Monitoring Architecture for Services and Servers in Healthcare Environment

Vasco Ramos

UMinho | 2022

Universidade do Minho
Escola de Engenharia

Vasco António Lopes Ramos

Monitoring Architecture for Services and Servers in Healthcare Environment

September, 2022

Universidade do Minho
Escola de Engenharia

Vasco António Lopes Ramos

Monitoring Architecture for Services and Servers in Healthcare Environment

Master's Dissertation
Master in Informatics Engineering

Work developed under the supervision of:
Doctor Hugo Daniel Abreu Peixoto

September, 2022

Acknowledgements

I would like to thank the ALGORITMI Center and the Portuguese Foundation for Science and Technology (FCT) for financial support through national funds and, when applicable, co-financed by the FEDER, within the scope of the project "Integrated and Innovative Solutions for the well-being of people in complex urban centers" (UIDB/00319/2020).

To Hugo Peixoto, I am grateful for all the opportunities, advice and guidance provided during my academic career which lead to the development of this project. Thank you also for your availability to accompany me regularly and for all your attention.

To my friend and colleague, Carolina Marques, who, besides supporting me unconditionally and being an excellent peer work partner, in parallel to this work, developed the web application that allows the result of this dissertation to be used more interactively and visually by its users.

I also thank those who, not having contributed directly to this work, have shown their support during my academic experience:

To those who accompany me and are my friends since my first year of university, Daniel Nunes and Pedro Ferreira, because they were not only colleagues of study or academic ventures, but also showed me their unconditional friendship in the ups and downs of this path and long after we followed different paths in our lives.

To João Vasconcelos, my colleague of choice in academic work and adventures, from whom I learned a lot and who always encouraged me to give the best of myself, inside and outside the classroom. Also, to Raquel Ramos and Margarida Silva, for teaching me so much in areas I never expected to learn and for putting up with me whenever I needed it.

To more recent friends, Maria Araújo and Constança Elias, who in the last two years have walked the path with me in the master's degree that ends with this work, for making me feel so welcomed and integrated in this different stage of my life and for always contributing to a more optimistic perspective of life's adversities.

To Maria Soeiro, Daniela Pais and Mariana Pinto, my long-time friends, always there to offer me good advice and celebrate new milestones.

And last, but certainly never least, to my parents, as it is undeniable that it would be impossible to get here without your love and support. I thank you for your encouragement to academic excellence from an early age.

STATEMENT OF INTEGRITY

“Software is a great combination between artistry and engineering.” (Bill Gates)

Abstract

Monitoring Architecture for Services and Servers in Healthcare Environment

Information systems are continuously evolving in nature and complexity. Infrastructure concerns such as availability, efficiency, and disaster recovery have been some of the most important drivers regarding how infrastructure is planned and executed. As these mechanisms evolved, so did the underlying foundation for their functioning — Information Technology (IT) infrastructure monitoring.

Inside the healthcare environment, it is important to discuss IT infrastructure monitoring and disaster prevention and recovery since availability and communication are vital for the proper functioning of healthcare units, whether acting in isolation or on a network. When acting on a network, it is especially important to be able to easily monitor and observe each unit from a single point of access so that actions can be swiftly taken when there is a problem.

Considering the wide range of available solutions and heterogeneous nature of IT infrastructure, even within the healthcare industry, the majority of the solutions either focus too much on a particular problem of some industry, healthcare or not, or are too generic and can't fulfill the needs of an increasingly connected and interdependent healthcare industry.

This Dissertation proposes a web and microservices-based IT infrastructure monitoring backend solution with a multi-site and multi-organization scheme at its core that is designed to be scalable, easily deployed and integrated with existing tools, and simple to further extend and improve. This solution has two main components, one server which is the central point of the solution, the guardian server, and the other one, which is the local client to be installed on each organization's infrastructure.

The produced backend solution was tested and validated in two healthcare organizations which provided useful feedback and a positive answer to the usefulness of a monitoring solution, such as the one developed in this Dissertation, in improving the efficiency and reliability of the organizations' IT infrastructure and, therefore, their healthcare services. A formal evaluation of the solution was also carried out with a combination of a Strengths, Weaknesses, Opportunities and Threats (SWOT) analysis and a risk assessment report, both mechanisms providing useful insights on the strengths and limitations of the solution, as well as possible improvement points.

Keywords: IT Infrastructure Monitoring, Microservices, Healthcare, Backend Architecture, Containerization

Resumo

Arquitetura de Monitorização para Serviços e Servidores em Ambiente Hospitalar

Os sistemas de informação estão em constante evolução tanto em índole, como complexidade. Questões como disponibilidade, eficiência e recuperação de falhas têm sido alguns dos fatores mais importantes no que diz respeito ao planeamento e execução de infraestruturas. À medida que esses mecanismos evoluíram, o mesmo aconteceu com a base subjacente para o seu funcionamento — a monitorização de infraestruturas de Tecnologia de Informação (TI).

O ambiente hospitalar é particularmente relevante quando se discute monitorização de infraestruturas de TI e a prevenção e recuperação de desastres, uma vez que a disponibilidade e a comunicação com terceiros são vitais para o bom funcionamento das unidades de cuidados de saúde, quer estas atuem isoladamente ou em rede. Ao atuar em rede, é especialmente importante ser capaz de facilmente monitorizar e observar cada unidade de saúde a partir de um único ponto de acesso, de modo a que, quando houver um problema, se possa agir de forma rápida e integrada.

Tendo em conta a vasta gama de soluções de monitorização disponíveis e a heterogeneidade das infraestruturas de TI, a maioria das soluções ou se concentra demasiado num problema específico de uma dada indústria ou setor, de cuidados de saúde ou não, ou é demasiado genérica e não consegue satisfazer as necessidades de uma indústria de cuidados de saúde cada vez mais conexa e interdependente.

Esta Dissertação propõe uma solução *backend* de monitorização de infraestruturas de TI baseada em microserviços *web*, com um esquema multi-local e multi-organização na sua fundação, que foi concebida para ser escalável, facilmente instalada e integrada com ferramentas já existentes, e simples de expandir e melhorar. Esta solução tem dois principais componentes, um servidor que é o ponto central da solução, o *guardian server*, e o outro que é o cliente local a ser instalado na infraestrutura de cada organização.

A solução *backend* produzida foi testada e validada em duas organizações de cuidados de saúde que forneceram opiniões úteis e construtivas, bem como uma resposta positiva à utilidade de uma solução de monitorização, como a desenvolvida nesta Dissertação, para melhorar a eficiência e fiabilidade da infraestrutura de TI das organizações e, consequentemente, dos cuidados de saúde que estas prestam. Foi também realizada uma avaliação formal da solução através da combinação de uma análise SWOT e de um relatório de avaliação de risco, em que ambos forneceram informação útil sobre os pontos fortes e limitações da solução, bem como possíveis pontos de melhoria.

Palavras-chave: Monitorização de Infraestruturas IT, Micro-serviços, Ambiente Hospitalar, Arquitetura de Backend, Containerização

Contents

List of Figures

List of Tables

List of Examples

Acronyms

API	Application Programming Interface 4, 8, 10, 15, 16, 18, 19, 20, 25, 26, 27, 31, 36, 38, 39, 40, 41, 44, 45, 48, 51, 53, 54, 56, 57, 58, 60, 72, 74
B2B	Business-to-Business 19, 31
CD	Continuous Delivery 55
CHTS	Tâmega e Sousa Hospital Center 2, 28, 62, 71, 73
CI	Continuous Integration 55
CORS	Cross-Origin Resource Sharing 44
CSV	Comma-Separated Values 50
DoS	Denial-of-Service 18
DSRM	Design Science Research Methodology 22, 23, 27
EADT	European Association for Digital Transition 1
EHR	Electronic Health Record 1
ELK	Elasticsearch, Logstash, Kibana 7, 8, 9, 10, 13, 14, 15, 20
EU	European Union 1
HTTP	Hypertext Transfer Protocol 4, 10, 11, 12, 15, 18, 19, 24, 25, 31, 33, 35, 37, 41, 42, 43, 48, 50, 60, 67
HTTPS	Hypertext Transfer Protocol Secure 44
IP	Internet Protocol 36
IT	Information Technology vi, vii, 2, 3, 4, 6, 7, 8, 9, 11, 18, 19, 21, 29, 31, 32, 40, 58, 62, 66, 67, 68, 71, 72, 73, 74, 75
JSON	JavaScript Object Notation 8, 18, 25, 26, 44, 45, 49, 50

JWT	JSON Web Token xi, 18, 19, 24, 46, 47
NoSQL	Not only SQL 19, 25, 32
OOP	Object-Oriented Programming 25, 48
RAM	Random-Access Memory 14
REST	Representational State Transfer 8, 25
SaaS	Software as a Service 15
SMS	Short Message Service 11, 75
SMTP	Simple Mail Transfer Protocol 11
SQL	Structured Query Language 10, 14, 19, 32
SSD	Solid State Drive 14
SSL	Secure Sockets Layer 36
SWOT	Strengths, Weaknesses, Opportunities and Threats vi, 5, 62, 66, 67, 68, 71, 73
TICK	Telegraf, InfluxDB, Chronograf, Kapacitor 7, 8, 9, 10, 11, 13, 14, 15, 20
TLS	Transport Layer Security 18
TSDB	Time Series Database 10, 12, 15, 21, 25, 26, 36
USLGuarda	Hospital Sousa Martins - Local Health Unit of Guarda 2, 28, 62, 71, 73
YAML	Yet Another Markup Language 26, 45, 51, 56

Chapter 1

Introduction

"Digitisation and the development of technologies must serve to strengthen healthcare services, improving citizens' access to these services, aiding better clinical practice, and enabling a healthcare plan that seeks to protect the health of persons and of society as a whole."

—European Association for Digital Transition (EADT) in [1]

The digital transition has been, for many years, a priority in the European Union (EU) in all sectors of society and with a particular focus on healthcare. This transition from analog systems to digital ones in healthcare was introduced and justified as a way to improve healthcare quality, efficiency and safety, as well as to reduce costs and create new service innovations [2]. In this spirit, new and more complex information systems and digital standards have arisen, such as the Electronic Health Record (EHR) and alternative frameworks surrounding their usage and implementation (e.g openEHR[1] and FHIR[2]). Thus, healthcare is becoming more and more digital, with the hope of increasing the quality and efficiency of healthcare services and, therefore, improving the lives of the citizens who need them and everyone around them.

According to the World Health Organization in [3], health information systems have four key functions: data generation, compilation and analysis, data synthesis, data communication, and data use. By this definition, an ideal health information system collects data from the healthcare sector and other relevant sectors, analyses and ensures the overall quality, significance and opportuneness of the data, converts that data into information and knowledge for health-related decision-making and, finally, shares that data with other, third-party, health information systems, enabling interoperability within the healthcare sector.

[1]https://www.openehr.org
[2]https://www.hl7.org/fhir

Being such systems so important and central to the proper functioning of the healthcare sector, it is essential to guarantee that these do not fail or, at the very least, fail as little as possible. This effort can only be achieved with a comprehensive and effective IT monitoring system. Insofar as health information systems are a tool that supports the decision-making process in the management of patient's health, IT monitoring systems, not just in the healthcare sector, are the tool that supports the decision-making process related to the management of the infrastructure that hosts the services and systems in question, that is, the health information systems.

Thereby, this Dissertation intends to provide an answer to what a comprehensive IT monitoring solution with a particular focus on multi-organization and multi-site healthcare environments, where different healthcare units work as a network to provide their services, should comprise and whether this architecture topology would benefit the healthcare organizations, both in the simplification and maintainability of their IT monitoring efforts and the availability and reliability of their systems. The study, evaluation and understanding of such concepts and possibilities is paramount to the advancement of the healthcare industry since quality and efficient health information systems help healthcare organizations to achieve and expand their main and ultimate goal, which is to provide their patients with the best care and services possible.

This Dissertation's work is part of a larger project within the ALGORITMI Center tasked with the development of a fully-functional IT monitoring solution for the healthcare sector. One of this initiative's other projects is the development of a web application by a colleague, Carolina Marques, in the scope of her own master's Dissertation. This web application's core goal is to integrate with the monitoring backend system developed in this Dissertation and serve as an interactive interface of the available monitoring management and consumption capabilities. Furthermore, it is worth noting this Dissertation also had the cooperation of two healthcare organizations, Tâmega e Sousa Hospital Center (CHTS) and Hospital Sousa Martins - Local Health Unit of Guarda (USLGuarda), both in the early understanding of the problem and the requirements for a solution to overcome them, as well as to test and validate the developed solution within their systems, allowing for two real-life case studies regarding the application of this Dissertation's work.

In this chapter, it will be introduced the motivation for this work, followed by a specification of the goals to be achieved and, finally, the outline of the document, summarising the main contents of each chapter.

1.1 Motivation

As stated above, the push of transitioning from the analog to the digital, within the healthcare industry, originated environments that rely more and more on technology and information systems to provide efficient and quality healthcare services to patients. However, it is important to ensure that technology does not fail when it is needed most [4]. In this context, IT infrastructure monitoring systems are essential, as they allow the analysis of the status of the provided services, as well as various metrics related to the servers and infrastructure on which they are hosted [5].

Moreover, the healthcare industry has evolved over the years into an interconnected reality where different healthcare units, whether belonging to a parent healthcare organization or not, work as part of a larger network of healthcare facilities, available to all patients. This flexibility and cooperation was built using interoperability technology and has allowed patients to quickly access healthcare services inside and outside their residential areas and even across countries. It has also been observed that the use of similar systems and services between different healthcare entities, whether they belong to the same organization or not, has increased. Hence, the importance of being able to create and analyze monitoring engines in an integrated way between different health units.

The motivation for this Dissertation came from realizing the difficulty and repetitive process of creating and replicating monitoring structures in healthcare facilities that belong to the same health organization or use the same information systems. This process is usually manual and not very efficient, which can lead to inconsistencies and failures in the ability to observe the state of the infrastructure at any given moment.

Therefore, to support these specificities, emerged the need to evaluate the current context of IT infrastructure monitoring solutions and design an architecture that, on one hand, supports a multi-organization and multi-site reality and, on the other hand, allows a dynamic configuration and management of the enforced monitoring mechanisms. Also important for this type of architecture is the decoupling of responsibilities and interoperability with third-party solutions, meaning it is essential that the designed architecture can export the monitoring data so that it can be used by a wide variety of information visualization systems and dashboards.

1.2 Goals and Expected Results

This Dissertation had as its origin a guiding research objective that transposes into the following question: "Can a multi-site and multi-organization monitoring solution improve the availability of the highly heterogeneous IT infrastructures of the healthcare industry?".

With this guiding question in mind, the central goal of this work is to create a robust and replicable backend monitoring architecture suitable for a multi-site and multi-organization healthcare environment and to carry out its validation. In turn, this goal can be broken down into several intermediate goals:

- Explore the different aspects (advantages and disadvantages) between monolithic and microservices architectures.
- Explore deployment and containerization concepts such as Docker and/or Kubernetes.
- Design the backend architecture for a multi-site and multi-organization monitoring backend architecture.
- Create and design a solution prototype artifact, which should include the following features:

  - Creation and management of various monitoring mechanisms (ping, Hypertext Transfer Protocol (HTTP), databases, etc.).
  - Real-time and scheduled multi-channel alerting.
  - User management, with data privacy and security mechanisms.
  - Storage data structure that can handle both local and global data (to support multi-site and multi-organization environments).
  - Production and consumption of monitoring data that can be interpreted by external clients, such as a frontend layer, via Application Programming Interface (API)s that comply with current web and data export standards.
  - Simple and reliable multi-environment deployment.

- Implement the solution's prototype as the Dissertation's final artifact.
- Test, validate and evaluate the system, using both subjective and formal analysis methods.

1.3 Document Outline

The content of this Dissertation Report is organized into seven chapters.

Chapter 1 (Introduction), describes the overall context and motivation of this work, presents the goals to be achieved and expected results and, finally, outlines the document's structure and its main contents.

