not of crucial use in testing the healthcare environment. In our research, we used this type of data to test some system microservices, such as entering a new patient's data into the system. The second group of attributes has dynamic behavior, and they are related to the patient's medical attributes, such as body temperature or blood pressure. A data generator can use some probability distribution to create values from this data type.

To perform tests and validations of some system components, we can use values obtained through the generation of artificial data. For example, to test and validate the monitoring of the vital sign data of an elderly person as close as possible to the real world using values with a valid format and range [71–73]. Although data simulation techniques were applied in this paper, the vital sign values could look unrealistic. However, we propose that the artificial data used are valid for the experiments presented in this paper.

The tests carried out were used to technically evaluate the performance and response times of system components and not the accuracy of the vital sign data used. The physiological data of the patient were modeled by Gaussian probability density functions. The Gaussian probability density function allows the generation of values around a mean and a specific value for the standard deviation [74].

Furthermore, the Ptolemy II simulator was used as an open-source software framework for modeling and simulation [75]. The simulator of the vital signs of a patient was built on a MacBook Pro. Figure 12 shows the model of the data generator that simulated the signal of vital signs. Figure 13 shows the patient monitor model that simulated the real-time behavior by simulates the amount of time given by the sampling period (in seconds) before producing an output in JSON format. Additionally, the simulator of vital signs uses the actual incoming serial port of the MacBook Pro to send simulation data to a real smartphone that acts as a gateway. For additional details on the JSON message structure of the patient's vital signs data, see Appendix G.

Figure 12. (a) The vital sign simulation model uses Gaussian distributions to generate artificial data. Some controls for the parameterization of the simulator are available to simulate certain health conditions. (b) The patient monitor model simulates the use of sensors that capture vital sign data. The sampling phases and the structuring of the messages in JSON format are included in the model.

Figure 13. Simulator of the sensing process of the patient's vital sign data. The simulation data are sent to a real smartphone via Bluetooth.

It is necessary to point out that a more exact simulation of the behavior of the patient and the WBAN is outside the limits of our work. Therefore, we have modeled the simulator considering only the sampling stages, integration of the sampled data, and the sending of the detected data from the MAC to the patient's smartphone. Despite the technological advances in medical sensors, microelectronics, and low-power miniaturization and communications, there are some open issues around WBANs that must be taken into consideration when designing and building this type of network. One of these challenges is the limitation of memory, processing, and power in the nodes of a WBAN [76].

6.2. Core Services and Specialized Services

The objective of these experiments was to analyze the performance and scalability of core services and specialized services through various workload tests. For this, we experimented with individual microservices that manage the elderly people table in the MongoDB database through the following operations: saving a document, the individual or group reading of documents, the updating of a document, and the deletion of a document. Flexible access to microservices was implemented through K8s Ingress's single IP address. Figure 14 shows the experimental scenario with the Apache JMeter test plans as XML files that include the configuration parameters for accessing the microservices. A K8s cluster was used to deploy Apache JMeter in distributed mode on which these test plans were executed. The execution of the first test plan called "Plan-microserviceforsave-1m3s.jmx" is essential since it creates records on the database.

The identifiers of the documents created on MongoDB were used to create CSV files that were necessary for the execution of the test plans for the other microservices. Apache JMeter used the RESTful API of system services to call microservices. An example of the data size of the input and output of each operation can be found in Appendix H.

Figure 14. The simulation environment for microservice testing.

To study the performance and scaling of the services, the test plans were executed independently, simulating the requests of n users during t ramp-up periods. In each of the cases, the e-Health-System-K8s-cluster nodepool-1 was configured so that the application scales from one to two instances of pods of the services and obtains the performance based on the number of requests that arrive to be processed. Since the end users are the ones who will use the operations through a graphical interface of the associated microservice, it was necessary to establish a limit as the response timeout value. The approximate value as the response timeout for web applications is estimated to be 10 s, and if this value is exceeded, end users may feel frustrated and are likely to abandon the operation [77]. In this way, in the experiments, the response timeout and the connect timeout of the requests simulated with JMeter were configured as 10 and 5 s, respectively.

Table 3 shows a summary of the tests carried out, where the configuration parameters of the JMeter test plans are indicated, as well as the results obtained when one and two pods are active. The metrics used for performance and scalability analysis were throughput and response time. We performed several tests for each operation to find the maximum throughput that included the first failed requests. The workloads were simulated through a certain number of threads (virtual concurrent users) executed with a ramp-up period of 180 s. In the specific case of the “Save” operation, Table 3 shows the information of the tests with which it was possible to reach an approximate maximum limit of the workloads with 0% requests with errors (Test 1 and Test 3). Additionally, Table 3 also provides the information of the tests with the approximate workloads where some failed requests appeared (Test 2, Test 4). Failed requests occurred because the response timeout and connect timeout limits were exceeded, and they had no direct relationship with the service capacity of the database or the network bandwidth.

Table 3. Information on the experiments carried out with the system's microservices.

Exper. ID	Operation of Microservices	Number of Pods	Apache JMeter							
			Numbers of Threads (Users)	Ramp-Up Period (s)	Duration of Experiment (hh:mm:ss)	Avg. Throughput (Request/s)	Avg. Response Time (ms)	Min. Response Time (ms)	Max. Response Time (ms)	Percentage of Requests with Errors (%)
1	save	1	48,000	180	00:03:09	254.43	253	3	2036	0.00
2	save	1	51,000	180	00:03:11	266.73	4043	4	13,059	5.38
3	save	2	96,000	180	00:04:42	341.05	625	2	4761	0.00
4	save	2	99,000	180	00:04:52	339.30	1001	2	11,184	0.20
5	read	1	69,000	180	00:03:12	360.48	162	3	1706	0.00
6	read	1	72,000	180	00:03:19	362.36	1145	2	6898	0.28
7	read	2	150,000	180	00:05:14	478.01	533	2	7714	0.00
8	read	2	153,000	180	00:06:00	425.78	557	2	15,454	0.00004
9	list	1	51,000	180	00:03:06	274.75	1513	7	3471	0.00
10	list	1	54,000	180	00:03:09	285.56	4099	6	10,502	2.87
11	list	2	99,000	180	00:04:13	391.57	972	4	6582	0.00
12	list	2	102,000	180	00:04:24	386.23	1302	4	11,115	0.35
13	update	1	51,000	180	00:03:09	269.37	397	3	3185	0.00
14	update	1	52,500	180	00:03:13	272.96	1769	3	10,022	0.17
15	update	2	105,000	180	00:04:42	372.17	312	2	3820	0.00
16	update	2	108,000	180	00:04:46	378.02	747	2	8798	0.07
17	delete	1	78,000	180	00:03:29	373.23	152	2	3550	0.00
18	delete	1	81,000	180	00:03:30	385.41	1104	2	6587	0.05
19	delete	2	144,000	180	00:04:51	494.31	86	2	7606	0.00
20	delete	2	147,000	180	00:05:52	418.19	235	2	17,388	0.07

First, when analyzing the results of the tests with 0% requests with errors, we observed that the “Save” operation and the “Update” operation reached a lower average throughput than the rest of the operations. The “Save” and “Update” operations require more processing time by requiring transformations to the database. Second, we observed that the “Delete” operation and the “Read” operation achieved higher average throughput than the rest of the operations. The “Delete” and “Read” operations are simpler actions for the database engine. Finally, the “List” operation that provided data for 20 documents has a lower average throughput than the “Read” operation that only delivers one document.

Regarding the scalability of the “Save” operation, Table 3 shows that in Test 3 (two pods), the system was able to serve 48,000 more requests than in Test 1 (one pod). The test results indicate that the average throughputs with one and two pods were 254.43 requests/s and 341.05 requests/s, respectively. Thus, the average throughput of the service increased by 34.0% ($(341.05 - 254.43) / 254.43$). The spreadsheet where we collected the parameters and results used in the preparation of tables and graphs for the “Save” operation is available in the Supplementary Materials (Spreadsheet S1).

Figure 15 shows the variations in the average response time and average throughput as a function of the number of requests for each operation when we use one and two pods. The data used in the creation of these graphs correspond to those tests with 0% requests with errors from Table 3. The horizontal scaling of the microservices increased the average throughput of the operations. Figure 15a shows the behavior of the “Save” operation when working with one pod and two pods. When one pod is running, the average throughput increases to a value close to 259 requests/s. Then, the average throughput remains roughly stable until ending with an average throughput of 254.43 requests/s and an average response of 253 ms.

When two pods are running, the average throughput increases as virtual users increase based on a 180 s ramp-up and stabilizes at approximately 495 requests/s. Then, once the first 3 min of the test are finished, JMeter does not inject any more requests. Finally, the service responds more quickly until it processes the last requests, reaching an average throughput of 341.05 requests/s.

Figure 15. Scalability based on the increment of 1 and 2 pods of core services on a K8s cluster with different types of operations: (a) Saving individual records. (b) Reading individual records. (c) Reading multiple records. (d) Individual update of records. (e) Individual deletion of records.

6.3. Monitoring Service

In the experiments with the monitoring service, we analyzed the performance of message broker clusters (EMQ cluster and Kafka cluster) as they are critical components of the messaging subsystem. The performance and scalability analysis of each of the message broker clusters was carried out in isolation from other system components. Through various workload tests, we tried to establish the data flow they can support and their differences. The metrics considered were the average throughput and the average response time.

6.3.1. EMQ Cluster Services

For the performance analysis of the message publishing service of the EMQ cluster, the JMeter-K8s cluster was used to simulate the request to publish messages from the patients' smartphones. Additionally, a preliminary scalability analysis of the EMQ cluster was performed using one and three brokers. The tests were not intended to reach the maximum service throughput offered by the broker cluster. The Apache JMeter plan for testing these scenarios is an XML file whose configuration parameters are shown in Figure 16. The test plan was mainly configured with the IP address of the load balancer used to access the cluster, the topic "vitalSignsTopic", and the value of the quality of service (QoS). Furthermore, the messages sent to the EMQ cluster had a structure and size expected in the real world (Section 6.1).

Figure 16. The simulation environment for the EMQ cluster and Kafka cluster tests.

An important aspect of the MQTT protocol is the possibility of setting a QoS value (0, 1, or 2) for publishers and subscribers, which affects the performance of the EMQ cluster service. In the experiments, we used a QoS equal to 2 to guarantee the delivery of messages to the EMQ cluster. Table 4 shows the results of the experiments with various workloads of publishing requests to the EMQ cluster with one and three brokers (see Supplementary Materials Spreadsheet S2).

Table 4. Information on the experiments with the EMQ cluster using 1 and 3 brokers.

Number of Test	Numbers of MQTT Brokers	Apache JMeter							Percentage of Requests with Errors (%)
		Numbers of Threads (#Users)	Ramp-Up Period (s)	Duration of Experiment (hh:mm:ss)	Avg. Throughput (Re-quest/s)	Avg. Response Time (ms)	Min. Response Time (ms)	Max. Response Time (ms)	
1	1	6000	1	00:00:09	716.33	1532	3	7226	0.00
2	1	9000	1	00:00:10	908.54	2599	3	7601	0.00
3	1	18,000	1	00:00:20	924.45	6801	8	16,742	0.00
4	1	27,000	1	00:00:39	695.52	12,858	828	32,520	0.00
5	1	18,000	60	00:01:06	275.66	60	1	2879	0.00
6	1	27,000	60	00:01:06	414.11	19	1	1479	0.00
7	1	36,000	60	00:01:08	533.16	12	1	1051	0.00
8	1	18,000	120	00:02:04	145.46	9	2	1150	0.00
9	1	27,000	120	00:02:04	218.97	6	1	356	0.00
10	1	36,000	120	00:02:05	288.01	6	2	739	0.00
11	1	18,000	180	00:03:04	97.93	6	2	578	0.00
12	1	27,000	180	00:03:04	147.31	5	2	410	0.00
13	1	36,000	180	00:03:10	190.43	19	1	2301	0.00
14	3	4500	1	00:00:09	518.67	1090	3	3773	0.00
15	3	6000	1	00:00:08	751.41	1426	3	5129	0.00
16	3	9000	1	00:00:10	968.89	3760	315	7986	0.00
17	3	18,000	1	00:00:18	1012.09	5697	6	14,104	0.00
18	3	27,000	1	00:00:23	1206.33	10,840	22	21,985	0.00
19	3	36,000	1	00:00:32	1156.55	12,488	8	26,808	0.00
20	3	18,000	60	00:01:03	286.25	10	2	965	0.00
21	3	27,000	60	00:01:04	423.66	14	2	942	0.00
22	3	36,000	60	00:01:10	515.43	646	2	16,502	0.00
23	3	18,000	120	00:02:02	147.50	5	2	262	0.00
24	3	27,000	120	00:02:03	219.50	5	2	284	0.00
25	3	36,000	120	00:02:09	279.38	30	2	1521	0.00
26	3	18,000	180	00:03:01	99.35	5	1	246	0.00
27	3	27,000	180	00:03:05	145.95	10	1	1798	0.00
28	3	36,000	180	00:03:05	194.39	7	2	449	0.00

From the JMeter-K8s cluster, messages were sent to the EMQ cluster using certain ramp-ups to analyze the functional behavior of the EMQ cluster for various periods of operation. For example, in tests 26, 27, and 28, the EMQ cluster flexibly handled the increase in workload variation during testing. The average throughputs in these tests were 99.35 requests/s, 145.95 requests/s, and 194.39 requests/s with average response time values of less than 15 ms. The highest service capacity was obtained with three brokers, which could be established by comparing tests 1, 2, 3, and 4 with tests 15, 16, 17, and 18.

Our final objective for the e-health system was to use an EMQ cluster with three brokers, so we analyzed in more detail the experiments with this type of cluster. Figure 17 shows the variation of the average throughput for tests 19, 22, 25, and 28 in which the same volumes of service requests were used. However, different ramp-up values were used to observe the effects of pressure reduction on the EMQ cluster. The results show that the EMQ cluster was able to respond to some tests with a response time of more than 2 s, as in the case of test 19 (Avg. RT = 12,488 ms, Max. RT = 26,808 ms). Average response times of more than 2 s are not desirable for some types of systems that require average response times between 1000 ms and 2000 ms. For the messaging subsystem of the health system, it is necessary to have a maximum average throughput and a minimum average response time to maintain a dynamic data flow.

Based on the conditions of the experiments with the three-broker EMQ cluster, Test 14 (Avg. RT = 1090 ms, Max. RT = 3773 ms) provides an approximation of the average throughput and the adequate average response time for the acceptable operation of the system. In this paper, we set 4500 requests/s (Test 14) as the reference value for the number of message publishing requests that can be sent to the EMQ cluster. The average throughput and the average response time with which the EMQ cluster handles 4500 requests/s are 518.67 requests/s and 1090 ms, respectively.

Figure 17. Average throughput of a 3-broker EMQ cluster subjected to a workload of 36,000 requests for 1 s (Test 19), 60 s (Test 22), 120 s (Test 25), and 180 s (Test 28).

It must be taken into consideration that the processing capacity (vCPU and memory) of each component of the system in the e-Health-System-K8s-cluster nodepool-2 is affected by the rest of the components within this same node pool. Finally, horizontal or vertical scaling techniques can be used to improve the processing capabilities of the system. Some scaling techniques are the increase in brokers, the addition of K8s nodes, and the use of K8s cluster node pools with greater memory and vCPU capacities.

6.3.2. Kafka Cluster Services

For the analysis of the performance of the message production service of the Kafka cluster, the JMeter-K8s cluster was used to simulate the transfer of messages to the Kafka cluster. Additionally, a preliminary analysis of the scalability of the Kafka cluster was carried out using one and three brokers. The search for the maximum throughput of the Kafka cluster is beyond the scope of this paper.

The Apache JMeter plan for testing these scenarios is an XML file whose configuration parameters are shown in Figure 16. The test plan was mainly configured with the IP address of the load balancers used to access the cluster, the topic “vitalSignsTopic”, and the value of QoS. Furthermore, the messages sent to the Kafka cluster have a structure and size expected in the real world as we established in Section 6.1.

A Kafka client can manage the Kafka cluster’s quality of service through the “acks” value (0, 1, −1). This value of “acks” (0, 1, and −1) affects the level of degradation of the cluster’s performance (none, medium, and strong) depending on the type of recognition of the producers’ message. In the experiments, the default value of acks was used, which is 1, thus guaranteeing the distribution of messages and minimally affecting the performance of the Kafka cluster messaging service.