Chapter 2 (State of the Art) introduces the necessary background context and state-of-the-art related to the topics of this Dissertation. The purpose and relevance of IT infrastructure monitoring are described and an analysis and comparison of the various existing IT infrastructure monitoring solutions, architectural options and web security concerns are provided. The chapter concludes with a proposal on how this Dissertation can contribute to this topic and overcome the current issues and shortcomings of the available commercial and open-source solutions.

In chapter 3 (Research Methodology and Technologies), it is presented the selected methodologies, mechanisms and technologies that were used in the research, design and implementation of this Dissertation.

Chapter 4 (Proposal) specifies the system modeling and architectural artifacts used to design this Dissertation's work. First, there is a description of the conducted technical questionnaire and a presentation of its results. Then, a detailed enumeration and explanation of the system requirements, followed by an explanation of the solution's domain model. Finally, the system architecture, both application and deployment, is presented. The chapter is concluded with a discussion of the proposed architectural design in light of the specified requirements.

In chapter 5 (Implementation and Results), a detailed description of the solution's implementation is presented. First, are addressed the selected data models, followed by a description of the internal components of each server, i.e., guardian and local servers. Then, are presented the documentation and deployment mechanisms. Finally, are showcased examples of the solution's core features and results external to this Dissertation which demonstrate its functionality and integration capabilities.

Chapter 6 (Validation and Discussion) describes the validation and evaluation of the implemented solution, with a parallel discussion of its results. First, the results from the validation via real-life case studies are presented, followed by a formal evaluation comprised of a SWOT analysis and a risk assessment report. This chapter is concluded with some observations on the solution's quality and effectiveness as well as a brief discussion regarding the overall outcome of this Dissertation and whether it was successful or not.

Finally, in chapter 7 (Conclusion), it is summarised the developed work and some final remarks on the accomplished results are made. It is also presented a compact enumeration of this Dissertation's technical and scientific contributions, as well as some future work suggestions.

Chapter 2

State of the Art

This chapter presents the current state of the art related to this work. First, the purpose and relevance of IT infrastructure monitoring solutions are described. Then, a description and subsequent comparison of the different existing monitoring solutions and a categorical architecture difference in terms of data collection mechanisms are presented. There is also a discussion on the various existing software architectures, with particular emphasis on the monolithic architecture and the microservices one, as well as the current concerns regarding web security and the mechanisms to overcome them. The chapter is concluded with a discussion of the solutions and approaches presented and how this Dissertation can contribute to this topic.

2.1 Monitoring Architectures: Pull vs. Push

A monitoring solution that is intended to be useful and provide accurate information about the monitored infrastructure must have data. To analyze the system's performance and reliability history as well as its current status in order to better understand possible future problems and constraints in the infrastructure, the monitoring solution must have large amounts of data. As such, it is clear that collecting data is an important part of any monitoring solution, whether simple or complex. Therefore, data collection is one of the most important aspects of IT infrastructure monitoring and one of the most controversial, as the chosen paradigm to accomplish this task is a subject of great discussion and debate [6], [7]. From a data collection perspective, there are two main monitoring architectures: **Pull** and **Push**, the main difference being the location of the data collection component [8], [9].

In the Pull architecture, the data collection component is the active part, which means that the agent requests a remote node to send data about itself [10]. These requests to collect metrics from the monitored targets occur at regular intervals and there is a central service responsible for this scheduling. This paradigm is satisfactory in scenarios such as network monitoring, although some authors argue that pull-based data collection is always a bad idea [8], [10]. The negative observation is due to perceived inefficiencies in IT fields such as server and application monitoring and scalability issues (since centralized systems are required to keep track of all known clients, handle scheduling, and parse returned data) [8]. Despite this line of thinking, there are important monitoring systems that follow a pull-based architecture, such as Prometheus and Nagios.

In the Push architecture, the data collection component is passive, and the metrics and events are sent by the client, i.e., the client (a server, an application, a metrics aggregator agent) sends data to the data collection component. The client can do this at regular intervals or when events occur. The collection of metrics and events is distributed to clients, eliminating the need for a central server to coordinate and manage monitored targets and polling schedules. Therefore, it is horizontally scalable and can have better redundancy and high availability. An excellent example of a push-based architecture that operates on a regular schedule is collectd[1].

To create a more comprehensive and dynamic paradigm, He Huang *et. al.*, in [12], explored and developed a hybrid Push/Pull model that combines the positive aspects of both approaches and mitigates their weaknesses. The authors concluded that with this type of hybrid approach, it is possible to achieve the high efficiency of a pull-based model and the high scalability of a push-based model. Other hybrid models and integrations have also been developed in commercial and open-source monitoring systems, such as the Telegraf, InfluxDB, Chronograf, Kapacitor (TICK) stack or the Elasticsearch, Logstash, Kibana (ELK) stack. Although these solutions are push-based monitoring solutions at their core, they have an optional metrics collector/aggregator that leverages some of the features of a pull-based architecture to facilitate and streamline metrics standardization.

2.2 Monitoring Tools and Solutions

Infrastructure problems are inevitable, and all infrastructure fails at some point. When it does fail, IT monitoring tools are essential to mitigate the impact of the incident. IT monitoring tools, also known as observability solutions, include a broad class of products that analysts and IT support teams can use to determine whether IT infrastructure is online and meets the expected service levels. These tools, in conjunction with infrastructure automation and provisioning, also play a large role in optimizing the available IT infrastructure to meet expected service levels at the lowest possible cost. Being such an important asset for any business and IT service, there is a significant market of IT infrastructure monitoring solutions, which brings the need to define objective evaluation and comparison aspects to perform more accurate and efficient analysis and ultimately decide which of the available solutions to choose. The aspects to be evaluated in any monitoring system are data collection, data storage, visualization, alerting, and data extraction [8]. These aspects are going to be taken into account in the following analysis. The

[1]A daemon, which collects system and application performance metrics at regular intervals [11].

monitoring solutions that are going to be analysed and compared are: ELK stack[2], TICK stack[3], Nagios[4], and Prometheus[5]. The main reasons for choosing these tools over others are primarily their modular and somewhat decoupled architectures and their open-source nature.

2.2.1 ELK Stack

The ELK stack, also known as Elastic Stack, is a comprehensive monitoring and log analysis solution built on a foundation of three open-source projects: Elasticsearch[6], Logstash[7], and Kibana[8]. ELK uses Elasticsearch for deep search and data analytics; Logstash for centralized logging management and finally, Kibana for powerful and insightful data visualisation [13].

Figure 1 shows the overall architecture of ELK stack with the referred three components and, as mentioned in section 2.1, the push paradigm on which ELK stack is based.

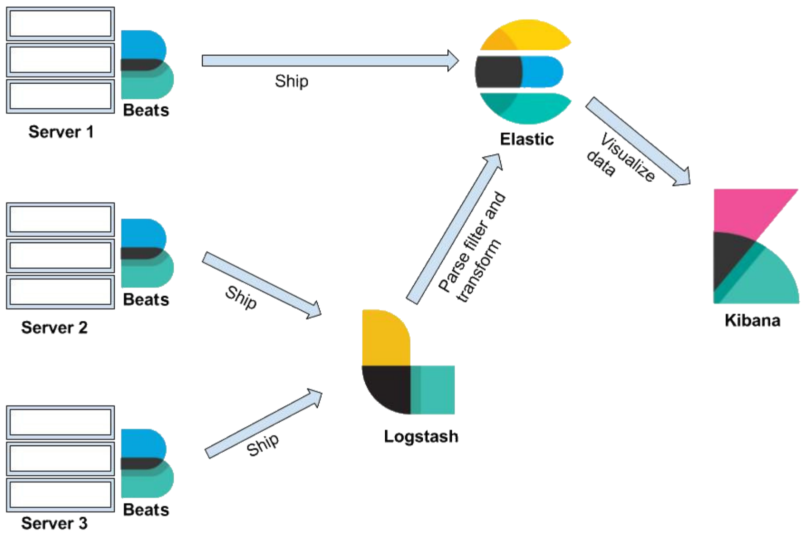


Figure 1: ELK stack architecture. Source: [14]

Elasticsearch is a distributed, document-oriented search and analytics engine that provides its functions with a sophisticated Representational State Transfer (REST) API, through JavaScript Object Notation (JSON). It provides horizontal scalability, reliability and multi-tenancy fulfilling current IT standards for high availability and performance.

Logstash is a data pipeline that enables the collection, parsing, and analysis of a rich class of structured and unstructured data. It is also capable of processing events that originate from a variety of systems. It provides plugins to connect to various input sources and platforms, such as logs, events, and metrics. In

[2]https://www.elastic.co/what-is/elk-stack
[3]https://www.influxdata.com/time-series-platform
[4]https://www.nagios.org
[5]https://prometheus.io
[6]https://www.elastic.co/elasticsearch
[7]https://www.elastic.co/logstash
[8]https://www.elastic.co/kibana

addition to the existing plugins, it is also possible to develop and publish custom plugins, contributing to an already large variety of custom plugins.

Kibana is a data visualization platform that simplifies visualizing all data (structured or unstructured) stored in Elasticsearch. It provides flexible real-time analytics and reports with highly customizable settings.

Although the three tools described above (Elasticsearch, Logstash, and Kibana) are the main foundations of the ELK stack, there is a fourth module, introduced more recently, that has significantly streamlined the process of gathering monitoring information, especially related to system and application metrics — Beats[9]. Beats is a collection of data shippers installed as agents on servers, containers, or deployed as functions. From there, those agents can send data directly to Elasticsearch or forward it to Logstash for further processing. Beats is capable of collecting various data, such as audit data (Auditbeat[10]), log files (Filebeat[11]), availability (Heartbeat[12]), metrics (Metricbeat[13]), among others.

The ELK stack has a strong foothold in the healthcare industry, according to their website. It is used mostly in the analysis of services' and systems' logs of healthcare organizations to aid the monitoring, troubleshooting and recovering of their IT infrastructures. Pfizer Digital and UCLA Health are some of their healthcare highest profile clients. Outside of healthcare, companies like Walmart, Adobe and Audi also take advantage of ELK stack capabilities to monitor their infrastructures [15].

2.2.2 TICK Stack

The TICK stack, also known as InfluxData Platform, is a widely-used general-purpose monitoring platform. It is a loosely coupled but tightly integrated set of four open-source projects — Telegraf[14], InfluxDB[15], Chronograf[16] and Kapacitor[17] — designed to collect and store massive amounts of timestamped data, with data processing, real-time monitoring and alerting capabilities. TICK uses Telegraf to collect and report metrics from various systems, applications, and environments, through the usage of plugin agents; InfluxDB for high-performance time-series storage and querying; Chronograf for an easy-to-setup administrative user interface and visualization engine and finally, Kapacitor for native data processing and data streaming [16].

Figure 2 shows the TICK stack architecture with the referred four components and, as mentioned in section 2.1, the predominantly push-based paradigm on which TICK stack is built. The component that is a hybrid between the pull and push paradigms is Telegraf, which first retrieves the data from the monitored targets and only then formats the metrics into InfluxDB Line Protocol and sends them to InfluxDB [17].

[9] https://www.elastic.co/beats
[10] https://www.elastic.co/beats/auditbeat
[11] https://www.elastic.co/beats/filebeat
[12] https://www.elastic.co/beats/heartbeat
[13] https://www.elastic.co/beats/metricbeat
[14] https://www.influxdata.com/time-series-platform/telegraf
[15] https://www.influxdata.com/products/influxdb
[16] https://www.influxdata.com/time-series-platform/chronograf
[17] https://www.influxdata.com/time-series-platform/kapacitor

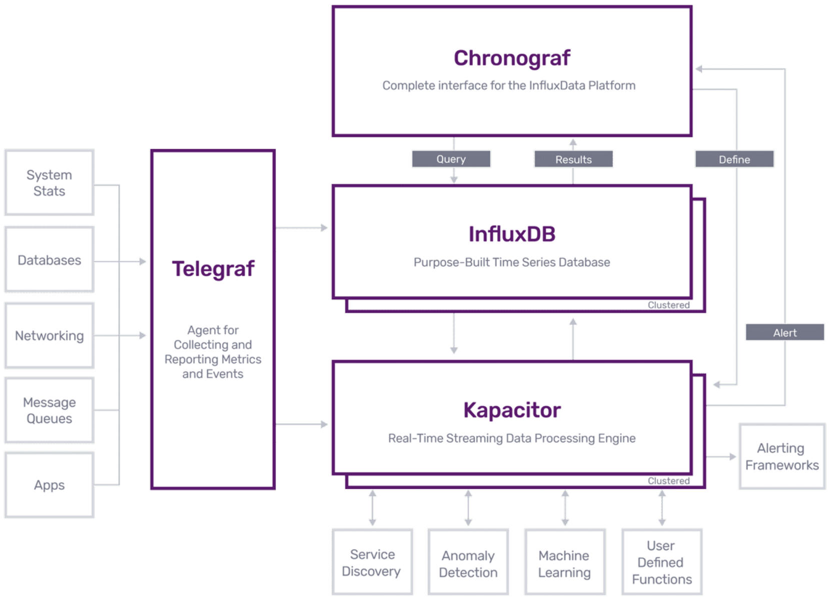


Figure 2: TICK stack architecture. Source: [18]

Telegraf is the module responsible for data collection within the TICK stack. It is designed with a plugin-based architecture with five different classes of plugins: Inputs, Outputs, Aggregators, Processors, and External [19]. As with Beats, from the ELK stack, Telegraf allows custom plugins to be developed and published, enabling a rapidly growing plugin ecosystem.

InfluxDB is a Time Series Database (TSDB) designed for high-speed data queries and writes and is a core component of the TICK stack. As part of a generic monitoring stack, InfluxDB is intended as a storage mechanism for all use cases dealing with large amounts of timestamped data, such as application metrics, real-time analytics, and DevOps monitoring [20]. Queries are done through HTTP APIs with a Structured Query Language (SQL)-like query language called "InfluxQL"[18] or a recently developed, more specialised query language called "Flux"[19] [9].

Chronograf is the visualization and administrative configuration tool component of the TICK stack. It is a web application that uses templates and libraries to effortlessly create dashboards with real-time visualizations. One of the most important and notable features is the support for multiple organizations and users through OAuth 2.0[20].

Kapacitor is a native data processing framework that facilitates alert creation, ETL job execution, and anomaly detection. It is designed to process batch and stream data. Streamed data is analyzed and processed in real-time using the TICKscript programming language. It integrates with multiple communication and alerting platforms, such as OpsGenie[21], Alerta[22], and Slack[23].

[18]https://docs.influxdata.com/influxdb/v2.0/query-data/influxql
[19]https://docs.influxdata.com/influxdb/v2.0/query-data/flux
[20]https://oauth.net/2
[21]https://www.atlassian.com/software/opsgenie
[22]https://alerta.io
[23]https://slack.com

Finally, being such a generic tool, the TICK stack doesn't have a special product or plugins that target specifically the **healthcare industry**. However, this is not a particularly relevant drawback since there were already case studies that show how simple and practical it is to use the stack to monitor everything healthcare-related, by using community plugins as identified in the healthcare company Allscripts case study of using the TICK stack to monitor their IT infrastructure [21]. Outside of the healthcare industry, the TICK stack is used by companies such as Trivago, MuleSoft, and Indeed, among others [22].

2.2.3 Nagios

Nagios is an agent-based monitoring framework focused on application, system, service and network monitoring [23]. It offers diverse commercial solutions — Nagios XI[24], Nagios Log Server[25], Nagios Fusion[26], and Nagios Network Analyzer[27] —, all based on the available open-source solution — Nagios Core[28]. Nagios Core, being the free version of the Nagios ecosystem, has a limited set of features for monitoring critical IT infrastructure. Some of the available features are: network monitoring (HTTP, Simple Mail Transfer Protocol (SMTP), etc.), systems resource monitoring, application monitoring, reporting (via a very limited web interface), and alerting (via email, Short Message Service (SMS) or custom scripts) [23], [24].