Before the execution of the tests, it was necessary to define the topic “vitalSignsTopic” in the Kafka cluster to provide redundancy and scalability. Thus, for the tests with one broker, we created the topic “vitalSignsTopic” with three partitions and one replica. Additionally, for the tests with three brokers, we created the topic “vitalSignsTopic” with three partitions and three replicas. Table 5 shows the results of the experiments with various workloads of message production requests to the Kafka cluster with one and three brokers (see Supplementary Materials Spreadsheet S3). From the JMeter-K8s cluster, messages were sent to the Kafka cluster using certain ramp-ups to analyze the functional behavior

of the Kafka cluster for various periods of operation. For example, in tests 22, 23, and 24, the Kafka cluster flexibly handled the increased workload variation during testing. The average throughputs in these tests were 99.36 requests/s, 148.73 requests/s, and 197.46 requests/s with average response times values of less than 210 ms. The highest service capacity was obtained with three brokers, which could be established by comparing Tests 1, 2, and 3 with Tests 13, 14, and 15.

Table 5. Information on the experiments with the Kafka cluster using 1 and 3 brokers.

Number of Test	# of Zookeepers, # of Kafka Brokers	Apache JMeter							Percentage of Requests with Errors (%)
		Numbers of Threads (#Users)	Ramp-Up Period (s)	Duration of Experiment (hh:mm:ss)	Avg. Throughput (Request/s)	Avg. Response Time (ms)	Min. Response Time (ms)	Max. Response Time (ms)	
1	(1, 1)	18,000	1	00:00:24	932.45	491	2	3814	0.00
2	(1, 1)	27,000	1	00:00:45	749.27	901	2	9192	0.00
3	(1, 1)	36,000	1	00:01:48	379.76	8332	1	60,626	0.00
4	(1, 1)	18,000	60	00:01:08	285.99	205	3	628	0.00
5	(1, 1)	27,000	60	00:01:16	401.88	221	1	1654	0.00
6	(1, 1)	36,000	60	00:01:40	412.22	439	2	8958	0.00
7	(1, 1)	18,000	120	00:02:07	147.48	204	12	529	0.00
8	(1, 1)	27,000	120	00:02:11	218.88	207	23	1040	0.00
9	(1, 1)	36,000	120	00:02:17	286.07	209	1	1295	0.00
10	(1, 1)	18,000	180	00:03:06	98.98	203	4	591	0.00
11	(1, 1)	27,000	180	00:03:14	145.02	205	9	755	0.00
12	(1, 1)	36,000	180	00:03:27	185.79	213	2	4767	0.00
13	(3, 3)	18,000	1	00:00:23	999.78	669	1	5965	0.00
14	(3, 3)	27,000	1	00:00:38	871.00	1222	2	10,783	0.00
15	(3, 3)	36,000	1	00:00:52	878.50	1624	1	13,578	0.00
16	(3, 3)	18,000	60	00:01:07	291.83	207	84	670	0.00
17	(3, 3)	27,000	60	00:01:11	433.79	211	1	717	0.00
18	(3, 3)	36,000	60	00:01:17	564.87	207	1	1205	0.00
19	(3, 3)	18,000	120	00:02:07	148.46	204	6	453	0.00
20	(3, 3)	27,000	120	00:02:12	221.19	207	3	538	0.00
21	(3, 3)	36,000	120	00:02:18	292.02	207	2	698	0.00
22	(3, 3)	18,000	180	00:03:07	99.36	203	6	468	0.00
23	(3, 3)	27,000	180	00:03:11	148.73	205	3	487	0.00
24	(3, 3)	36,000	180	00:03:16	197.46	206	9	598	0.00

Our final objective for the e-health system was to use a Kafka cluster with three brokers, so we analyzed in more detail the experiments with this type of cluster. Figure 18 shows the variation in the average throughput for Tests 15, 18, 21, and 24 in which the same volumes of service requests were used. However, different ramp-up values were used to observe the effects of pressure reduction on the Kafka cluster. The results show that the Kafka cluster was able to respond to some tests with a response time of more than 2 s, as in the case of Test 15 (Avg. RT = 1624 ms, Max. RT = 13,578 ms), which is not valid for systems that require average times between 1000 ms and 2000 ms.

Based on the conditions of the experiments with the three-broker Kafka cluster, Test 13 (Avg. RT = 669 ms, Max. RT = 5965 ms) provides an approximation of the average throughput and the adequate average response time for the acceptable operation of the system. In this paper, we established 18,000 requests/s (Test 13) as the reference value of the number of requests for production messages that can be sent to the Kafka cluster. The average throughput and the average response time with which the Kafka cluster handles 18,000 requests/s are 999.78 requests/s and 669 ms, respectively. The results obtained indicate that the Kafka cluster can support the flow of data that can come from the EMQ cluster. Additionally, the Kafka cluster provides a mechanism to handle the back pressure exerted by the EMQ cluster. Kafka's back pressure mechanism is based on the use of reliable storage for the persistence of messages. Messages can be stored in the Kafka cluster until the consumer can retrieve them.

Figure 18. Average throughput of a 3-broker Kafka cluster subjected to a workload of 36,000 requests for 1 s (Test 15), 60 s (Test 18), 120 s (Test 21), and 180 s (Test 24).

It must be taken into consideration that all the components of the messaging subsystem compete for the resources of the node pool. The throughput of one component affects the others within the node pool. Thus, horizontal or vertical scaling techniques may be required. In general, the performance analysis of system components is important to define the capabilities of a K8s cluster or K8s cluster node pools to meet the system requirements.

6.3.3. Dashboard for Monitoring Vital Signs

In the monitoring process, the vital signs of the patient are transferred from the elderly person's smartphone to the messaging subsystem and Kafka Connect collects the data from Kafka to record them in the database. The interface collects the data in a set of variables and points out the level of these variables. Figure 19 shows the instant when the elderly person has "prehypertension", a temperature level classified as "fever", and a heart rate considered as "normal".

6.4. Emergency Service

The emergency service sends messages to the smartphones of paramedics when the health indicators of elderly people are in an abnormal state. The key metric obtained in the experiments is the average response time of the system to provide an emergency alert. In parallel, we maintain several workloads on the messaging subsystem to verify the effects on the delivery times of the alert messages. In this paper, two possible locations for alert managers were analyzed:

1. Alert manager 1: Can be placed on the patient's smartphone as an additional lightweight module within the app that collects vital sign data;
2. Alert manager 2: Can be placed as a Spark app in the cloud computing environment. It is a streaming service that continuously and dynamically monitors the vital signs in near real time.

These configurations were chosen to understand and explore two issues: (a) the impact of the alert manager location on its development and functionality and (b) whether a near-real-time rule engine is feasible in an IoT environment. The tests performed in this section used mainly the messaging subsystem in a controlled environment. Figure 20 shows the relevant components used in the tests:

1. A simulator of multiple patient vital sign messages. It is a JMeter–K8s cluster that publishes messages to the EMQ cluster to create various types of workloads on the messaging subsystem. The data generated by the JMeter–K8s cluster only include data from patients in normal health;
2. A simulator of vital signs of a patient in critical condition. It is a Ptolemy simulator supported by a MacBook that sends the simulation data to the patient's real smartphone via Bluetooth. The simulator is capable of generating messages with normal conditions of the patient's vital signs and messages with abnormal conditions;
3. A real smartphone with an app that collects data from the simulator of vital signs with the topic "vitalSingsTopic/s1". This smartphone forwards the patient data to the EMQ cluster with the same topic "vitalSingsTopic/s1". Additionally, in case the alert manager 1 installed on this smartphone detects abnormal conditions in the patient's vital signs, it generates emergency messages to the EMQ cluster with the topic "emergencyTopic";
4. The messaging subsystem in the cloud computing environment consists of the EMQ cluster, MQTT–Kafka bridge, and the Kafka cluster;
5. A Spark application that was implemented as alert manager 2. It is a rules engine that uses near-real-time analytics processing based on Apache Spark. Additionally, in case this Spark application detects abnormal conditions in the patient's vital signs, this application sends emergency messages to the EMQ cluster with the topic "emergencyTopic";
6. A real smartphone belonging to medical personnel with an app that collects the emergency message from the EMQ cluster. The mobile app is an MQTT client subscribed to the topic "emergencyTopic". The routes for sending the emergency message to the paramedics due to the two possible locations of the alert manager are a–b and c–d (see Figure 20).

Figure 19. The dashboard of the patient monitoring service.