Nagios' paid plan targets the healthcare industry by having specific monitoring configurations as well as performance indicators, through a custom product version and dedicated support to the organizations of this sector. However, none of these capabilities are included in the core, free version of the software [25]. Outside of the healthcare industry multiple companies, big and small, use mostly the paid version of Nagios' software, e.g. Uber, Twitch, Dropbox [26].

Figure 3 shows Nagios architecture with the referred components and, as mentioned in section 2.1, the pull-based paradigm on which Nagios is based.

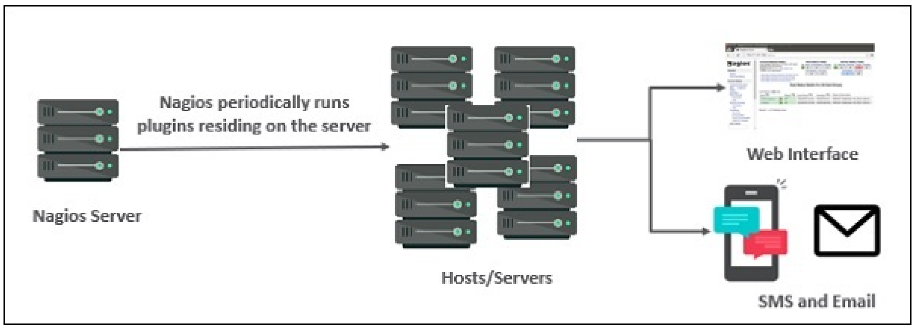


Figure 3: Nagios architecture. Source: [27]

[24]https://www.nagios.com/products/nagios-xi
[25]https://www.nagios.com/products/nagios-log-server
[26]https://www.nagios.com/products/nagios-fusion
[27]https://www.nagios.com/products/nagios-network-analyzer
[28]https://www.nagios.com/products/nagios-core

2.2.4 Prometheus

Prometheus is an open-source, metrics-based monitoring system with an alerting toolkit. It collects and stores metrics as timestamped data in a TSDB. It uses its own query language, "PromQL"[29], which was designed to be flexible enough to handle multi-dimensional data models [28]. Prometheus has a highly capable architecture that consists of several components [29]:

- The main server[30], which collects and stores time-series data.
- Exporters[31] to easily collect metrics from known services and systems.
- Client libraries[32] to develop custom monitoring agents.
- A push-gateway[33] to allow ephemeral batch jobs to submit their metrics to Prometheus.
- A service discovery tool to allow Prometheus to know what to monitor and notice when something that should be monitored is not responding.
- A simple web-based visualization tool, Expression Browser[34], used mainly for ad-hoc queries and debugging.
- An alerting tool[35] for creating, managing, grouping, forwarding, and inhibiting alerts.

Figure 4 shows the Prometheus architecture with the components mentioned above and, as described in section 2.1, the pull-based monitoring architecture on which Prometheus is built. Recently, the push-gateway component has been added to the overall architecture, introducing a more flexible approach and allowing external jobs to report their metrics to Prometheus via this new gateway.

As stated above, Prometheus provides two different possibilities for data collection: Client Libraries and Exporters. Client libraries lead to the production of highly custom metrics, as it allows the insertion of monitoring data collection actions and rules into the application code. On the other hand, as not all executed code is self-owned and self-controlled, Exporters represent an alternative for collecting metrics through communication protocols and interfaces, such as HTTP. Prometheus uses exporters to collect metrics at regular intervals and consumes that information in a standard exposition format called OpenMetrics[36] [9], [29].

[29]https://prometheus.io/docs/prometheus/latest/querying/basics
[30]https://github.com/prometheus/prometheus
[31]https://prometheus.io/docs/instrumenting/exporters
[32]https://prometheus.io/docs/instrumenting/clientlibs
[33]https://github.com/prometheus/pushgateway
[34]https://prometheus.io/docs/visualisation/browser
[35]https://github.com/prometheus/alertmanager
[36]https://github.com/OpenObservability/OpenMetrics

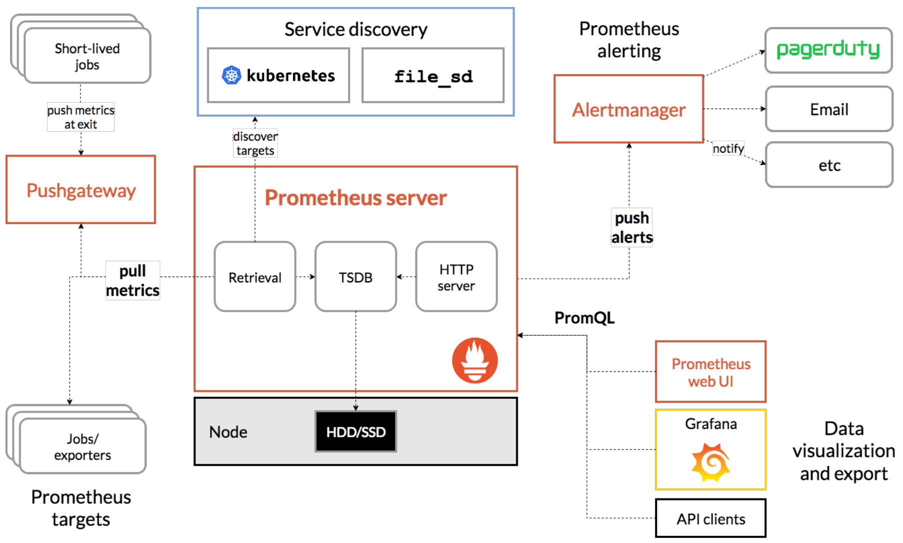


Figure 4: Prometheus architecture. Source: [28]

Prometheus is a very focused ecosystem. As such, its visualization tool is not that powerful and ultimately not suitable for monitoring data visualization. Hence, Prometheus allows integration with Grafana[37], an open-source visualization tool, to create, manage and customize visualization dashboards.

The alerts manager focuses on grouping, classifying, and forwarding alerts via the supported communication channels.

Finally, like TICK stack, Prometheus does not focus particularly on the healthcare industry. Outside the healthcare industry, Prometheus is used by many companies, such as DigitalOcean, Docker, Robinhood, and many others [30].

2.2.5 Monitoring Tools Comparison

While comparing the four monitoring solutions, it will be taken into account its core features — data collection, data storage, visualization, alerting, and data extraction — and aspects such as learning curve and deployment requirements.

Of the four monitoring solutions, Nagios (Core) is the most limited stack since most of Nagios' features are only available in commercial (paid) options. One of the major features accessible exclusively in commercial editions of Nagios is an integrated database, meaning that long-term storage is not available in the Nagios platform's "free tier". On the other hand, Nagios' main server is written in C, which translates into a highly efficient and cost-effective system. Nagios also has an evolved plugin ecosystem and an active community of developers.

ELK is the analyzed monitoring tool that is most different from the others when it comes to its purpose. It was designed as a search engine, providing fast and efficient deep search and log analysis in real-time,

[37]https://grafana.com

while the other solutions focus primarily on monitoring metrics and alerts. It is a completely free and open-source monitoring stack, meaning high market adoption and widespread deployment. As it is not focused on metrics, ELK stack can be expensive and resource-consuming when optimizing for metrics processing and analytics. Despite allowing its users to scale horizontally for free, which is a useful capability in real-life deployments, ELK stack has substantial resource requirements such as the availability of at least 3.5GB of Random-Access Memory (RAM) and the recommended usage of Solid State Drive (SSD). Also, apart from Beans, all components are written in Java, leading to high resource consumption in some usage scenarios. Conforming to all those hardware and performance requirements can be a significant problem in more resource-limited environments [31].

Prometheus and TICK stack are both entirely written in GO. This leads to a straightforward installation and deployment since all that is needed is the static lightweight GO binaries. The fact that these two monitoring solutions are written in GO does not mean they are anything alike, starting with the data collection architecture where Prometheus is pull-based, and the TICK stack is push-based, which completely differentiates the overall architecture of these two tools.

Prometheus benefits from a wide ecosystem with low deployment requirements with multiple data source plugins and extensive community support. On the other hand, it has strong limitations regarding the data that can be stored, and it needs to be integrated with an external visualization tool, such as Grafana, leading to higher installation complexity. Prometheus also has a powerful and efficient query language, PromQL, but it is also a complex language that has a high learning curve, which makes it difficult to use and increases the adoption effort [31].

TICK stack uses columnar storage with high compression rates, providing adequate metric support, and stores different data types natively, such as int, float, text, and boolean. Furthermore, it offers a "SQL-inspired" query language that is simple to get started with, providing a smaller learning curve. To more experienced users, it has a highly efficient processing language, Flux, that is more complicated to learn but offers more flexibility and performance. On the other hand, despite storing and processing event logs, it is less efficient in this use case to the extent that text search may need to be executed with a brute-force full-column scan. As a final note, TICK stack only provides the option for horizontal scaling with a commercial license.

Table 1 shows a structured comparison that, in addition to the aspects already mentioned above, introduces more detailed comparative information about each tool, such as the availability of alerts and data extraction capability.

Table 1: Comparison between monitoring solutions

	ELK Stack	TICK Stack	Nagios	Prometheus
Source	Open-source	Open-source	Mostly open-source	Open-source
Licensing	Free	Mostly free	Free/optional paid[38]	Free
Design/purpose	Search engine and log analytics	Monitoring and alerting	Monitoring and alerting	Monitoring and alerting
Data collection architecture	Push-based	Push-based	Pull-based	Pull-based
Data collection sources	Few/not many[39]	Many. Allows to develop new	Many. Possible, but not easy to develop new	Many. Easy to develop new
Integrated database	ElasticSearch (document-based)	InfluxDB (TSDB)	Paid	Internal TSDB
Alerting	Possible, but not powerful	Possible, with Kapacitor	Possible	Possible, with Alert Manager
Data analytics and visualization	Kibana	Chronograf	Integrated Management UI	Possible to integrate with Grafana
Data extraction	CSV or Log format	Line protocol, via HTTP API or Flux	CSV	PromQL
Deployment requirements	High	Moderate	Moderate/low	Low
Scalability	Clustering	Clustering, paid	Paid[40]	Prometheus Federation[41]
Main development language	Java	Go	C	Go
Unique feature(s)	Highly efficient search engine	Software as a Service (SaaS), cloud deployment	Active community of developers and users	Multiple plugins, wide community support

[38]Only Nagios Core is free.
[39]Beats allows a more comprehensive list of sources, but it is not part of the original stack.
[40]Deployment architecture varies according the chosen commercial solution.
[41]Prometheus Federation is a Prometheus server that can scrape data from other Prometheus servers [32].

2.3 Monolithic vs. Microservices Architectures

Nowadays, the world demands high-speed, efficient, and reliable application and information systems [33]. These high standards can only be met by choosing an appropriate software architecture. The choice of which software architecture to apply has an impact on the entire resulting system, reflecting on its efficiency, scalability, deployability, and other aspects. Hence, the choice of the appropriate software architecture being such an important aspect in the early phases of software design and development.

Although there are multiple options when it comes to software architectures, the most relevant and popular ones are the Monolithic and Microservices architectures.

2.3.1 Monolithic Architecture

Of the two architectures, Monolithic architecture is the most traditional one. Its core approach is to package and deploy software as a single application that includes all components, such as the presentation layer, business logic, authentication and authorization, database access, integrations, and others. Despite allowing internal modularity and responsibility separation, the application is offered as a single package that can be installed and replicated as much as needed.

Figure 5 shows a model example of a monolithic architecture.

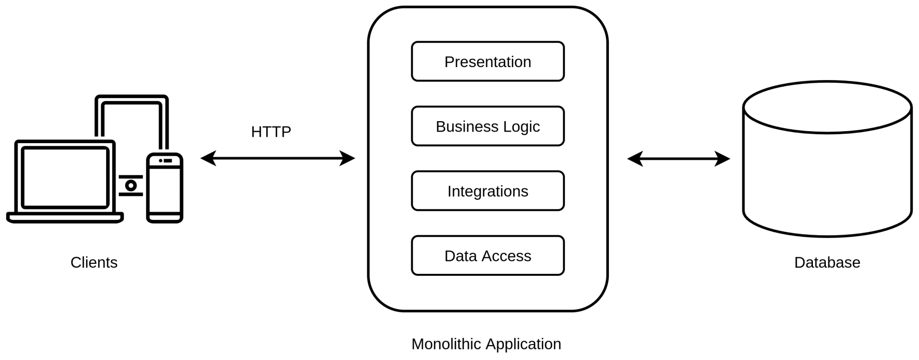


Figure 5: Monolithic architecture. Adapted from: [34]

2.3.2 Microservices Architecture

The microservices architecture consists of structuring the application as a collection of service-oriented APIs — microservices. Each microservice is a small application that encapsulates a core business functionality as well as the underlying implementation [35]. The different microservices that work independently but in a structured and organized way are not usually served directly to the clients. Instead, they are served through an API gateway (or a reverse proxy) responsible for tasks such as load balancing, access control, and caching [36].

Figure 6 shows a model example of a microservices architecture, with five different independent services, the API gateway, the presentation layer, and, finally, the clients.

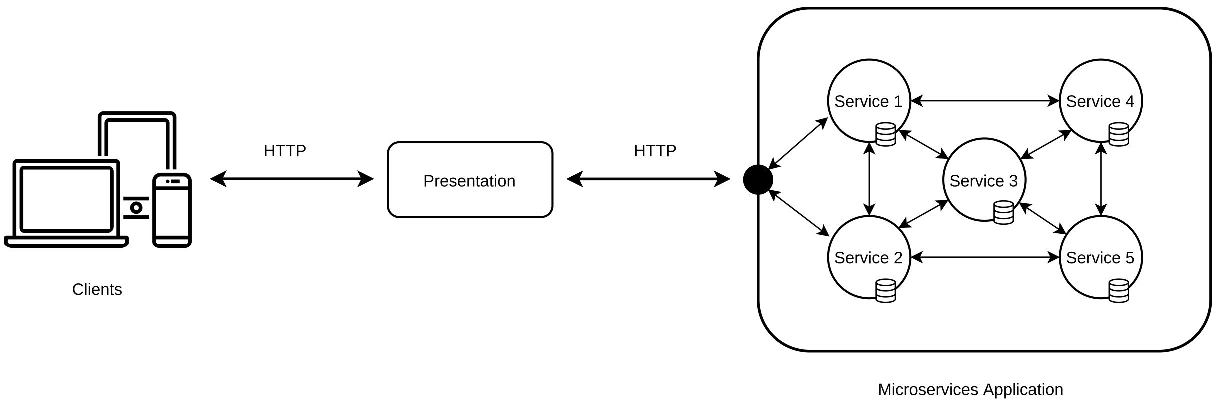


Figure 6: Microservices architecture. Adapted from: [34]

2.3.3 Architectures Comparison

As is comprehensible, each architecture has its advantages and disadvantages, meaning no option can be applied to every scenario or specification. The right architecture depends on the application's specificities and purposes. It is possible, however, to evaluate and compare each approach to make a more informed and thoughtful decision.

A monolithic-based application is simple to develop, test and deploy since it only needs to install the packaged application on the desired host. It is also simple to scale horizontally by launching multiple instances of the application on the available infrastructure and organizing them under a load balancer [36]. This simplicity, although desirable, has a cost: flexibility. Monolithic applications need to be redeployed on each update difficulting continuous deployment and availability. Over time this problem intensifies because as these applications evolve, they usually grow in size, which affects the startup time. Moreover, monolithic applications have reliability issues because one fault can bring down the entire application and disrupt all provided functionalities [33]. Finally, it was also found that even with multiple instances, monolithic applications can experience performance bottlenecks with heavy usage and extreme workloads [37].