Figure 20. Test environment to measure the response time to generate an emergency alert. This environment consists of (1) a simulator of multiple patient vital sign messages, (2) a simulator of vital signs of a patient in critical condition, (3) a real smartphone with an app that collects data from the simulator of vital signs y contains the alert manager 1, (4) the messaging subsystem in the cloud computing environment consists of the EMQ cluster, MQTT-Kafka bridge, and the Kafka cluster, (5) a Spark application that was implemented as alert manager 2, and (6) a real smartphone belonging to medical personnel with an app that collects the emergency message from the EMQ cluster. Additionally, the figure shows the two analyzed routes through which emergency messages could be sent to paramedics: a–b and c–d.

The tests were initially conducted with a single MQTT-Kafka bridge until scaled enough to study behavior with multiple MQTT-Kafka bridges. As the MQTT-Kafka bridge was set up as a custom application to route data from the EMQ cluster to the Kafka cluster, the scalability of the bridge generates duplicate messages. Therefore, the scalability of the bridge was accompanied by a division and distribution of the topic “vitalSignsTopic”. The number of subtopics used is a function of the number of MQTT-Kafka bridges used. The JMeter test plan was set up to accommodate multiple topic publishing. On the other hand, this means that the elderly population should be divided according to the number of subtopics.

The tests lasted 180 s to obtain a minimum controlled time window. The size of this time window allowed for covering the stability of the data flow in the messaging subsystem at the beginning of each test and covering the minimum expected response time of the system for the alert notifications generated. Additionally, the tests took into consideration two simulation conditions (“A” and “B”). The test carried out with simulation condition “A” was a reference test to measure the response time of the system to produce an alert, sending only emergency messages to the messaging subsystem. In this test, the Ptolemy simulator sent to the messaging system 10 messages (approximately at the rate of one message every 4 s) with the vital signs of one patient in a critical health condition. The

tests carried out with simulation condition “B” consider the most realistic use of the system components and their impact on the response time of the system to notify an alert. In these tests, the K8s-JMeter cluster simulates the data flow of vital signs (normal health status) of N patients. In parallel, the Ptolemy simulator sent to the messaging system 10 messages (approximately at the rate of one message every 4 s) with the critical health status. The data with the abnormal condition were sent 90 s after the start of the experiment (approximately 1/2 of the total experiment time). Table 6 shows the parameters for the execution of the JMeter test plan and the scalability conditions of the MQTT-Kafka bridge. Furthermore, Table 6 gives the results obtained by Apache JMeter, the number of messages that arrive at the Kafka cluster, and the measurement of the system response time to notify an emergency. The spreadsheet where we collected the parameters and simulation results used in the preparation of tables and graphs is available in the Supplementary Materials (Spreadsheet S4). The following events considered for measuring the response time of the system to notify an emergency alert were:

1. The time when the patient monitor simulator performs data sampling;
2. The moment when the alert notification message (generated in alert manager 1) reaches the smartphone of the paramedic;
3. The moment when the alert notification message (generated in alert manager 2) reaches the smartphone of the paramedic.

Figure 21 shows a fragment of the logs supported by smartphones and related to one of the messages sent from the Ptolemy simulator. The response time of the alert notification service was measured using the record of the timestamps collected in each of the smartphones. From Figure 21 we were able to extract:

1. The message with the patient’s vital signs and the moment in which the data were sampled. The timestamp is “Wednesday, 25 November 2020, 20:41:55.607000000” (see Figure 21a);
2. The alert message generated by alert manager 1 and received on the paramedic’s smartphone. The arrival timestamp is “Wednesday, 25 November 2020, 20:41:58.569000000” (see Figure 21b);
3. The alert message generated by alert manager 2 and received on the paramedic’s smartphone. The arrival timestamp is “Wednesday, 25 November 2020, 20:41:59.593000000” (see Figure 21c).

Table 6. Information on the test environment and results of the measurement on the average response time to notify an emergency.

Number of Test (#)	Simulation Condition	Initial Time of the Simulation of Emergency States (s)	Apache JMeter		Bridge			Results							
			Total Number of Threads (or Users)	Ramp-Up Period (s)	Scaling of the MQTT-Kafka Bridge (# of Bridges)	Distribution of the Topic “vitalSignsTopic”	Total Expected Messages for Each Instance of the Bridge	Apache JMeter		Number of Messages Reaching Kafka (#)	System Response Time to Communicate an Emergency				
								Duration of Experiment (hh:mm:ss)	Throughput of MQTT Cluster (Users/s)		Avg. Response Time from MQTT Cluster (ms)	Option 1		Option 2	
												Average (s)	Standard Deviation (s)	Average (s)	Standard Deviation (s)
1	A	90	N/A	N/A	1	vitalSignsTopic/s1	N/A	N/A	N/A	10	1.61	0.218	2.31	0.382	
2	B	90	3000	180	1	vitalSignsTopic/s1	3000	00:03:01	16.6/s	377	3010	1.28	0.193	1.98	0.525
3		90	9000	180	1	vitalSignsTopic/s1	9000	00:03:01	49.7/s	191	9010	1.48	0.194	2.11	0.308
4		90	21,000	180	1	vitalSignsTopic/s1	21,000	00:03:03	115.0/s	377	21,010	3.33	0.294	72.48	7.432
5		90	24,000	180	1	vitalSignsTopic/s1	24,000	00:03:05	129.8/s	1100	24,010	3.25	0.221	97.19	44.178
6		90	25,500	180	1	vitalSignsTopic/s1	25,500	00:03:05	137.5/s	857	11,722	3.27	0.2396	The bridge ran out of memory ¹	
7		90	12,000	180	2	vitalSignsTopic/s1 vitalSignsTopic/s2	6000	00:03:02	65.9/s	576	12,010	2.65	0.208	3.28	0.351
8		90	18,000	180	2	vitalSignsTopic/s1 vitalSignsTopic/s2	9000	00:03:03	98.5/s	629	18,010	2.73	0.197	3.40	0.397
9		90	33,000	180	2	vitalSignsTopic/s1 vitalSignsTopic/s2	16,500	00:03:02	181.6/s	203	33,010	3.07	0.237	11.24	2.189
10		90	35,400	180	2	vitalSignsTopic/s1 vitalSignsTopic/s2	17,700	00:03:03	193.5/s	361	26,522	3.16	0.186	26.12 ²	2.939
11		90	18,000	180	3	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3	6000	00:03:01	99.5/s	224	18,010	2.28	0.458	2.99	0.519
12		90	36,000	180	3	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3	12,000	00:03:01	198.7/s	271	36,010	2.31	0.296	2.83	0.662
13		90	42,300	180	3	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3	14,100	00:03:02	232.8/s	581	42,310	2.91	0.323	5.40	0.685
14		90	45,000	180	3	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3	15,000	00:03:02	247.2/s	228	39,074	2.73	0.185	7.57 ³	0.555
15	90	24,000	180	4	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3 vitalSignsTopic/s4	6000	00:03:02	131.6/s	585	24,010	2.30	0.503	2.97	0.773	

Table 6. Cont.

Number of Test (#)	Simulation Condition	Initial Time of the Simulation of Emergency States (s)	Apache JMeter		Bridge		Total Expected Messages for Each Instance of the Bridge	Apache JMeter			Number of Messages Reaching Kafka (#)	Results			
			Total Number of Threads (or Users)	Ramp-Up Period (s)	Scaling of the MQTT-Kafka Bridge (# of Bridges)	Distribution of the Topic “vitalSignsTopic”		Duration of Experiment (hh:mm:ss)	Throughput of MQTT Cluster (Users/s)	Avg. Response Time from MQTT Cluster (ms)		System Response Time to Communicate an Emergency			
												Option 1		Option 2	
												Average (s)	Standard Deviation (s)	Average (s)	Standard Deviation (s)
16		90	44,400	180	4	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3 vitalSignsTopic/s4	11,100	00:03:02	243.6/s	537	44,410	2.09	0.241	2.67	0.635
17		90	54,000	180	4	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3 vitalSignsTopic/s4	13,500	00:03:04	293.4/s	1208	54,010	2.14	0.418	2.81	0.634
18		90	60,000	180	4	vitalSignsTopic/s1 vitalSignsTopic/s2 vitalSignsTopic/s3 vitalSignsTopic/s4	15,000	00:03:02	329.3/s	494	40,068	2.22	0.234717	22.13 ⁴	1.302

¹ Messages did not arrive in the Spark App. ² The 2nd bridge ran out of memory. ³ The 2nd bridge ran out of memory. ⁴ The 2nd and 3rd bridges ran out of memory.

Figure 21. Smartphone log extracts show: (a) the vital sign data collected by the patient's smartphone at home. (b) The emergency message generated by alert manager 1 and received by the paramedic's smartphone (c) The emergency message generated by alert manager 2 and received by the paramedic's smartphone.