On the other hand, the microservices approach is easier to maintain since each service constricts a core functionality that is the responsibility of a team of developers. When it comes to scalability, microservices are just as easy to scale as a monolithic application but much more reliable because the failure of one microservice does not affect the whole system. Microservices also present availability advantages over monolithic applications since each microservice can be redeployed on its own and requires little downtime for that service [33]. One thing where microservices applications are undeniably different is the overall complexity of the resulting system. Microservices applications are, in its essence, distributed systems, meaning there are a group of issues, mainly communication-related, that need to be addressed. These issues and concerns, typical of a distributed system, make this kind of architecture inherently more complex than the monolithic counterpart. Microservices applications are also more complex to deploy since each microservice has multiple instances, and each instance needs to be configured, scaled, and monitored [36]. Finally, there is also a conceptual detail that can lead to a potential complexity problem —

the granularity of each microservice. If not properly designed, a microservices application can have too many microservices because the granularity by which the system functionality was divided was too fine, making the system more complex than it could and should be. This particular problem is common when transitioning from a monolithic architecture to a microservices one [38].

2.4 Web (API) Security

As stated in section 2.2, IT monitoring tools usually have different components that communicate through HTTP APIs. This communication must be done in a secure way, especially communication that originates from the end-user, i.e., all actions done through web interfaces and dashboards [39].

There are widely-known best practices and common ways to strengthen the security of an API, such as access tokens, encryption, signatures, quotas, throttling and the use of an API gateway. Access tokens are a mechanism for controlling access to services and resources by assigning tokens to trusted entities, with specific access permissions, within the system. Encryption and signatures are used in protocols such as Transport Layer Security (TLS) to increase API security by providing data integrity and data secrecy and also ensuring only the intended recipients can see the private information being transmitted. Placing quotas on how often a client can make requests to an API and creating throttling rules are effective ways to protect APIs from usage spikes and Denial-of-Service (DoS) attacks. API gateways authenticate traffic and analyse how the APIs are used [40].

Since security is not the main focus of this Dissertation, the most focused security topic, both during research and development, was user authentication. The most commonly used web authentication mechanisms are **basic authentication** and **token-based authentication**.

Basic authentication is the most straightforward authentication mechanism and consists of sending the user's credentials in every API call. Despite being a simple approach, it has some considerable drawbacks when applied in modern web applications and systems. Since each API call must include the user's credentials, there is a considerable chance of accidentally exposing those credentials in one of the API calls because credentials are sent in plain text. Moreover, this approach has significant performance costs as password validation is an expensive process that would be executed in every API call, thus seriously limiting API throughput.

Token-based authentication solves the problems enunciated above, albeit being a more complex mechanism than basic authentication. In token-based authentication, the user's credentials are provided once to a dedicated login endpoint which issues a limited-time access token that is given to the client and used in place of its credentials in subsequent API calls until the token expires. There are different types of token-based authentication, the most recognised of which being **traditional session cookies**, **modern token-based authentication** (without session cookies), and **self-contained tokens** (JWTs[42]) [42].

[42]JWT is an open standard (RFC 7519) for compact and self-contained security tokens. A JWT consists of a set of claims related to a specific user in the format of a JSON object, with a format descriptor as the header. JWT's information can be trusted and verified against tampering because it is digitally signed [41].

Session cookies are the most traditional and simplistic implementation of token-based authentication. It is an adequate implementation for browser-based clients hosted on the same physical site of the API. After the user provides the authentication credentials, the login endpoint returns a `Set-Cookie` header on the response instructing the web browser to store the session token in the cookie storage. Subsequent requests to the same domain will include the token as a `Cookie` header. To validate session cookie requests, the server can look up the cookie token in the cookie data store, typically a database, to see which user, if any, is associated with that token [42].

As technology evolved, new computing devices appeared, particularly mobile devices and native mobile applications. Though cookies work great for web browser clients, they are less natural for native applications as the client needs to explicitly manage them. Thus, the need to find a more broad and flexible implementation of token-based authentication increased. It is in this context that modern token-based authentication appears. This implementation follows a similar approach to session cookies since both clients and servers still need to store the generated token but with some major differences. In this approach, the storage method is independent of session modules meaning the tokens are going to be stored on mechanisms such as local storage on the client-side and databases (SQL or Not only SQL (NoSQL)) on the server-side. After the token is issued, the client establishes the authentication using the standard Bearer authentication scheme for HTTP. The major improvement over session cookies is the possibility to use this authentication mechanism not only on browser clients but also on mobile applications, and Business-to-Business (B2B) communications [42].

Although the two previous solutions are adequate, each in its context, both suffer from the same problem: the need to store the access tokens and their state on the server-side. When the user base increases, the server-side token databases start to be a source of performance bottlenecks. Self-contained tokens, also known as stateless tokens, solve this problem by removing the need to have a token database. Thus, becoming quite popular since the standardization of JWTs, their most popular implementation. The idea behind this implementation is based on encoding the token state directly into the token and sending it to the client rather than storing it in the database. Even though not having a token data store is a big benefit in terms of scalability, it has a downside: token revocation. If there is no token database, there will be no database from which to delete the revoked or invalid token. There is, however, a solution to this concern since it is possible to revoke stateless JWTs by keeping an allow-list or block-list of tokens in a database [42]. Of course, this reintroduces the problem of keeping server-side state for token management; however, this state is still much simpler to maintain than the previous alternatives.

2.5 Discussion

This chapter has addressed various issues related to infrastructure monitoring. As a first conclusion, it can be said that IT monitoring solutions are a complex and broad but interesting topic. Its relevance becomes even more evident in healthcare when considering how critical the IT services used in this context are, both in terms of availability and performance. The positive impact is undeniable when it comes to managing and, more importantly, preventing disasters or critical outage situations.

Looking at all the solutions and approaches presented, especially those related to data collection (pull-based and push-based) and software architectures (monolithic and microservices), we can draw some conclusions. The first is that none of these architectures is perfect, nor the only possible answer to every problem. Nevertheless, some of these options are more flexible and open to future changes and modifications. For example, since push-based architectures do not take a centralized approach to metrics collection, i.e., who manages the monitoring process and sends the respective metrics are the clients, creating a more decoupled and distributed system instead of a centralized one. However, pull-based architectures offer a more manageable design that provides good performance and low complexity, which is suitable for a monitoring solution that is more focused on specific metrics and targets, while push-based architectures are more suitable for generic-purpose monitoring solutions without specific targets or metrics. For software architectures, the same dilemma arises: a more centralized system or a distributed system. And the arguments are much the same. A monolithic architecture allows for a less complex and more coherent approach while making it more difficult to make structural changes or introduce disruptive functionality in some internal modules. On the other hand, microservices architectures are more complex and modular systems that thrive in the most difficult scenarios for monoliths, although sometimes they are too complex for the requirements and goals at hand. As such, the choice of an architecture, be it software, data collection, or another nature is always a complex decision that has to consider multiple factors. These factors include the inherent complexity of the system so that the chosen architecture does not introduce unnecessary complexity to the solution, the maintainability, deployability, and replicability of each component, and the performance that the solution needs and each architecture allow to achieve, among other factors. A better architectural decision is based on the combination and consideration of these various factors, prioritizing the most relevant ones for the specific solution and its usage and application goals. In the specific case of this Dissertation, the most important factors to consider when choosing a software architecture and a data collection architecture are solution performance, reliability, deployability, and scalability, as will be made clearer in the next chapter.

The analyzed solutions were mostly metric-oriented, with the ELK stack being the exception. For this work, this is a positive aspect, as most monitoring events in healthcare involve metrics rather than free text content. However, none of the solutions simultaneously supported multi-organization and multi-site schemas and full API-based monitoring management. For example, the TICK stack supports multi-organization and multi-site schemas but enforces file-based configuration for monitoring agents and does not allow this type of configuration or management via APIs. Prometheus, on the other hand, allows API-based monitoring management and configuration but does not provide full support for a system with multiple organizations and sites. This limitation of the currently available solutions is one of the main reasons for developing a new system instead of using the previously mentioned ones.

Considering these conclusions, this work will focus on the implementation of a full-fledged solution that builds on the work and solutions studied but also addresses the problems and shortcomings mentioned above. Nevertheless, since these systems are mostly composed of different modules that are relatively independent of each other, if it is possible to integrate some of the components from these solutions, it is advisable to do just that instead of re-implementing already developed and widely tested and used components.

The knowledge and insights gained from studying existing solutions can and will be applied to the design of the IT monitoring solution being developed in this Dissertation. The system should be modular and separate the concerns of each module, namely the TSDB, the metrics receiving and parsing component, the data exporter, the alerting tool, and the authentication server.

2.6 Summary

In this chapter, it was confirmed that IT infrastructure monitoring solutions are an important part of any IT infrastructure, especially in healthcare. First, the different architectures for data collection are explained, followed by a description of existing commercial and open-source solutions, as well as related work and documentation on this topic. Then, an analysis on two other topics is presented: Software Architectures, with a distinction and comparison between monolithic architecture and microservices architecture, and Web Security, focusing mainly on authentication. Finally, all the previous topics are discussed and it is noted that all the solutions, although solid and widely used, have their weaknesses and none of them supports all the features required for the scope of this Dissertation. The chapter concludes with a proposal for implementing a solution that addresses these missing features and shortcomings.

Chapter 3

Research Methodology and Technologies

This chapter describes the methodologies and technologies used throughout the research development and validation of this Dissertation. First, it is introduced the applied research methodology – Design Science Research Methodology (DSRM). Next, it is explained the use of questionnaires in the research and validation phases of the Dissertation. And, finally, all the chosen technologies are presented, as well as the respective selection process and justification.

3.1 Design Science Research Methodology

This Dissertation arose from the identification of the insufficiency of the existing monitoring solutions when applied to the healthcare environment and requires the development of a final artifact that should be later tested and evaluated in a real environment. As such, considering this is the exact execution path of the DSRM, this methodology was considered the most appropriate for the development of this Dissertation's work.

As can be seen in figure 7, the DSRM consists of six different phases: **problem and motivation identification**, **goals definition**, **design and development**, **demonstration**, **evaluation** and, finally, **communication** [43].

The first phase focuses on defining the issues and concerns that need to be addressed and explaining the importance and merit of developing a solution. Justifying the value of a solution has significant advantages, such as motivating the researcher and target audience to develop the solution and accept consequent results and expediting the researcher's understanding of the problem [43].

In the second phase, the solution's objectives are inferred from the problems previously characterized, taking into account what is feasible. The objectives can be both quantitative (e.g. description of how the solution to be developed would be superior to the existing ones) and qualitative (e.g. how the developed solution can support new components to address problems that were not originally identified) [43].

The third phase encapsulates the artifact creation, which includes the specification of the desired functionality and architecture, followed by the actual implementation [43].

The fourth phase consists of demonstrating the produced artifact as a solution to the identified problems. This demonstration can take many forms, such as scientific experiments, simulations, and case studies, to name a few [43].

The fifth phase serves to evaluate the extent to which the artifact solves the problems found. This evaluation is accomplished by comparing the previously defined goals of a solution to the obtained results from the developed artefact [43].

The sixth and final phase is to communicate the problem and its importance, as well as the artifact and its usefulness. This communication is carried out through research and professional publications [43].

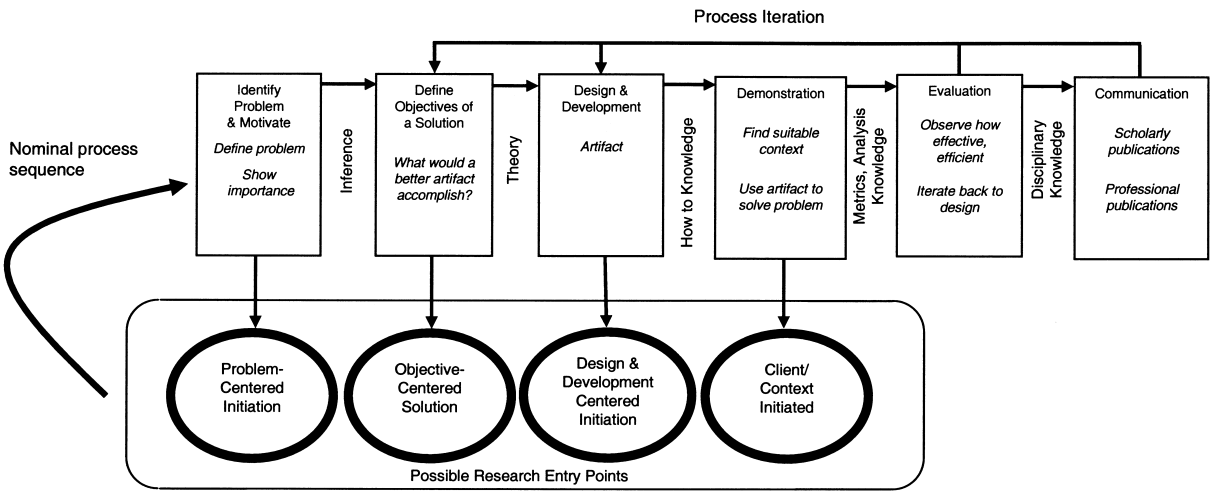

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 7: DSRM process model. Source: [43]

Figure 7 also shows that the DSRM can be applied in multiple ways, depending on how the process is initiated. If the methodology is applied with a problem-centered approach, then the process initiates in phase one. If the objectives are the focus of the research, then the process initiates in the second phase. If the approach focuses on the solution design and implementation, the process initiates in the third phase. Finally, if the approach is based on analyzing a practical solution that already exists, then the process initiates in the fourth phase.

As already mentioned, this Dissertation started with the identification of a problem and the lack of functionalities in the existing monitoring solutions. Thereby, it is logical to conclude that the DSRM will be applied with a problem-centered approach, therefore executing the six phases of the methodology.

3.2 Questionnaires

Questionnaires are a type of research instrument that consists of a series of questions designed to collect information from respondents via face-to-face, telephone, computer, online forms, or the mail.

Questionnaires are commonly used to assess the behavior, attitudes, preferences, views and intentions of a large number of people. However, when conducted in person or through a recognizable protocol of communication, there is a typical problem with respondents lying owing to social desirability, as most individuals want to portray a favorable picture of themselves and would lie or bend the truth to make themselves appear good. This difficulty is frequently solved by conducting anonymous questionnaires in which the responders cannot be identified [44].

In this Dissertation, questionnaires were carried out via anonymous online forms as a means to collect useful information from possible users and stakeholders regarding important topics, both from the point of view of modeling and design, as well as validation of results.

During the development of this Dissertation, two different questionnaires were carried out. One distributed at an early stage of the requirements gathering and system modeling process, to better understand the problem and understand the needs of professionals and teams that identified with it. The other to assess whether the developed system served to solve or, at least, fill the identified problem, as well as the associated needs.

3.3 Implementation Technologies

The technological domain of each project varies greatly depending on its purpose and usage. In this work, it is necessary to evaluate and choose technologies regarding two different layers: **business logic**, and **data**. Thereby, it is first discussed the chosen technologies regarding the web services and server (business logic), followed by a discussion regarding what technologies were chosen to store and retrieve data in the scope of this work.

3.3.1 Authentication

To guarantee that the system is only accessed and used by the intended users that have proper credentials, the system needs an authentication module. As analyzed in section 2.4, token-based authentication, namely using self-contained tokens (JWT) is the current standard and one of the most effective ways to accomplish web services authentication. As such, this is the technology that is going to be used and implemented to generate and validate authentication credentials.

3.3.2 Web Servers and Services

To expose the solution functionalities through HTTP requests is necessary to encapsulate these in web servers and services. As such, it was decided to use **JavaScript** as this work's main programming language. JavaScript is a prominent programming language in the web context. This prominence is due to various factors. The most important ones are an ongoing attempt to make JavaScript, and all its frameworks, the standard tool for full-stack web development and its technological capabilities, such as object-oriented and event-based functionalities [45].

When it comes to backend technology, the core focus of this work, JavaScript has a plethora of tools and frameworks available, being the combination of **Node.js**[1] and **Express.js**[2] the most recognized approach.

Node.js is a powerful open-source backend tool that allows the development of JavaScript-based asynchronous HTTP web servers. Formally, Node.js is an asynchronous event-driven JavaScript runtime designed to build scalable network applications [46]. Its lightweight and asynchronous nature is particularly important and advantageous when assembling a microservices architecture [47].