For this sample, the response time of the system to produce the emergency alert could be calculated as follows:

- System response time to produce the alert using alert manager 1 = 20:41:58.569000000 – 20:41:55.607000000 = 2.962 s;
- System response time to produce the alert using alert manager 2 = 20:41:59.593000000 – 20:41:55.607000000 = 3.986 s.

The average system response time to communicate an emergency in each test was evaluated as the average of the response times to produce the alert of the 10 samples generated in the Ptolemy simulator. The results in Table 6 show the impact of various workloads and the increase in the number of MQTT-Kafka bridges. For example, Test 15 indicates that the workload could be supported using four bridges, obtaining the following results:

1. An average system response time and a standard deviation of 2.3016 s and 0.503399378 s using alert manager 1, respectively;
2. An average system response time and a standard deviation of 2.9726 s and 0.7731912519 s using alert manager 2, respectively.

The average system response time using alert manager 2 was mainly affected by the scalability of the MQTT-Kafka bridge. We observed that if we did not increase the number of MQTT-Kafka bridges when the workload on the messaging subsystem increased, the following occurred:

1. The average system response time increased (Test 10 and Test 18);
2. The messaging subsystem service completely stopped the data flow toward Kafka (Test 6) due to bridge failure;

3. The messaging subsystem service partially stopped the data flow toward Kafka due to the failure of some bridges (Tests 10, 14, and 18) with loss of messages (% lost of Test 18 = $[(60,000 - 40,068)/60,000] \times 100 = 33\%$).

Although the bridge was scaled on a single and exclusive node to avoid affecting the capacity of the other components, the bridge scaled weakly compared to, for example, the MQTT cluster. This weakness is because our MQTT–Kafka bridge was built as a custom application to route data from the EMQ cluster to the Kafka cluster. Additionally, the bridge had to contend with the limits of the resources on the VM on which it was deployed.

From Table 6, we can assert that the best option to carry out alert notifications is the use of alert manager 1. The use of alert manager 1 does not require the MQTT–Kafka bridge, and the response times of the system are shorter. However, using alert manager 2 can have other benefits. We can deploy the alert manager as a Spark app in the cloud computing infrastructure for individual or group analysis of data in near real time. Additionally, the management of the update and installation of alert manager software on the smartphones of all patients is not very flexible. However, the management of the alert manager software as a Spark app in the cloud computing infrastructure facilitates the management of the code, as it is a single point of update and use of the software.

Finally, in the future we will implement the MQTT–Kafka bridge with new design strategies to scale more flexibly. There are solutions to directly bridge MQTT and Kafka, but these solutions are only incorporated in the commercial versions of Confluent and EMQ.

7. Conclusions and Future Work

Currently, the growth in the number of elderly individuals is a concern of their families and health centers, which will increase in the coming years. One solution to address these needs is the use of e-health systems to provide healthcare services that monitor the activities of elderly people in their homes. However, currently, e-health solutions for healthcare are built with different technologies and without a common or explicit approach for the development of health services. Additionally, these solutions often have scalability, interoperability, and extensibility issues.

The uses of ICTs can help meet the requirements for the care of elderly people by building system architectures that combine technologies to enable the implementation of health applications. By exploiting the opportunities of a widely connected world, it is feasible to build pervasive, fast, and reliable services, which also contribute to the establishment of smart cities. This paper presents an approach to build cloud-based IoT reactive services in the area of e-health for elderly care at home.

In summary, the approach used the software architecture process by Rozanski and Woods to design certain architectural views and obtain the architecture description. Some architectural patterns were applied to design a system architecture whose components, deployed in a public cloud, met the requirements related to the IoT subsystem and all software services around the care of elderly people. To address some of the issues, we took into consideration providing reactive features to the system: responsive, resilient, elastic, and message driven. Furthermore, the design has the characteristics of an emerging fast data architecture with a big data subsystem to meet the needs related to the IoT subsystem and the data analytics subsystem. In addition, the software service development process followed the DDD as a foundation for the structuring of microservices. Finally, the system components were deployed on a public cloud with a CaaS model, and the deployment of microservices was made more flexible through DevOps practices with a CI/CD pipeline to obtain a dynamic workflow.

EMQ and Confluent are elements of great importance to configure an emerging fast data architecture that provides a data ingestion channel and allows the interconnectivity of some other system components. EMQ and Confluent can have high availability and can scale to achieve a system capable of ingesting, processing, and serving data in near real time. The big data subsystem was built with MongoDB and Apache Spark, where MongoDB helps store the big data and guarantees dynamic operations to users, and Apache Spark

permits near-real-time analysis. For example, an alert manager, as a Spark application, collects data from Confluent and manages emergencies with a predefined data analysis logic applied in near real time.

CaaS is a cloud service model that was used to manage containers through Kubernetes, which helped to leave behind concerns related to operating system configuration, considering only tasks such as container deployments or scalability. In this way, complex on-premise infrastructures, which are difficult to scale, can be taken to a public cloud to scale and increase their availability.

From the developer's point of view, a unified agile development environment is offered to build IoT services in the e-health area. The architecture is extensible since the use of Apache Kafka, as a distributed streaming platform complemented with Kafka Connect, helps to quickly build new data streams using existing connectors for other types of common data sources and sinks. Moreover, dynamic workflows were established that reduced the deployment time of services in the cloud computing infrastructure through the inclusion of DevOps practices with CI/CD pipelines. Thus, new subapplications implemented as microservices can be deployed flexibly to extend system services. Although batch services are beyond the scope of this paper, it is possible to introduce this type of service into a new workflow that uses Apache Spark and MongoDB to meet a specific functional requirement, such as obtaining medium-sized machine learning models.

Unlike systems whose construction is carried out on an infrastructure that is difficult to scale (for example, monolithic systems), the system presented in this article provides near-real-time data flow management, high-capacity messaging systems, and near-real-time data analytics. The authors conclude that the proposed approach helps build e-health services on a fast, scalable, high-availability, and reliable infrastructure. These characteristics play a key role in e-health systems as they add useful properties to services for the benefit of elderly people and for users who work collaboratively to provide care for elderly people, taking advantage of data flow management in near real time. With the accomplishment of these results, one can envision using this approach for building other e-health reactive services (e.g., cognitive function care services).

In future work, the deployment of the system could be carried out through a hybrid strategy that manages the local infrastructure in a private cloud combined with a public cloud. System components that require high processing capacity and high storage capacity can be maintained in a public cloud, and confidential processes essential for monitoring the health of elderly people, such as real-time reports, can be maintained in the private cloud. Additionally, we will investigate other essential innovative services for the well-being of elderly people, such as sleep care services, medication intake care services, and cognitive function care services.

Supplementary Materials: The following are available online at <https://www.mdpi.com/article/10.3390/app11115172/s1>, Spreadsheet S1: Information of test of microservices, Spreadsheet S2: Information of test of EMQ broker cluster, Spreadsheet S3: Information of test of Kafka broker cluster, Spreadsheet S4: Information of test of emergency service.

Author Contributions: The presented work is a product of the intellectual collaboration of all authors. L.J.P. is the main researcher of the work. J.S. supervised the work and contributed to manuscript organization. Both authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank the "Secretaría de Educación Superior, Ciencia, Tecnología e Innovación (SENESCYT) of the Republic of Ecuador" in this research and its progress.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Tables of data model.

Table A1. Description of the data model entities.

Entity	Description
elderlyPeople	It includes any information related to the personal data of all elderly people.
healthCarePersonnel	It includes medical staff data.
familyMemberContacts	It is the information of family members who have responsibility for the care of the elderly person.
appointments	It contains the data of appointments made for regular check-ups by the medical staff
homes	It contains a set of features related to where elderly people live.
homeSpaces	It saves a set of features related to each of the rooms in the home.
sensors	This table defines the data of all the sensors used in the home.
vitalSignsTracks	It stores the data from the process of monitoring vital signs.
roomTemperatureTracks	It stores the data from the process of monitoring the room temperature.
diets	It is the catalog of diets with a set of digitized diets. It contains basic diet plans with daily schedules per week.
patientDiets	It contains the diet information assigned to each individual with the type of diet, the schedule, and the duration.
foodProgramIngest	It stores information about diet meals prepared by the housekeeper.
dailyIntake	This table stores information about the daily intake data.
primaryPhysicalData	It maintains the physical information of the individuals, such as age, weight, height, and the PAL. These data are necessary each time a prediction of a diet is made through the EER.