Express.js is an unopinionated and low-scope web framework, meaning that it is very flexible regarding the project structure and internal module architecture. It allows the creation of all sorts of web artifacts, such as web applications and REST APIs, and inserts little computational overhead because it deeply integrates with Node.js APIs, which results in highly performant applications [47].

Thus, at the software architecture level, it was chosen to use this combination of Node.js and Express.js based on its asynchronous and event-based nature, allowing it to handle multiple requests at once, without system blocking. Other factors that were taken into consideration were the high performance and scalability that this combination of technologies enables.

Finally, there was a last tool applied together with the ones already mentioned — **TypeScript**[3]. TypeScript is an extension of JavaScript and offers useful additions to the stack, such as type safety, reducing the number of runtime errors and bugs associated with mixing data and variables of different data types; a larger set of Object-Oriented Programming (OOP) features, such as interfaces, generics, inheritance, and method access modifiers [48]. These extra functionalities and capabilities were the main reasons to adopt this technology that ultimately will help develop a more reliable and resilient solution.

3.3.3 Databases

To store all generated data, this work is going to need two databases: one to store the system's configurations (monitoring agents, alerts, etc) and the other to store the actual monitoring metrics. Therefore, to store the system's configuration, it is useful to use a document-based database since even in the same configuration category (e.g. monitoring agents), the fields and information to be stored can be quite varied and dynamic. To store the actual monitoring metrics, it is necessary a TSDB as previously seen during the state-of-the-art research documented in section 2.2.

MongoDB[4] is a NoSQL, document-based database that uses a JSON-based format to store data. It has a scale-out architecture, meaning it is designed to scale horizontally and has high query performance, making it a good choice to store the system configurations [49]. Its dynamic JSON schema was also a very

[1]`https://nodejs.org`
[2]`https://expressjs.com`
[3]`https://www.typescriptlang.org`
[4]`https://www.mongodb.com`

important factor when deciding what database to use since the entities that are going to be stored, such as monitoring agents and notification channels configurations, have very different fields depending on their internal type (e.g. a database monitoring agent has different configuration fields than a ping monitoring agent).

Regarding the TSDB, it was decided to use **InfluxDB**. During the state-of-the-art research and comparison of monitoring solutions, it was possible to identify three potential candidates: InfluxDB, ElasticSearch, and the Prometheus internal TSDB. The first one to be ruled out was the Prometheus TSDB since it was necessary to use all the tools in Prometheus' stack to use TSDB, which did not satisfy the requirements of this work, as explained in section 2.5. After that, it was a matter of choosing between InfluxDB and ElasticSearch. ElasticSearch is a document-based database that allows to index documents using timestamps. However, as already discussed in section 2.5, it is more focused and optimized for logs and full-text entries. Since this work is more focused on monitoring metrics, as opposed to logs, and InfluxDB is, at its core, a TSDB focused on metrics, it was decided to use InfluxDB instead of ElasticSearch. It was also taken into account that InfluxDB's companion stack is mostly API-based and can be used as a starting point to the implementation of some components of this work, which was considered a positive surplus of the decision.

3.3.4 API Gateway

To encapsulate and secure all the solution's infrastructure and services, it is needed an API gateway. As stated in sections 2.3.2 and 2.4, API gateways are useful when there is the need to authenticate traffic, distribute effort load (via load balancing) while taking advantage of caching and usage analytics, which were the main reasons to apply this pattern in the solution to be implemented. As such, the following API gateway technologies, which are current leaders in this field [43], were analysed: **NGINX**[5], **Tyk**[6] and **Kong**[7].

The first to be dropped was NGINX due to only making its API gateway product available in the paid version (NGINX Plus), which was an insurmountable setback for not being an open-source option nor without associated costs.

The second to be discarded was Tyk because despite making available many of the most used features in the open-source version, it only supports scalable and fault-tolerant multi-tenant deployments in the paid licensing version.

Finally, Kong was selected as the tool to implement the API gateway. In addition to supporting all the most used features, as well as redundancy and fault tolerance in the open-source version, it also allows all its configuration to be done declaratively through a JSON or Yet Another Markup Language (YAML) specification.

[5]https://www.nginx.com/products/nginx/api-gateway
[6]https://tyk.io
[7]https://konghq.com/kong

3.4 Summary

This chapter presented and discussed the selected methodology, solution design and architecture, and the underlying technology stack of this Dissertation. First, there is an analysis of the selected research methodology, **DSRM**, and the approach of its application, by using a problem-based strategy. Then, it is described what is the use of questionnaires in research processes and explained how they were particularly used in the context of this Dissertation's research and validation. Finally, there is a discussion and justification of the selected technologies, which were **Node.js**, **Express.js** and **TypeScript** for the web servers, **MongoDB** and **InfluxDB** for the databases, and **Kong** for the API gateway.

Chapter 4

Proposal

This chapter presents the solution proposal, as well as the context and details of the solution's modeling and architectural design. First, there is an explanation of the technical questionnaire execution and its results. Then, are presented the solution's requirements, both functional and non-functional, followed by an overview of the solution's domain model. Finally, the solution architecture is presented, including both high-level and low-level architectural views as well as the solution's global deployment architecture. The chapter is concluded with a discussion of the proposed architectural design in light of the specified requirements.

4.1 Technical Questionnaire

The first step to better understand the problem and its associated needs was to undertake an anonymous online survey, the questions of which are available in appendix A. This questionnaire was distributed in a Google form and was directed to system administrators of Portuguese hospitals in the northern and central regions, namely the CHTS, in Penafiel, and the USLGuarda, with a final result of 18 answers. The questionnaire is divided into two sections: one with closed questions on the respondent's history, context, and perspective to gain a more objective understanding of their ideas, and the other with open questions to gather requirements and functionality suggestions.

The questionnaires were treated through organization and data interpretation tools, namely Google Sheets[1]. The data from the closed questions were treated as being of the nominal type, allowing a statistical representation of the data, followed by the respective analysis and interpretation. On the other hand, the answers to the open questions were treated individually, manually extracting the topics and suggestions

[1]https://www.google.com/sheets/about

considered most important and relevant for the development of the Dissertation, given the context of its execution.

The first important figure noticed was that most IT administrators, roughly 94% of the respondents, already use some type of monitoring system or solution in their organizations, regardless of its sophistication or utility, corroborating that the study of monitoring systems and its applicability in the healthcare environment is an important and relevant topic. The distribution between respondents that use and don't use a monitoring solution in their organizations is available in the figure 8.

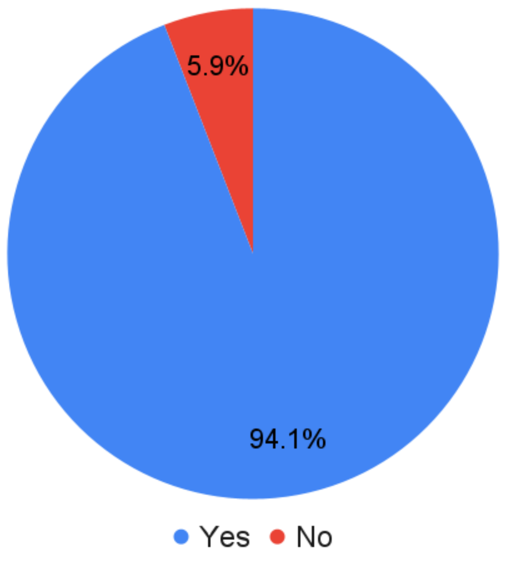


Figure 8: Distribution of IT administrators who use a monitoring solution in their organization

The next step in analyzing the responses was to understand how this work's assumptions were effectively valid for the target audience. This analysis was carried out by studying the statistics associated with the answers to the six statements of the second closed question, which aimed to understand the respondents' perception regarding the use of monitoring solutions and their impact on the management and availability of the systems. The statements present in the questionnaire are as follows (further detail on appendix A):

1. Currently, whether there is a failure or not, you can observe and control the state of your system.
2. When there is a failure in the system, that failure is noticeable.
3. When a failure in a system is detected, you can determine what the implications are and what other services are impacted.
4. A monitoring system is important for systems management.
5. A good monitoring system makes it possible to improve the availability of systems.

6. A monitoring system capable of analyzing and displaying, in an integrated way, the status of different targets and organizations, streamlines the systems management process.

The mean and median values of the answers to each statement of the second closed question were calculated, and it is possible to see a representation of this analysis in the figure 9. These values represent the degree of agreement (or not) with the presented statements that are directly mappable on the Likert scale presented in table 2, which is the same as the one presented in the questionnaire.

The interpretation of these results led to the conclusion that it is a common opinion on the part of IT administrators that monitoring solutions are, in fact, necessary and useful for the management and availability of systems. Furthermore, and with a particular focus on the last statement, it was also possible to perceive that there is a high level of agreement for the need to have a system that is capable of analyzing and displaying, in an integrated platform, different monitoring targets and organizations to simplify and streamline the systems' management process.

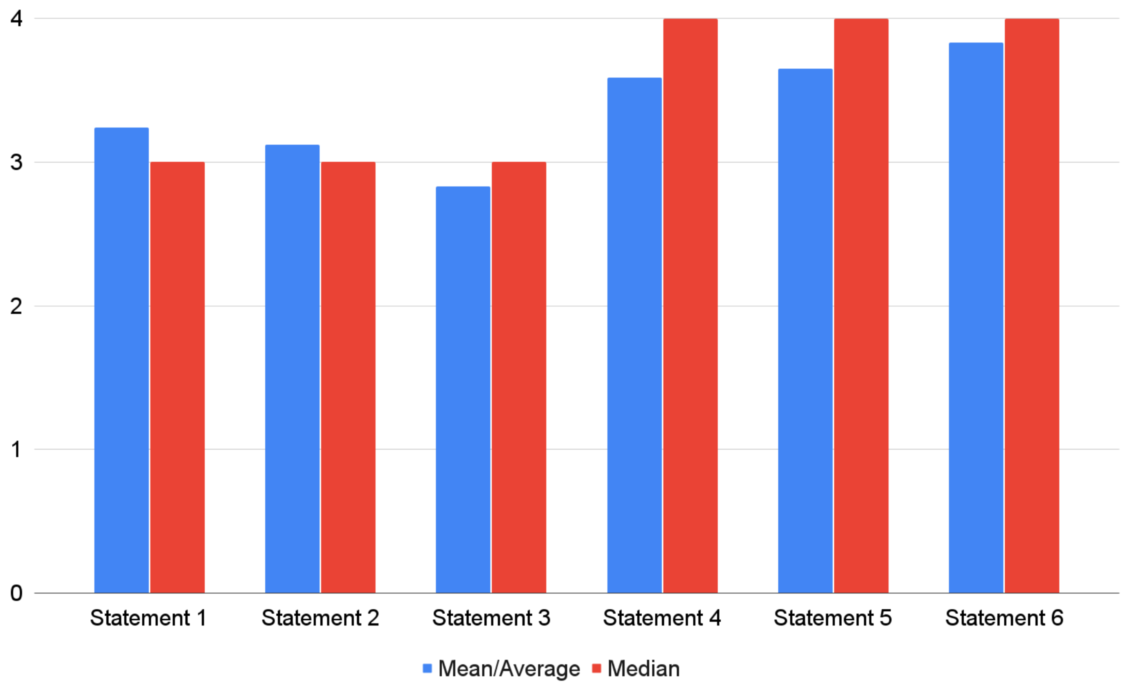


Figure 9: Distribution of responses to the statements of the questionnaire's second closed question (see appendix A)

Table 2: Likert scale used in the statements' analysis. Adapted from [50]

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Numeric Value	0	1	2	3	4

Finally, the answers to the open questions were analyzed individually. These, once again, continued the trend of agreement with what had been defined as the problem and a possible solution similar to what is being proposed.

All this, meaning, both the answers to the closed and open questions and the respective trends and conclusions, strengthen and corroborate the motivations, premises, and objectives of this work.

4.2 Requirements

After understanding that the problem was real and the respective assumptions and objectives were indeed aligned with the stakeholders' needs, we moved on to thoroughly defining the requirements this solution needs to fulfill. This Dissertation aims to build a full-fledged and fully integrated backend monitoring solution, focused on meeting the needs of the healthcare environment, but comprehensive enough to support a wide range of use cases and scenarios. Thus, the following requirements were divided into two logical sets: functional and non-functional, considering their nature and purpose in the system.

4.2.1 Functional Requirements

Functional requirements focus more deeply on the capabilities the system needs to offer to the end-user or B2B client, as well as the respective modeling processes. Hence, this section presents the core functional requirements for the backend monitoring solution.

Data collection

Data collection is a cornerstone functionality when it comes to IT monitoring. As referenced in section 2.1, there are two distinct approaches to data collection agents: pull-based and push-based and, as was also seen in that same section, a hybrid approach where the solution supports the two data collection paradigms has been the most common harmonization approach. Therefore, the proposed backend IT monitoring solution must support the two most common types of data collection and monitoring agents as possible metric data sources.

Furthermore, as far as pull-based monitoring agents are concerned, the solution must allow the full creation and management of metric collecting agents of various types and targets, whether more generic, such as database monitoring, health checks through ping, HTTP requests, or hardware-focused, such as disk and network information. Monitoring agents types that were found to be core needs of the healthcare environments and organizations where the solution was deployed and validated are as follows:

- HTTP Endpoint: Since so many services and applications are exposed through HTTP APIs and endpoints, it is important to be able to monitor the availability and performance of these endpoints.

- Ping: Among an infrastructure of servers and services, it is important to be able to monitor the availability of these services and servers. A ping mechanism is a simple and effective way to do this.

- Databases: Efficient and available data storing and access is paramount in every information system. Therefore, it is important to be able to monitor the availability and performance of the databases that are part of the systems' infrastructure. The databases, SQL or NoSQL, that must be supported by the solution are:

  - MySQL
  - PostgreSQL
  - OracleSQL
  - Redis
  - MongoDB

- NGINX: Many web services and applications use this piece of software as their web serving layer, since it is a popular and widely used web server, load balancer and reverse proxy. As such, it is important to be able to monitor it and guarantee its well functioning within every service or application using it.

Although the list of monitoring agents' types is representative of the implementation prioritization in this Dissertation's work, it is important to note that, in the future, more monitoring agent types can and will be added.

Finally, it must be possible to specify the various status or health levels of the respective agents throughout the creation process, i.e., provide the criteria and logic that govern whether a particular agent is in an OK, critical, or warning condition. Given that the most worrying condition for each monitored service is the critical one, which must be resolved as soon as possible, the critical state conditions must be defined at the creation of any monitoring agent.

Data storage

Since the goal of this Dissertation is to propose an architecture that can handle multi-site and multi-organization IT monitoring, it is necessary to specify how data should be handled and stored. Thus, the proposed solution needs to be able to collect all metrics from organizations or healthcare facilities and store them taking into account the security, independence, separation, and isolation of data from each of the source organizations. The storage mechanism must also take into account that all monitoring data should be accessible inside and outside the organization and healthcare facility, without forgetting, of course, that only authorized users can do so.

Data extraction/export

All the collected metrics and data are important themselves but can only be truly useful if it is possible to access, analyze and interpret them. As such, the monitoring backend solution must allow extraction and export of all the generated data in a standard format, or formats, to allow data consumption via external solutions, such as web applications and visualization engines or dashboards.

Alerting

Alerting is particularly useful when systems fail or go into bottlenecks. Thus, the solution must be able to create, manage and send alerts of different nature, whether via email, Slack, Discord, and HTTP endpoints that allow integration with third-party services for the most diverse uses (alerting, status visualization, export of information, and notification of external services, among others).

Alerts will be sent whenever the status of the monitored service changes to a more problematic condition according to the health levels defined in the monitoring agent configuration, that is, the sending of alerts associated with a given agent will be triggered when it transitions from:

- an OK status to a warning one;
- an OK status to a critical one;
- a warning status to a critical one.

Configuration and customization

All the aforementioned requirements focus on monitoring and data consumption functionality. However, when configuring monitoring agents, health level definitions, and alerting rules, these processes quickly become repetitive, exhausting, and time-consuming. As a result, the system must support high-level configuration and customization capabilities where standard entities, such as notification channels, i.e., commonly used mailing lists or sets of mails, slack or discord webhooks, or HTTP services, can be defined and used as templates when creating new alerts by simply selecting these previously created settings and configurations.