Appendix B

Detail of the development view.

Figure A1 shows the essential elements of the application development process, some of which are described below:

1. Developing applications are shown as a set of subapplications that are implemented as microservices (Sub-App1 = {uServices} and Sub-App2 = {uServices});
2. There is a set of libraries necessary for the operation of microservices (such as MongoDB, Apache Spark, EMQ, Apache Kafka, and Apache Zookeeper libraries);
3. The third-party software in Docker containers (for instance, MQTT broker and Apache Kafka) can be used to create an agile development and testing environment for subapplications on the same computer. In addition, the subapplication microservice versions can be updated using a distributed version control system such as GitLab;
4. Finally, there is a platform used to support the development, integration, and versioning of subapplications. Some of the main elements of the platform are the following: simple build tool (SBT), software development kit (SDK), Scala-SDK, Play Framework support, IntelliJ IDEA with Git integration plugin, Maven integration plugin, and GitLab project plugin.

Characteristic	Value
Image type	Ubuntu
Cores	2 vCPUs
Memory	7.5 GB/Node
Architecture	amd64
Machine Type	n1-standard-2
Boot disk type	Standard persistent disk
Boot disk size (per node)	100 GB

Table A4. JMeter–K8s cluster.

Characteristic	Value
Master version	1.16.13-gke.401
Total size	4 nodes (1 JMeter master y 3 JMeter slaves)
Node zones	europa-west2-b (Node 1, Node 2, Node 3, Node 4)
Pod address range	10.48.0.0/14

Appendix D

Kubernetes patterns.

Table A5. K8s patterns and their uses in our architecture.

Kubernetes Patterns	Description
Multiple Availability Zone Design [78,79]	This pattern provides independent power, network, security, and isolation from failures in other availability zones. Availability zones within the same region have low latency network connectivity between them. It provides sufficient configuration within the scope of our system testing.
Single Container [80]	A single container per pod is the simplest and most common Kubernetes use case. Each pod is used to run a single instance of a given application. It is a basic pattern for the placement of the components of the system applications. All system components were deployed under this pattern as there are no components that must work together in a single pod.
Automated Placement [81]	It helps influence the placement of pods in the cluster to control the impact on the availability, performance, and capacity of the distributed systems. We use the following strategies: Node-Name: The simplest way to place a pod on a specific node. It was used to deploy the MQTT–Kafka bridge and scale it on a single and exclusive node. Pod-Anti-affinity: Spread the pods of a service across nodes or availability zones, e.g., to reduce correlated failures. It was applied in the deployment of the three messaging brokers of the EMQ cluster in different nodes.
Stateful Service [82,83]	The stateful service pattern provides building blocks through the Kubernetes StatefulSet resource for the management of distributed stateful applications. It provides persistent identity, networking through services resources, storage (persistent disks), and ordinality (instantiation order/position of pods). In this paper, this pattern was suitable for implementing clusters of ZooKeeper, Kafka, MongoDB, and MQTT that required unique and persistent identities.

Table A5. Cont.

Kubernetes Patterns	Description
Service Discovery [30]	<p>The Service Discovery pattern provides a stable endpoint to provide access to system services. We use the following mechanisms:</p> <p>Internal Service Discovery: It is the implicit mechanism for accessing the pods from inside the Kubernetes cluster that contains them. In this paper, the MQTT–Kafka bridge uses the internal services of the MQTT and Kafka clusters to connect to them.</p> <p>Load Balancer Service Discovery: It provides access to system components that facilitate its use via a cloud provider’s load balancer. In this work, a load balancer is used to publish the patient’s vital sign data in the EMQ cluster.</p> <p>Application Layer Service Discovery: The advantage of this mechanism is that the HTTP request contains the host and path to address multiple services under the same IP address. It is implemented through a Kubernetes Ingress. In this paper, access to microservices was carried out via K8s Ingress.</p>
Environment Variable Configuration [30]	<p>It provides mechanisms to parameterize the operation of system components. The simplest mechanism is the use of a reduced set of environment variables to store configuration data. Another strategy used is the use of Kubernetes configuration resources (ConfigMap and Secret) to provide storage and management of key–value pairs.</p>
Fixed Deployment [30]	<p>It is the way to update the system components. In this paper, the only strategy used is the one that ensures the deployment and existence of a single version of the component in the cluster. The blue–green and canary release strategies are not used because they are used in production to maintain several versions coexisting in the cluster.</p>
Elastic Scale [30]	<p>Scaling strategies help the system not break down and react to heavy loads. In this research, horizontal scaling was used by increasing pods and increasing nodes in the cluster.</p>

Appendix E

Details of the e-health subapplications and specific software deployment views.

Figure A2 shows GitLab as a complete DevOps platform and e-Health-System-K8s-cluster nodepool-1 containing the deployed microservices. In general, the GitLab platform integrates GitLab projects into a Kubernetes cluster through components such as Helm-Tiller, Ingress, Cert-Manager, and GitLab Runner. The main element for CI/CD tasks is the GitLab Runner that interacts with the GitLab repository and executes CI/CD jobs for the deployment of applications to preproduction [36]. The final automatic deployment process carried out by GitLab of the subapplications leaves as a result the Ingress Service component that provides access to the services of the subapplications.

The GitLab CI/CD pipeline for the integration and automatic deployment of system microservices was designed through a set of scripts with basic stages for building, testing, and deployment (gitlab-ci.yml file). When the developer code is pushed to the repository in GitLab, the GitLab triggers the CI/CD pipeline. The pipeline builds and stores the image in the GitLab container registry, runs the test, and invokes the GitLab Runner that automatically deploys the microservice on the Kubernetes cluster.

Figure A2. Detailed view of the deployment of the e-health subapplication microservices in the cloud from the GitLab repository.

Figure A3 shows the deployment of EMQ, Confluent (Kafka Broker and ZooKeeper), and Apache Spark on the e-Health-System-K8s-cluster nodepool-2. These components were deployed as pod resources and service resources in the node pool. The control of the deployment was carried out via Kubernetes through the execution of YAML files and with the help of Helm-Tiller to obtain the components from Docker Hub.

Figure A3. Detailed view of the deployment of the specific software in the cloud from the Docker Hub repository.

Appendix F

Graphical user interfaces for access to the services of the e-health system.

(a)

(b)

(c)

Figure A4. (a) Main graphical user interface for access to the services of the e-health system. (b) The graphical user interface for access to the microservices of the system core. (c) The graphical user interface for access to diet management microservices.

Appendix G

Reference for the mean and standard deviation of vital signs in normal conditions and the structure of the messages in JSON format of the patient's vital signs data.

Table A6 presents the implicit values used for the means of vital signs with their corresponding arbitrary standard deviations. The model allows the personalization of these data within a certain range to simulate other cases, such as a certain special state of the patient, including emergencies. An instance of the JSON encoded message payload and sent to the real smartphone is:

```
{"patientMedicalData":{"BodyTemp":36.018081958971,"DiastolicBloodPressure":79.9891106294567,"HeartRate":70.2594760619214,"SystolicBloodPressure":120.0101334093251},"sampleTime":"Thu Feb 27 00:00:34.680000000 +0100 2021","sensorId":"1"}
```

Table A6. Reference for the mean and standard deviation of vital signs in normal conditions.

Vital Sign	Mean	Standard Deviation
Body temperature	36 °C	0.01 °C
Diastolic pressure	80 mm Hg	0.01 mm Hg
Systolic pressure	120 mm Hg	0.01 mm Hg
Cardiac rhythm	70 beats per minute	0.01 beats per minute

Appendix H

An example of the characteristics of the requests and responses of the core operations of the system.

Table A7 shows an example of the data size of the input and output of each operation. The total size of the data sent in the "Insert One Document" and "Update One Document" operations is larger than in those operations where only the document key is necessary. On the other hand, the "Read One Document" and "Read Many" operations receive the complete content of a document or multiple documents as a response, so the responses Apache JMeter receives are larger.

Table A7. Characteristics of the requests and responses of the system core operations.