4.2.2 Non-Functional Requirements

Non-functional requirements specify the quality attribute of a software system by judging the system based on a set of standards that are critical to the system's success [51]. The most prominent non-functional requirements for the system are:

- **Information security**: Since the architecture must support multiple organizations and deployment sites, it is important to ensure that only allowed users access the respective monitoring information, that is, user authentication and data isolation;

- **Reliability**: Considering the periodic execution of monitor agents (to collect data), it is important to ensure that each execution is configured, started, and concluded properly. When a critical error occurs during a data collection execution or any other task, the system must be able to handle the error and successfully recover from it;
- **Scalability**: A monitoring solution deals with a multitude of different monitoring agents and large volumes of data. As such, this solution must correctly handle a substantial number of active monitoring agents, as well as vast amounts of data, while keeping good performance and short response times;
- **Deployability**: Since this solution aims to be deployed in various healthcare facilities, the deployment must be a straightforward, low-risk, self-contained and automated process.

4.3 Domain Model

Once the requirements were identified, it was useful to create a high-level representation of the system's core logical entities and how they connect, using a domain model diagram. By definition, a domain model diagram aims to provide a visual representation of the problem context through the definition and relationship between the various conceptual classes, that is, key entities. This visual representation is useful to help understand and more clearly specify the system's core logic and business rules, as well as how it is built [52]. Figure 10 presents the referred solution's domain model diagram.

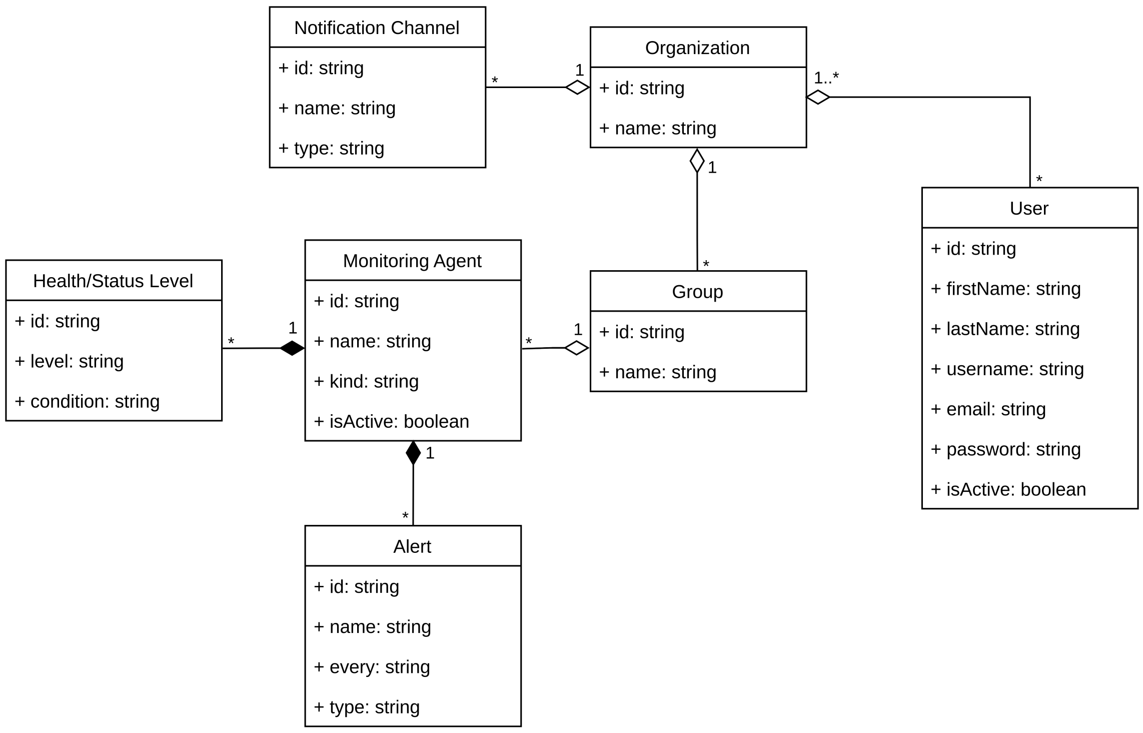


Figure 10: Domain model diagram of the proposed monitoring solution

The top-level entity, as figure 10 suggests, is the "Organization" which represents each healthcare unit or facility. Each organization can be accessed by multiple users and each user can access various organizations, each of which must be associated with at least one organization.

Moreover, each organization has many groups, each with its own set of monitoring agents. Using groups as an aggregate entity, monitoring agents may be grouped or classified. Each organization can also have multiple notification channels, which are organization-wide templates that can subsequently be used as a full copy, i.e., a replication, on the alerts' setup.

Finally, each "Monitoring Agent" is composed of various alert settings, with each alert belonging to a specific monitoring agent, as well as multiple health/state level settings, e.g. "ok", "critical", and "warning" status.

4.4 Architecture

To satisfy the specified requirements and business logic represented in the domain model diagram, it was designed a high-level architecture that can be seen in figure 11, comprising two different servers: the **local server**, which will be deployed in each healthcare facility and is responsible for all monitoring operations and tasks of the facility, and the **guardian server**, which is deployed in an off-site location and provides aggregated monitoring information and management regarding each and all healthcare facilities. Each local server communicates its monitoring information to the guardian server over HTTP requests.

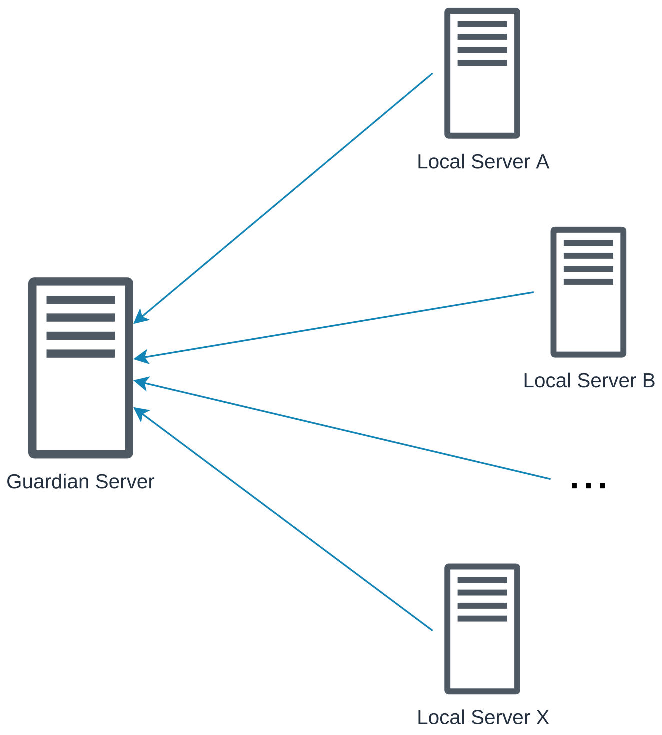


Figure 11: High-level architecture of the proposed monitoring solution

Each server, i.e., the local server and the guardian server, encloses several components, modules and/or storage mechanisms that satisfy the requirements proposed in section 4.2, which are succinctly described in the subsections below. Moreover, since the solution as a whole has a considerable dimension and each module has a very specific responsibility, it is important to ensure that each service runs and scales independently. As such, it was decided that both the local server and the guardian server should be designed following a microservices approach, i.e., each server being composed of small, self-contained, and autonomous units.

Thus, and taking into account the advantages of microservices architecture already mentioned in section 2.3, this architecture design is the most suitable for the work of this Dissertation.

4.4.1 Guardian Server

The guardian server has four main services: **authentication service**, **core service**, **alert service**, and **data export service**. All these services are virtually aggregated, safeguarded and carefully exposed to outside communication through an API gateway, which enables load balancing, if necessary, protection from attacks since the actual services' Internet Protocol (IP) addresses are never revealed, caching and Secure Sockets Layer (SSL) encryption [53], [54] and credential validation (authentication and authorization).

The **authentication service** is responsible for user authentication and authorization. It is also responsible for the creation and management of users. This service is used by the API gateway to authenticate requests based on the provided credentials.

The **core service** contains all the logic related to the creation and management of organizations, groups and monitoring agents. It is also responsible for the creation and management of the monitoring agents' health/status level settings.

The **alert service** handles the creation and management of the notification channels and the alerts' configurations. It is also responsible for the scheduling and execution of the tasks associated with the alerts' events, that is, understanding when and to whom an alert must be sent and sending it.

The **data export service** exposes all the monitoring information in standard formats so each client, whatever it may be, can interpret the data. It also provides information on the current status, as well as the status history, of each monitoring agent, taking into account the registered metrics and their health/status level settings.

To support the four services, and as detailed in section 3.3.3, there are two databases: a TSDB, InfluxDB, to store and consume the monitoring information, and a document-based database, MongoDB, to store and manage all the remaining information (users, organizations, groups, monitoring agents, health/status levels, notification channels and alerts).

Figure 12 presents a diagram that illustrates the described architecture.

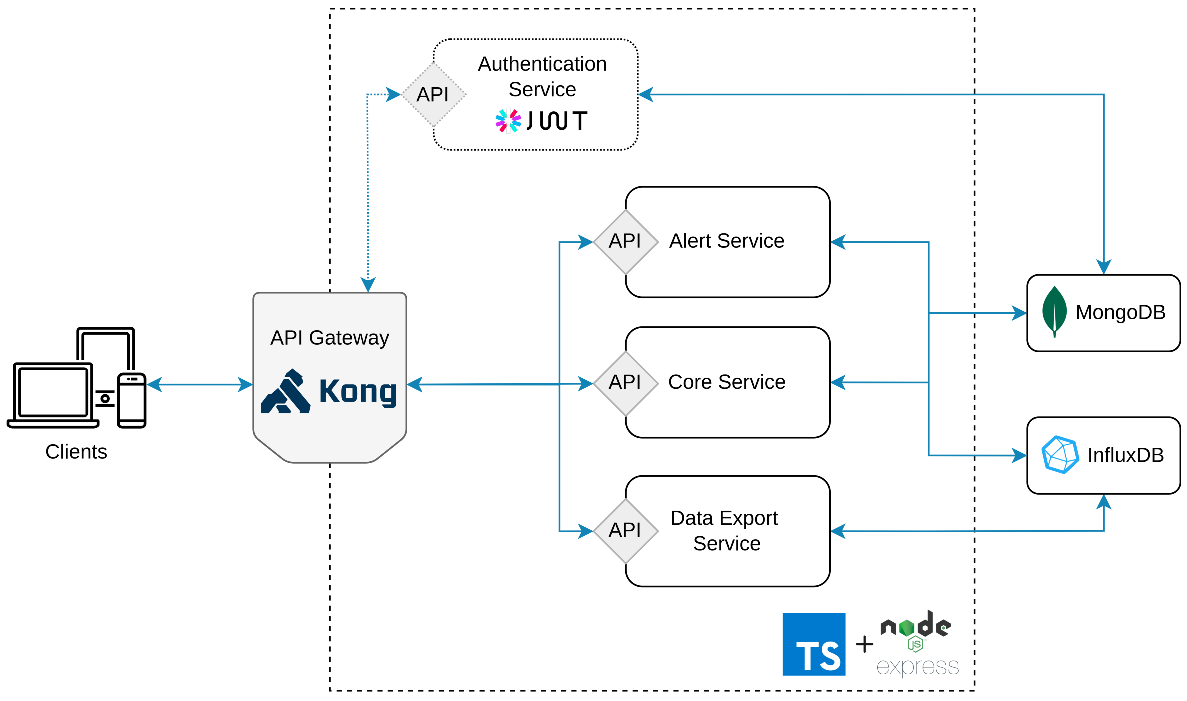


Figure 12: Guardian server — internal architecture

4.4.2 Local Server

The local server has two services: **push-based service**, also known as "receiver", and **pull-based service**, also known as "executor". These services communicate with the guardian server over HTTP requests using access tokens provided to each organization.

This architecture with two services exists due to the need to support the two aforementioned data collection paradigms — pull-based and push-based —, as the services' names indicate.

The **push-based service**, or "receiver", is the one responsible to handle all monitoring agents that are of the "deadman" type, i.e., those that do not need a service to be querying some target for metrics or information. Instead, the monitoring services are the ones that report its state to the "receiver". The "receiver" then sends the information to the guardian server to be stored, analyzed and acted on (if there is a need to notify any problems, i.e., send alerts).

On the other hand, the **pull-based service**, or "executor", handles the other types of monitoring agents, that is, the pull-based ones. This service is responsible for frequently contacting the guarding server to update its internal information on what pull-based monitoring agents its organization has. Based on each monitoring agent repetition configuration, it schedules the pull-based monitoring agent's execution and for each execution reports the obtained data to the guardian server to, as was the case in the previous service, be stored, analyzed and acted on.

Figure 13 presents a diagram that illustrates the described architecture.

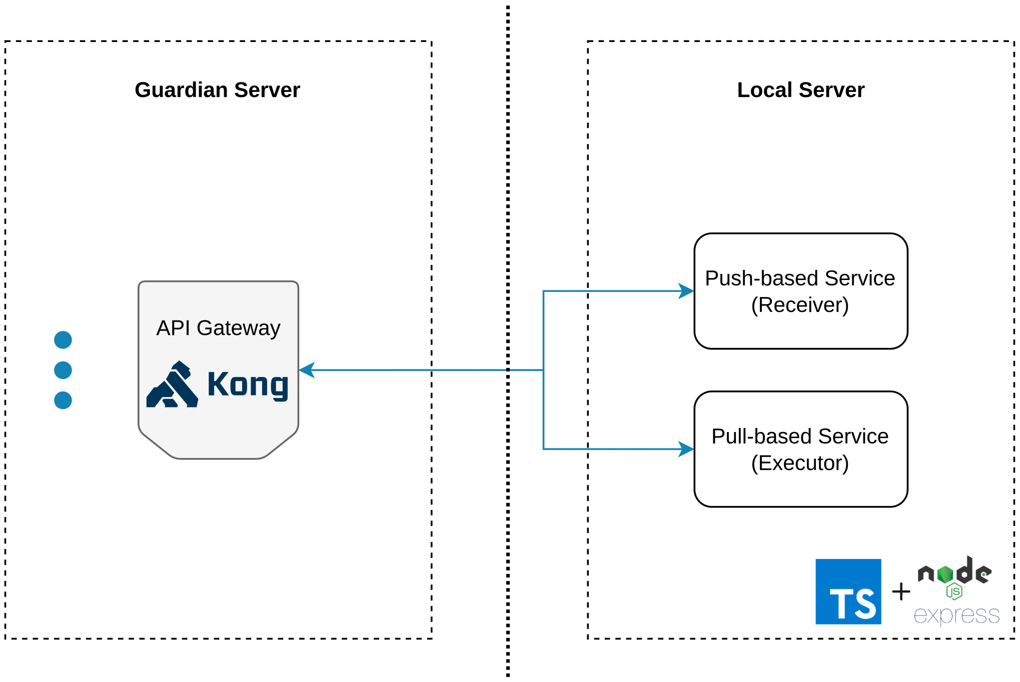


Figure 13: Local monitoring server — internal architecture

4.4.3 Deployment

One of the core non-functional requirements, as detailed in section 4.2.2, is the deployability of the solution. To ensure this, it was designed a modular and efficient deployment architecture. The core basis of the architecture is Docker[2] which is an open-source containerization platform that enables developers to package applications into lightweight and easy to use containers [55]. As such, all services and infrastructure components, i.e., API gateway and databases, are packaged into Docker containers, which are then deployed on each targeted server. To summarize, the components of the guardian server are Docker containers installed on its off-site server and the local servers' components are Docker containers installed on each organization's on-site server. Figure 14 presents a diagram that illustrates the global deployment architecture.

As is possible to see in figure 14, every component is a Docker container and only the containers that are supposed to be reached outside the Docker network have external ports, that is, the API gateway in the guardian server (so that external clients and services can communicate with it) and the push-based service in the local servers (so that the push-based services that are being monitored can push their monitoring information to the service). Otherwise, the containers only have an internal port that is only accessible by other containers through the Docker's internal network.

[2]https://www.docker.com

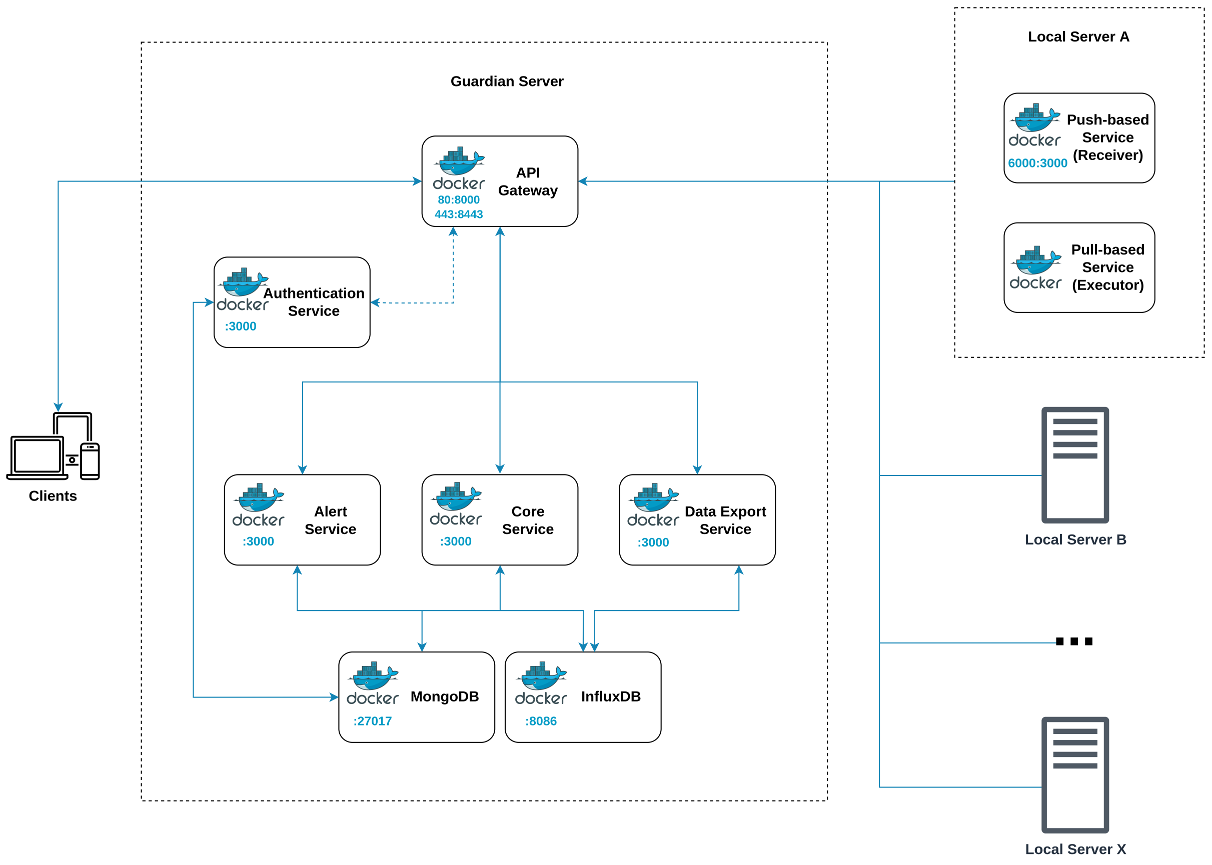


Figure 14: Global deployment architecture

4.5 Discussion

This chapter started with the analysis of the stakeholders' input and perspective followed by a description of the solution's requirements. This section will discuss how those requirements are effectively addressed by the presented architecture.

Data collection is handled by the local server, either via the pull-based service or the push-based service, depending on the monitoring agent's type. This data can be accessed via the guardian server's data export service, encapsulated in the API gateway.

Multi-site and multi-organization support is also provided since each organization can be added and consulted independently and each monitoring site, i.e., each monitoring facility, has its own local server and access token.

Alerting and configuration are also ensured since the guardian server has services that handle these requirements, alert service and core service, respectively.

Furthermore, all core logic management operations specified in the requirements and domain model, that is, monitoring agents' and health/status level settings' configuration, are handled by the guardian server's core service.

Regarding non-functional requirements, the guardian server's authentication service and API gateway security methods (namely, SSL certificate and throttling rules) solve all information security issues as thoroughly as feasible within the scope of this Dissertation. The selection of a microservices-based architecture, as well as technological options and deployment architecture and strategies, address the mentioned reliability, scalability and deployability concerns. The latter, i.e., the deployment strategies, will be covered in greater depth in the following chapter.

Overall, the proposed architecture is capable of meeting the presented requirements through the components that comprise it, meaning it is ready to be implemented, tested and validated.

4.6 Summary

This chapter presented all the exploration and modeling of the proposed IT monitoring solution for the healthcare environment. First, the technical questionnaires' results were outlined, followed by the description of the different requirements, functional and non-functional. Then, it is presented the domain modal to better understand the system's core logic and, finally, the architecture design, which is microservices-based, with both high-level and low-level architectural views. As such, through the proposed solution a user can create, manage and consume the monitoring agents and all related entities, as well as the produced/collected monitoring information and health status. These functionalities are leveraged by two different servers: the guardian server and the local server. The guardian server accumulates the responsibilities of all its services, namely, the authentication and authorization of the users, the creation and management of the organizations, groups, monitoring agents, health/status level settings, notification channels and alerts' configurations. On the other hand, the local server is responsible for collecting the monitoring information of each monitoring agent and sending it to the guardian server.

Chapter 5

Implementation and Results

In this chapter, it is described the implementation of the solution proposed and presented in the previous chapter, followed by the presentation of the subsequent results.

First, the data models used to store all the required information are presented, followed by details regarding the implementation of the API gateway, as well as the solution's two comprising servers, the guardian server and the local server. Next, it is described how every microservice is documented through market-standard API documentation, as well as the implementation of the solution's deployment using containerization techniques. Finally, are showcased some examples of the system's core features and some results regarding an external web application outside of the scope of this Dissertation that integrates with the implemented solution to provide visual examples of the solution's capabilities.

5.1 Data Models

The developed solution handles different data with different formats and purposes in different storage mechanisms/solutions. All the handled data can be grouped by the storage solution in which they are stored: monitoring data, which is stored in InfluxDB, and entities' configuration data, which is stored in MongoDB.

5.1.1 Monitoring Data

The data in InfluxDB is stored in a particular format in lines, each line containing three types of information: the **measurement**, which specifies a namespace for each type of data being recorded, the **fields**, which are the primary information for each measurement, and the **tags**, which are complementary information for each measurement. As an example, this is the information structure for the HTTP's monitoring data:

- **measurement**: http
- **tags** - list of tags associated with the measurement
  - **id** - the unique identifier of the monitoring agent
  - **name** - the name of the monitoring agent
  - **group** - the group in which the monitoring agent belongs
  - **org** - the group's parent organization in which the monitoring agent belongs
- **fields** - list of fields associated with the measurement
  - **url** - the URL of the HTTP service being monitored
  - **method** - the method of the HTTP request
  - **status_code** - the status code of the HTTP request
  - **response_time** - the response time of the the HTTP request, in seconds
  - **result_code** - an integer value, 0 or 1, representing a boolean condition if the target being monitored is approachable and compliant

All other monitoring agent metrics have a similar data structure, in which the tags are always identical and the fields are different for each agent except for the **result_code**, which is a generic metric to standardize status output across all agent types.

5.1.2 Entities' Data

The data stored in MongoDB is representative of the entities' configuration data, namely every organization, group, user, agent and alert configured in the system. The data is stored in a particular document-based format in which each entity is represented by a MongoDB collection.

The organizations and groups are both used to represent the system's hierarchy and have very similar information which includes their unique identifier, name and creation and last update timestamps, as can be seen in examples 1 and 2. This hierarchy representation is not based on a fixed tree structure, but a dynamic aggregation tree structure that is refreshed every time needed since the necessary logical information is stored in the user's and agent's configurations, as can be seen in examples 3 and 4. This means that rather than having a structure that defines which groups each organization has and which users have access to which organizations, this information is stored in the agents' and users' documents, respectively.

```json
{
  "id": "623f22355286eedbede51619",
  "name": "CHTS",
  "createdAt": 1648304693,
  "updatedAt": 1655196089
}
```

Example 1: Organization's data structure

```json
{
  "id": "623f222a5286eedbede51613",
  "name": "AIDA_Database",
  "createdAt": 1648304693,
  "updatedAt": 1655196089
}
```

Example 2: Group's data structure

The user's collection stores all information related to the system's users, including their unique identifier, name (first and last names), email, password, and creation and last update timestamps. The password is stored as a hash of the original password and an additional salt. The user's collection also stores the user's access to the system's organizations, which is represented by a list of the organizations' unique identifiers and a flag that represents if the user is currently active or not. An example of the user's data structure can be seen in example 3.

```json
{
  "id": "6240a0901f445e549b378f0a",
  "firstName": "Joe",
  "lastName": "Smith",
  "email": "joesmith@example.com",
  "organizations": [
    "623f22355286eedbede51619",
    "..."
  ],
  "password": "$2b$10$4UIK6NzXwi0YBaWnJvq1zOlvJv5WrAyYB8awdMopaBQRht5ePKPt2",
  "isActive": true,
  "createdAt": 1648398976,
  "updatedAt": 1649154966
}
```

Example 3: User's data structure

The agent's collection stores the information related to which organization and group it belongs to and all configuration details. This includes the agent's unique identifier, kind (an agent to OracleSQL, HTTP endpoint, push-based external service, i.e., deadman, among others), name, configuration details, creation and last update timestamps, a flag that represents if the agent is currently active or not and the agent's health/level status. The configuration details structure has a common base to every agent's kind, which is the repetition's definition and all the remaining specifications vary according to each kind and its requirements.

The example 4 shows the configuration of an active agent that monitors an OracleSQL target with a repetition window of five minutes and that has the mandatory critical health level defined. It is worth noting that for each agent it is also possible to define the optional "warning" and "ok" health levels and that the definition of these health levels follows the representation of a logical statement in JSON as specified in the JsonLogic format[1], which is going to be more detailed in the subsection 5.3.2.

```json
{
  "id": "b25d2c86-c278-4d75-bcef-546ed9dba788",
  "kind": "oraclesql",
  "org": "623f22355286eedbede51619",
  "group": "623f222a5286eedbede51613",
  "name": "AIDA BD",
  "is_active": true,
  "config": {
    "repetition": {
      "value": 5,
      "unit": "min"
    },
    "conn_string": "192.168.237.78:1666/dba",
    "username": "user_dummy",
    "password": "user_dummy_password"
  },
  "levels": {
    "critical": {
      "==": ["result_code", 1]
    }
  },
  "createdAt": 1648304693,
  "updatedAt": 1658259856
}
```

Example 4: Agent's data structure

Finally, the alert's collection encloses all the information related to the alert's configuration. This includes the alert's unique identifier, name, type, i.e., Email, Slack, Discord, among others, creation and last update timestamps, the associated agent, the repetition window of execution and all the necessary settings of each alert type. An example of an email alert data structure can be found in example 5, which clearly defines all of the aforementioned information as well as the specific information to the email alert, that is, the required mailing list to which the alert should be sent.

5.2 API Gateway

The API gateway is the entry point from which all clients interact with the solution. As such, it needs to be able to route and forward requests to each service depending on the context and information of the request, handling Cross-Origin Resource Sharing (CORS) permissions, ensuring authentication and authorization of the requesting entity of each communication, providing an SSL certificate, and enforcing communication over Hypertext Transfer Protocol Secure (HTTPS) to maximize security.

[1] https://jsonlogic.com/operations.html

```json
{
  "id": "09ba9537-7077-4a4b-bc78-b5626e14116c",
  "name": "Email Alert 09ba9537-7077-4a4b-bc78-b5626e14116c",
  "type": "email",
  "every": "1m",
  "agentId": "b25d2c86-c278-4d75-bcef-546ed9dba788",
  "mailingList": [
    "joesmith@example.com",
    "..."
  ],
  "createdAt": 1648306162,
  "updatedAt": 1657642634
}

```

Example 5: Alert's data structure - Email

By using Kong it was possible to integrate all these features into a single, secure, and well-defined component that is completely configurable in a declarative manner, using JSON or YAML. Example 6 contains a portion of the Kong API gateway's YAML specification, namely the core service's configuration.

```yaml
- name: core
  host: core
  port: 3000
  protocol: http
  routes:
    - name: core-docs
      paths:
        - /core
        - /core/api
        - /core/api/docs
      strip_path: false
    - name: core
      paths:
        - /core/api/agents
        - /core/api/groups
        - /core/api/organizations
      strip_path: false
      plugins:
        - name: custom-auth
          config:
            authentication_endpoint: auth:3000/api/auth/validate
            token_header: Authorization
```

Example 6: Kong configuration of the core service

5.3 Guardian Server

The guardian server, as already stated, is the centralized component of the solution. It is where all the management, storage and consumption components of the solution are located and made available. As specified in previous chapters, more precisely, 3.3 and 4.4, the guardian server is comprised of four core components, all designed following a microservices-based architecture. The technologies used in their implementation were a combination of Express.js with Typescript and efforts were made so that all components, which, for simplicity, are going to be called services from now on, would have a similar structure, both in terms of folders, files and code organization.

Furthermore, each service needs some configurations to be set, which followed a standardized structure to maintain coherence and reusability across all services. These configurations are stored in a ".env", which is a simple text file that contains key-value pairs, used by the "dotenv" package[2] to load the environment variables to the application.

Finally, so the solution would be observable and possible to monitor and debug, all services have a logging mechanism, implemented using the "winston" package[3], that automatically saves all logs to a file and outputs those same logs to a console if one is attached.

5.3.1 Authentication

The authentication service handles the creation and validation of system credentials. As stated in section 3.3.1, the technology used to implement authentication in the system is token-based authentication using JWTs. This service exposes endpoints that allow the user to login into the system, i.e., generate token-based credentials to communicate with the system and validate authentication credentials (JWTs) to use the other services of the solution.

To implement this functionality, this service has access to the users' collection in order to validate login information and it uses a node package called "jsonwebtoken"[4] to generate the JWT token-based credentials.

The sequence diagrams illustrated in the figures 15 and 16 summarise the client authentication process carried out for each request on the system. Figure 15 illustrates the sequence of actions that are performed when a client tries to log in to the system to receive his authentication token, and figure 16 the sequence of actions undertaken to authenticate each request.

Figure 15 illustrates the actions that are executed if a client does not already have a stored token or when the token is no longer valid, i.e., the 24-hour period has expired. When a client receives the token, it is expected to save it for future usage in new requests; otherwise, the client will have to log in again. When a client already has a valid token stored, the processes portrayed in figure 16 are followed for each request.