Microservice Operation	HTTP Method	Request (Bytes)	Response Header (Bytes)	Response Body (Bytes)	Response Total (Bytes)
Insert One Document (12 fields/document)	Post	452	758	77	835
Read One Document (12 fields/document)	Get	152	934	7696	8630
Read Many (20 Documents) (7 fields/document)	Get	132	935	11,777	12,712
Update One Document (12 fields/document)	Put	554	758	32	790
Delete One Document	Delete	186	758	73	831

References

1. World Health Organization. Ageing. Available online: <https://www.who.int/news-room/facts-in-pictures/detail/ageing> (accessed on 30 March 2021).
2. Casale, G.; Chesta, C.; Deussen, P.; Di Nitto, E.; Gouvas, P.; Koussouris, S.; Stankovski, V.; Symeonidis, A.; Vlassiou, V.; Zafeiropoulos, A.; et al. Current and future challenges of software engineering for services and applications. *Procedia Comput. Sci.* **2016**, *97*, 34–42. [\[CrossRef\]](#)
3. Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Gener. Comp. S* **2013**, *29*, 1645–1660. [\[CrossRef\]](#)

4. Riazul, I.S.M.; Kwak, D.; Kabir, M.H.; Hossain, M.; Kwak, K. The Internet of Things for health care: A comprehensive survey. *IEEE Access* **2015**, *3*, 678–708. [\[CrossRef\]](#)
5. Ganti, R.K.; Ye, F.; Lei, H. Mobile crowdsensing: Current state and future challenges. *IEEE Commun. Mag.* **2011**, *49*, 32–39. [\[CrossRef\]](#)
6. Oussous, A.; Benjelloun, F.; Ait, L.A.; Belfkih, S. Big Data technologies: A survey. *J. King Saud Univ. Comp. Info. Sci.* **2018**, *30*, 431–448. [\[CrossRef\]](#)
7. Ud, D.I.; Guizani, M.; Hassan, S.; Kim, B.; Khurram, K.M.; Atiquzzaman, M.; Hassan, S. The Internet of Things: A review of enabled technologies and future challenges. *IEEE Access* **2019**, *7*, 7606–7640. [\[CrossRef\]](#)
8. Reactive Manifesto. Available online: <http://www.reactivemanifesto.org> (accessed on 30 March 2021).
9. Wampler, D. *Fast Data Architectures for Streaming Applications*, 2nd ed.; O'Reilly Media Inc.: Sebastopol, CA, USA, 2018.
10. What Are Containers and Their Benefits-Google Cloud. Available online: <https://cloud.google.com/containers> (accessed on 30 March 2021).
11. Jurado, L.; Salvachúa, J. e-Health IoT reactive services for elderly care at home in Smart City built on an emerging Fast Data Architecture. In Proceedings of the 2018 International Conference on Parallel and Distributed Processing Techniques & Applications (2018 PDPTA), Las Vegas, NV, USA, 30 July–2 August 2018; pp. 35–41. Available online: <https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/PDP3615.pdf> (accessed on 30 March 2021).
12. Hemairy, M.A.; Serhani, M.A.; Amin, S.; Ahmed, M.A. Integrated and scalable architecture for providing cost-effective remote health monitoring. In Proceedings of the 2016 9th International Conference on Developments in eSystems Engineering (DeSE), Liverpool, UK, 31 August–2 September 2016; pp. 74–80. [\[CrossRef\]](#)
13. Gahlot, S.; Reddy, S.R.N.; Kumar, D. Review of smart health monitoring approaches with survey analysis and proposed framework. *IEEE Internet Things J.* **2019**, *6*, 2116–2127. [\[CrossRef\]](#)
14. Kirtana, R.N.; Lokeswari, Y.V. An IoT based remote HRV monitoring system for hypertensive patients. In Proceedings of the 2017 International Conference on Computer, Communication and Signal Processing (ICCCSP), Chennai, India, 10–11 January 2017; pp. 1–6. [\[CrossRef\]](#)
15. Pescosolido, L.; Berta, R.; Scalise, L.; Revel, G.M.; De Gloria, A.; Orlandi, G. An IoT-inspired cloud-based web service architecture for e-Health applications. In Proceedings of the 2016 IEEE International Smart Cities Conference (ISC2), Trento, Italy, 12–15 September 2016; pp. 1–4. [\[CrossRef\]](#)
16. Raji, A.; Kanchana Devi, P.; Golda Jeyaseeli, P.; Balaganesh, N. Respiratory monitoring system for asthma patients based on IoT. In Proceedings of the 2016 Online International Conference on Green Engineering and Technologies (IC-GET), Coimbatore, India, 19 November 2016; pp. 1–6. [\[CrossRef\]](#)
17. Abawajy, J.H.; Hassan, M.M. Federated Internet of Things and Cloud Computing pervasive patient health monitoring system. *IEEE Commun. Mag.* **2017**, *55*, 48–53. [\[CrossRef\]](#)
18. Vuppalapati, C.; Ilapakurti, A.; Kedari, S. The role of Big Data in creating sense EHR, an integrated approach to create next generation mobile sensor and wearable data driven Electronic Health Record (EHR). In Proceedings of the 2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService), Oxford, UK, 29 March–1 April 2016; pp. 293–296. [\[CrossRef\]](#)
19. Zhang, Y.; Qiu, M.; Tsai, C.; Hassan, M.M.; Alamri, A. Health-CPS: Healthcare Cyber-Physical System assisted by Cloud and Big Data. *IEEE Syst. J.* **2017**, *11*, 88–95. [\[CrossRef\]](#)
20. Ma'arif, M.R.; Priyanto, A.; Setiawan, C.B.; Winar Cahyo, P. The design of cost efficient health monitoring system based on Internet of Things and Big Data. In Proceedings of the 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea, 17–19 October 2018; pp. 52–57. [\[CrossRef\]](#)
21. Taher, N.C.; Mallat, I.; Agoulmine, N.; El-Mawass, N. An IoT-cloud based solution for real-time and batch processing of Big Data: Application in healthcare. In Proceedings of the 2019 3rd International Conference on Bio-Engineering for Smart Technologies (BioSMART), Paris, France, 24–26 April 2019; pp. 1–8. [\[CrossRef\]](#)
22. Cioara, T.; Anghel, I.; Salomie, I.; Barakat, L.; Miles, S.; Reidlinger, D.; Taweel, A.; Dobre, C.; Pop, F. Expert system for nutrition care process of older adults. *Future Gener. Comp. S* **2018**, *80*, 368–383. [\[CrossRef\]](#)
23. Wickramasinghe, M.P.N.; Perera, D.M.; Kahandawaarachchi, K.A.D. Dietary prediction for persons with Chronic Kidney Disease (CKD) by considering blood potassium level using ML algorithms. In Proceedings of the 2017 IEEE Life Sciences Conference (LSC), Sydney, NSW, Australia, 13–15 December 2017; pp. 300–303. [\[CrossRef\]](#)
24. Alloghani, M.; Hussain, A.; Al-Jumeily, D.; Fergus, P.; Abuelmaatti, O.; Hamden, H. A mobile health monitoring application for obesity management and control using the internet-of-things. In Proceedings of the 2016 Sixth International Conference on Digital Information Processing and Communications (ICDIPC), Beirut, Lebanon, 21–23 April 2016; pp. 19–24. [\[CrossRef\]](#)
25. Harous, S.; Serhani, M.A.; El Menshawy, M.; Benharref, A. Hybrid obesity monitoring model using sensors and community engagement. In Proceedings of the 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC), Valencia, Spain, 26–30 June 2017; pp. 888–893. [\[CrossRef\]](#)
26. Dutta, J.; Gazi, F.; Roy, S.; Chowdhury, C. AirSense: Opportunistic crowd-sensing based air quality monitoring system for smart city. In Proceedings of the 2016 IEEE SENSORS, Orlando, FL, USA, 30 October–3 November 2016; pp. 1–3. [\[CrossRef\]](#)
27. Rozanski, N.; Woods, E. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, 2nd ed.; Addison Wesley: Boston, MA, USA, 2012.

28. Wolff, E. *Microservices: Flexible Software Architecture*; Addison-Wesley: Boston, MA, USA, 2016.
29. Bass, L.; Weber, I.; Zhu, L. *DevOps: A Software Architect's Perspective*; Addison Wesley: Boston, MA, USA, 2015.
30. Ibryam, B.; Huß, R. *Kubernetes Patterns*; O'Reilly Media Inc.: Boston, MA, USA, 2019.
31. Qian, K.; Fu, X.; Tao, L.; Xu, C.; Diaz-Herrera, J. *Software Architecture and Design Illuminated*; Jones and Bartlett Publishers: Burlington, MA, USA, 2009.
32. Richards, M. *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*; O'Reilly Media Inc.: Sebastopol, CA, USA, 2015.
33. Amundsen, M.; McLarty, M.; Mitra, R.; Nadareishvili, I. *Microservice Architecture*; O'Reilly Media Inc.: Sebastopol, CA, USA, 2016.
34. Hwang, K.; Dongarra, J.; Fox, G.C. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2012.
35. Marinescu, D.C. *Cloud Computing: Theory and Practice*, 2nd ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2017.
36. GitLab: The Entire DevOps Lifecycle in One Application. Available online: <https://about.gitlab.com/stages-devops-lifecycle/> (accessed on 30 March 2021).
37. Docker: What Is a Container? Available online: <https://www.docker.com/resources/what-container> (accessed on 30 March 2021).
38. Docker Hub Quickstart. Available online: <https://docs.docker.com/docker-hub/> (accessed on 30 March 2021).
39. U.S. Department of Health and Human Services; U.S. Department of Agriculture. *2015–2020 Dietary Guidelines for Americans*, 8th ed.; U.S. Department of Health and Human Services: Washington, DC, USA, 2015. Available online: https://health.gov/dietaryguidelines/2015/resources/2015-2020_Dietary_Guidelines.pdf (accessed on 30 March 2021).
40. Catharine, R.A.; Caballero, B.H.; Cousins, R.J.; Tucker, K.L.; Ziegler, T.R. *Modern Nutrition in Health and Disease*, 11th ed.; Wolters Kluwer Health Adis (ESP): Philadelphia, PA, USA, 2014.
41. Google Cloud: Products & Services. Available online: <https://cloud.google.com/products/> (accessed on 30 March 2021).
42. Kubernetes: What Is Kubernetes. Available online: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> (accessed on 30 March 2021).
43. Play Framework. Available online: <https://www.playframework.com/> (accessed on 30 March 2021).
44. MongoDB: The Database for Modern Applications. Available online: <https://www.mongodb.com/> (accessed on 30 March 2021).
45. MySQL-MySQL 8.0 Reference Manual-1.2.1 What Is MySQL? Available online: <https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html> (accessed on 30 March 2021).
46. EMQ: The Massively Scalable MQTT Broker for IoT and Mobile Applications. Available online: <http://emqtt.io/> (accessed on 30 March 2021).
47. Confluent Platform. Available online: <https://www.confluent.io/product/confluent-platform/> (accessed on 30 March 2021).
48. Confluent Platform: Kubernetes Helm Charts. Available online: https://docs.confluent.io/5.1.0/installation/installing_cp-cp-helm-charts/docs/index.html (accessed on 30 March 2021).
49. Apache Spark: Lightning-Fast Unified Analytics Engine. Available online: <https://spark.apache.org/> (accessed on 30 March 2021).
50. Spark 2.4.0: Running Spark on Kubernetes. Available online: <https://spark.apache.org/docs/2.4.0/running-on-kubernetes.html> (accessed on 30 March 2021).
51. Akka Streams Kafka. Available online: <https://doc.akka.io/docs/alpakka-kafka/0.11/home.html> (accessed on 30 March 2021).
52. Maven Repository: Paho Akka. Available online: https://mvnrepository.com/artifact/com.sandinh/paho-akka_2.11/1.3.0 (accessed on 30 March 2021).
53. MongoDB Sink. Available online: <https://docs.lenses.io/connectors/sink/mongo.html#kubernetes> (accessed on 30 March 2021).
54. Spark 2.4.0 Documentation: Spark Streaming + Kafka Integration Guide. Available online: <https://spark.apache.org/docs/2.4.0/streaming-kafka-integration.html> (accessed on 30 March 2021).
55. MongoDB Documentation: MongoDB Connector for Spark. Available online: <https://docs.mongodb.com/spark-connector/master/> (accessed on 30 March 2021).
56. Android Developers: Platform Architecture. Available online: <https://developer.android.com/guide/platform> (accessed on 30 March 2021).
57. EMQ 2.2-Erlang MQTT Broker: User Guide. Available online: <https://emq-docs-en.readthedocs.io/en/latest/guide.html> (accessed on 30 March 2021).
58. Confluent: Security. Available online: <https://docs.confluent.io/current/security/index.html> (accessed on 30 March 2021).
59. Silhouette. Available online: <https://www.silhouette.rocks/> (accessed on 30 March 2021).
60. MongoDB Documentation: Security. Available online: <https://docs.mongodb.com/manual/security/> (accessed on 30 March 2021).
61. Spark 2.4.0 Documentation: Security. Available online: <https://spark.apache.org/docs/2.4.0/security.html> (accessed on 30 March 2021).
62. Google Cloud: Google Infrastructure Security Design Overview. Available online: <https://cloud.google.com/security/infrastructure/design/?hl=es-419> (accessed on 30 March 2021).
63. Kubernetes: Overview of Cloud Native Security. Available online: <https://kubernetes.io/docs/concepts/security/overview/> (accessed on 30 March 2021).

64. Selvaraj, P.; Doraikannan, S. Privacy and Security Issues on Wireless Body Area and IoT for Remote Healthcare Monitoring. In *Intelligent Pervasive Computing Systems for Smarter Healthcare*; Sangaiah, A.K., Shantharajah, S., Theagarajan, P., Eds.; JohnWiley & Sons Inc.: Hoboken, NJ, USA, 2019; pp. 227–253.
65. Li, Y.; Jeong, Y.; Shin, B.; Park, J.H. Crowdsensing Multimedia Data: Security and Privacy Issues. *IEEE MultiMedia* **2017**, *24*, 58–66. [CrossRef]
66. Google Cloud–Compliance. HIPAA. Available online: <https://cloud.google.com/security/compliance/hipaa-compliance> (accessed on 30 March 2021).
67. HIPAA Compliance–Amazon Web Services (AWS). Available online: <https://aws.amazon.com/compliance/hipaa-compliance/> (accessed on 30 March 2021).
68. Al-Marsy, A.; Chaudhary, P.; Rodger, J.A. A Model for Examining Challenges and Opportunities in Use of Cloud Computing for Health Information Systems. *Appl. Syst. Innov.* **2021**, *4*, 15. [CrossRef]
69. Apache JMeter: Apache JMeter Distributed Testing Step-by-Step. Available online: https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html (accessed on 30 March 2021).
70. Load Testing as a Service (LTaaS) with Apache Jmeter on kubernetes. Available online: <https://github.com/kubernauts/jmeter-kubernetes> (accessed on 30 March 2021).
71. Understanding Blood Pressure Readings–American Heart Association. Available online: <https://www.heart.org/en/health-topics/high-blood-pressure/understanding-blood-pressure-readings> (accessed on 30 March 2021).
72. Body Temperature Norms: MedlinePlus Medical Encyclopedia. Available online: <https://medlineplus.gov/ency/article/001982.htm> (accessed on 30 March 2021).
73. All About Heart Rate (Pulse)–American Heart Association. Available online: <https://www.heart.org/en/health-topics/high-blood-pressure/the-facts-about-high-blood-pressure/all-about-heart-rate-pulse> (accessed on 30 March 2021).
74. Montgomery, D.C.; Runger, G.C. *Applied Statistics and Probability for Engineers*, 6th ed.; John Wiley & Sons: New York, NY, USA, 2014.
75. Ptolemy II Home Page. Available online: <http://ptolemy.eecs.berkeley.edu/ptolemyII/> (accessed on 30 March 2021).
76. Majumder, S.; Mondal, T.; Deen, M.J. Wearable Sensors for Remote Health Monitoring. *Sensors* **2017**, *17*, 130. [CrossRef] [PubMed]
77. Measure Performance with the RAIL Model. Available online: <https://web.dev/rail/#goals-and-guidelines> (accessed on 30 March 2021).
78. Vohra, D. *Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms*, 1st ed.; Apress: New York, NY, USA, 2017.
79. Types of Clusters–Kubernetes Engine Documentation–Google Cloud. Available online: <https://cloud.google.com/kubernetes-engine/docs/concepts/types-of-clusters> (accessed on 30 March 2021).
80. Pod–Kubernetes Engine Documentation–Google Cloud. Available online: <https://cloud.google.com/kubernetes-engine/docs/concepts/pod> (accessed on 30 March 2021).
81. Assigning Pods to Nodes–Kubernetes. Available online: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/> (accessed on 30 March 2021).
82. Overview of Deploying Workloads–Kubernetes Engine Documentation. Available online: <https://cloud.google.com/kubernetes-engine/docs/how-to/deploying-workloads-overview> (accessed on 30 March 2021).
83. StatefulSets–Kubernetes. Available online: <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/> (accessed on 30 March 2021).