[2]https://www.npmjs.com/package/dotenv
[3]https://www.npmjs.com/package/winston
[4]https://www.npmjs.com/package/jsonwebtoken

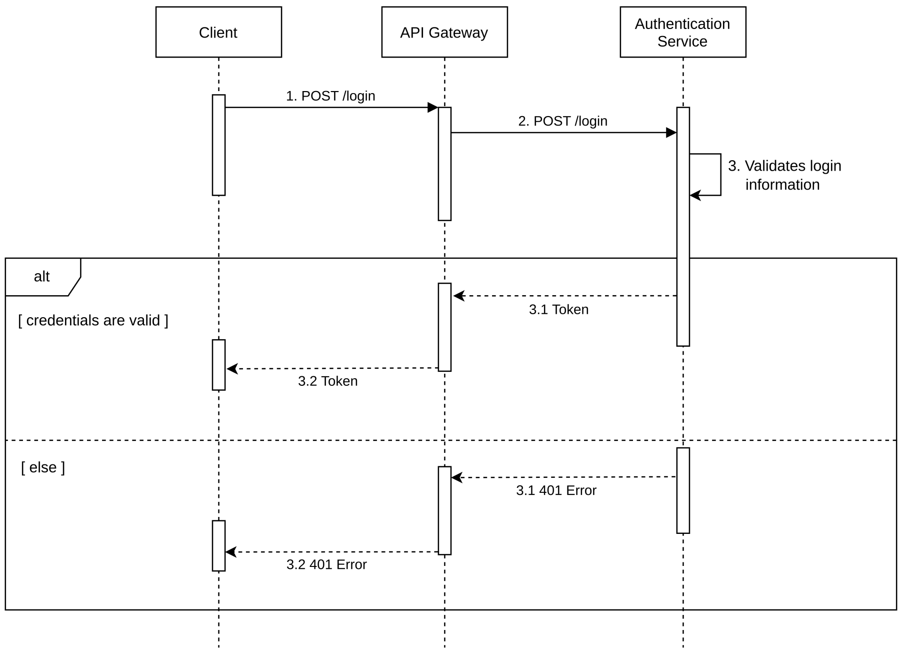


Figure 15: Sequence diagram of the interactions between the client and the solution to obtain a JWT token

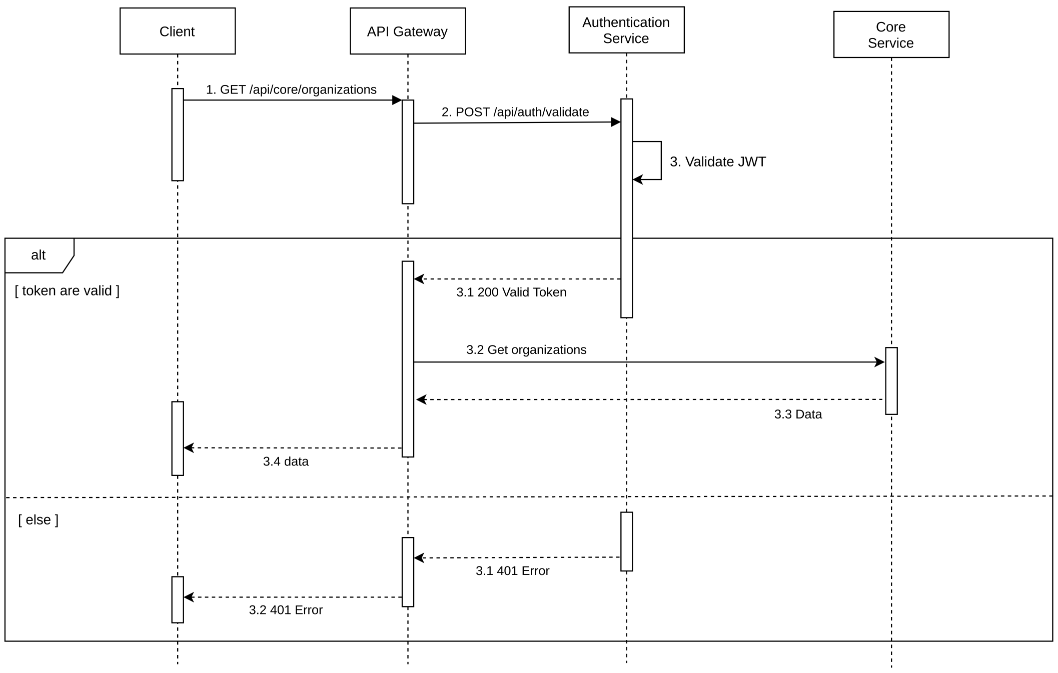


Figure 16: Sequence diagram of the interactions between the client and the solution when performing a request

Figure 16 depicts the series of events that occur in response to each HTTP request. The diagram exemplifies a request to list all organizations, but it may be used for any other request. Each request's HTTP Authorization header contains the token, which is verified by the API gateway using the authentication service. If the token is invalid, the client receives an error message with response code 401, which is the HTTP response status code that indicates a request was not processed because the provided authentication credentials were invalid or insufficient to acquire the desired resource.

5.3.2 Core Business Logic

The core service is the biggest component of the solution. As already stated in section 4.4.1, it is the one that manages all entities except the alerts, that is, the organizations, groups, users, and monitoring agents, including the respective health/level status definitions.

The logic regarding organizations, groups and users is really simple since these three entities always have the same type of information and their operations are of creation and management alone. As such, it is not necessary for any further detailing on their implementation.

On the other hand, the monitoring agents and respective health level definitions are more complex. Since the system must be ready to support a large variety of types of monitoring agents, these need to be programmatically defined in a way that enables the addition of new monitoring agents and even the extension of previously defined ones. To accomplish this, it was decided that the monitoring agents would be defined using OOP concepts such as abstraction and inheritance, which, again, was one of the main reasons to use Typescript instead of Vanilla JavaScript for the microservices' implementation.

Figure 17 illustrates the applied strategy. The monitoring agent base definition is represented by the abstract class **Monitoring Agent** which includes the common information all monitoring agents should contain, despite their kind. Then, all the actual monitoring agents extend the abstract class, each one being a subclass with its custom configuration properties.

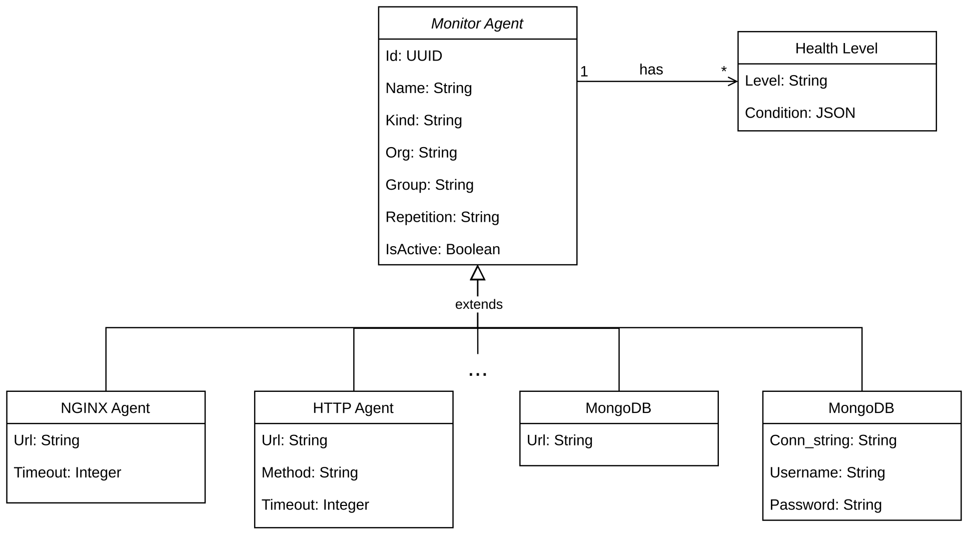


Figure 17: Class diagram depicting the use strategy to represent the monitoring agents' configurations

The health level definitions, as figure 17 further illustrates, are always associated with a monitoring agent and have two core properties: the health/status level, which is a simple string field, and the logical condition that must be met to fall under that level. As stated in section 5.1.2, the logical condition is represented as a JSON object that follows the JsonLogic format which is a simple way to define logical conditions with the possibility of logical validation using a JSON logical parser. Examples 7 and 8 illustrate the use of the JsonLogic format.

```
{
  "and": [
    {
      "==": ["result_code", 1]
    },
    {
      "or": [
        {
          "<=": ["status_code", 400]
        },
        {
          ">=": ["resp_time", 500]
        }
      ]
    }
  ]
}
```

Example 7: Logical condition based on “AND” and “OR” operations

```
{
  "or": [
    {
      "!=": ["result_code", 0]
    },
    {
      ">=": ["status_code", 400]
    },
    {
      ">=": ["resp_time", 500]
    }
  ]
}
```

Example 8: Logical condition based on an “OR” operation

5.3.3 Alerting

The alert service, as stated in section 4.4.1, is responsible for creating and managing the alerts configured or to be configured in the system. Furthermore, it also has the responsibility of crunching the metrics data stored in the InfluxDB database, determining the health/status level of the monitoring agents by comparing the crunched data with the health status definitions and sending the respective alerts, if applicable, according to the existing configurations associated with each monitoring agent. This mechanism was implemented by using and extending the underlying InfluxDB components, namely the Kapacitor[5] component.

Furthermore, in addition to the service allowing the dispatch of every alert type available in the system, it also allows an additional dispatch configuration that is associated by default with each monitoring agent to allow real-time alerting consumption of external applications. This is accomplished using an event-driven library, called Socket.io[6], that enables real-time, bi-directional communication between web clients

[5]https://www.influxdata.com/time-series-platform/kapacitor
[6]https://socket.io

and servers. This functionality is provided by exposing a consumption endpoint connection to which each client application can connect and communicate with the alert service.

5.3.4 Data Exporting

Finally, the last component of the guardian server, the data export service, is responsible for the exporting of data to external clients, whether raw metrics related to monitoring agents or their current and past health status. This service is intended to be used as a data source for other applications, such as dashboards, that need to display the monitoring and health status data. All the data is possible to be exported by using a total of three provided HTTP endpoints in the form of an array of JSON objects or as a Comma-Separated Values (CSV). One for the consumption of metrics data of each monitoring agent with multiple filtering and manipulation mechanisms (such as start and end time, data aggregation and others), the other for the consumption of the historic health status levels of a monitoring agent, and the last one for the consumption of the current health status level of each monitoring agent.

5.4 Local Server

The local server is the software that runs in each healthcare facility or organization. Although it is called a server, it runs as a multi-part client responsible for the data collection of an organization. As stated in section 4.4.2, the local server is composed of two different services, one for each data collection paradigm, that is, pull-based and push-based data collection.

Moreover, as in the guardian server, these services were implemented using a combination of Express.js with Typescript while maintaining a similar structure to the guardian server's microservices, both in terms of folders, files and code organization. The strategies of configuration management, logging and observability used in the guardian server were applied to the local server as well.

5.4.1 Push-based Data Collection

The push-based data collection service is responsible to report to the guardian service all the data received from all the services that are being monitored via a push-based data collection paradigm. It was implemented via a web server that has specific endpoints that handle the data communication from the monitored services and every time new data is received from those monitored services, first, it is validated if the monitored service exists and is indeed valid in that organization, to avoid attempts of fake services intrusion and then, relays that information to the guardian server to be processed, analyzed, stored and made available for later inspection and consumption.

5.4.2 Pull-based Data Collection

On the other hand, the pull-based data collection service handles all the monitoring agents that were defined with the pull-based data collection paradigm, which are executed using cronjobs[7]. This service is a script client that has a periodical sync/update procedure. The periodicity is defined by the monitoring agent's configuration and has a default value of two and a half minutes. In addition, this client has also an initial mechanism to recover and reschedule the agents if the service crashes or is restarted. Algorithm 1 has a pseudo-code specification of the overall service implementation. As it can be seen, every time the service restarts, the first thing it does is recover (create and schedule) the monitoring agents configured in the system. After that, it begins the cycle of sync and updates with the guardian server in three stages: first, it unschedules and removes the monitoring agents that no longer exist in the system, then it creates and schedules the monitoring agents that were configured and created since the last sync cycle, and finally, it updates, if necessary, the monitoring agents that are already scheduled.

5.5 API Documentation

Since the developed monitoring solution was designed with external integration, communication and consumption in mind, it was essential that the entire solution was documented and that this same documentation was freely accessible on the web. As such, every service and endpoint is documented through standard API documentation. The documentation was accomplished by using an open-source API documentation engine called Redocly[8]. While analyzing the API documentation technological possibilities were found two main ones: Swagger and Redocly, both supporting the OpenAPI specification[9]. The main reason for choosing Redocly over Swagger was Redocly's ability to visually represent different property definitions of an object depending on the choice of the value of a given attribute, which is essential to this solution since each monitoring agent has different configuration properties depending on the agent type. Figures 18 and 19 exemplify the aforementioned pretended behavior which changes the configuration's parameters list depending on the selected value of the "kind" parameter.

The documentation definition was implemented using two complementary node packages: swagger-jsdoc[10] and redoc-express[11]. The swagger-jsdoc package was used to initialize the OpenAPI specification and the redoc-express package, which builds on the OpenAPI definition created with the swagger-jsdoc package, was used to generate and serve the documentation by applying YAML-based comment decorators to the endpoints with a similar approach to the one in example 9.

[7] A cron job is a Linux command that is used to schedule tasks to be run in the future. This is often used to schedule a job to be run on a routine basis, such as sending an HTTP request to an external web service every two hours [56].

[8] https://redocly.com/reference

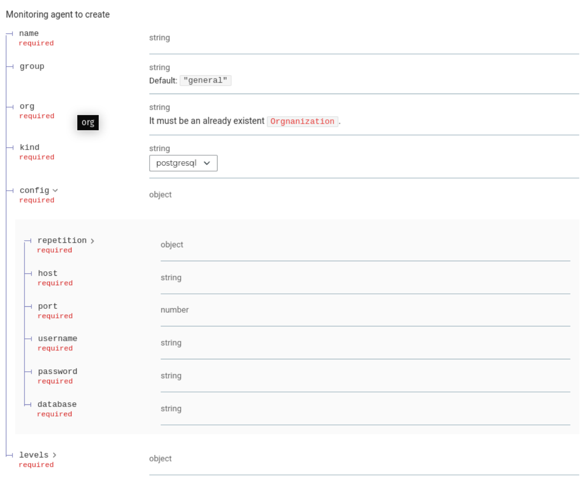
[9] https://swagger.io/specification

[10] https://www.npmjs.com/package/swagger-jsdoc

[11] https://www.npmjs.com/package/redoc-express

Algorithm 1: Local pull-based service agents' sync/update algorithm

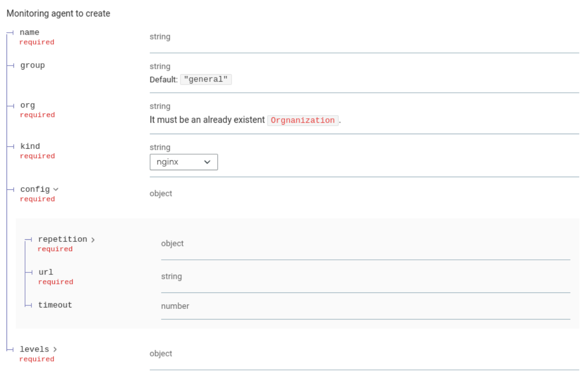
```
$agents \leftarrow getPullBasedAgents()$ // pull-based monitoring agents in the system
$currentAgents \leftarrow getCurrentAgentsFromSchedulingService()$

// Recover/recreate agents if service went down
foreach agent in agents do
    if $agent$ not in $currentAgents$ then
        createAndSchedule($agent$)
    end
end

// Sync/update agents execution/scheduling with the system
while true do
    $agents \leftarrow getPullBasedAgents()$ // update agents' list
    $currentAgents \leftarrow getCurrentAgentsFromSchedulingService()$

    // Delete agents that no longer exist in the system
    foreach currentAgent in currentAgents do
        if $currentAgent$ not in $agents$ then
            unscheduleAndDelete($currentAgent$)
        end
    end

    // Create agents that do not yet exist locally
    foreach agent in agents do
        if $agent$ not in $currentAgents$ then
            createAndSchedule($currentAgent$)
        end
    end

    // Update, if necessary, the remaining agents
    foreach agent in agents do
        if $agent$ in $currentAgents$ then
            if $agent$ updated then
                updateAndReschedule($currentAgent$)
            end
        end
    end

    sleep(2.5) // Sleep 2 minutes and 30 seconds between each sync/update cycle
end
```



Monitoring agent to create

name required	string
group	string Default: "general"
org required	string It must be an already existent Orgnanization.
kind required	string postgresql
config required	object
repetition required	object
host required	string
port required	number
username required	string
password required	string
database required	string
levels required	object

Figure 18: API documentation: PostgreSQL agent



Monitoring agent to create

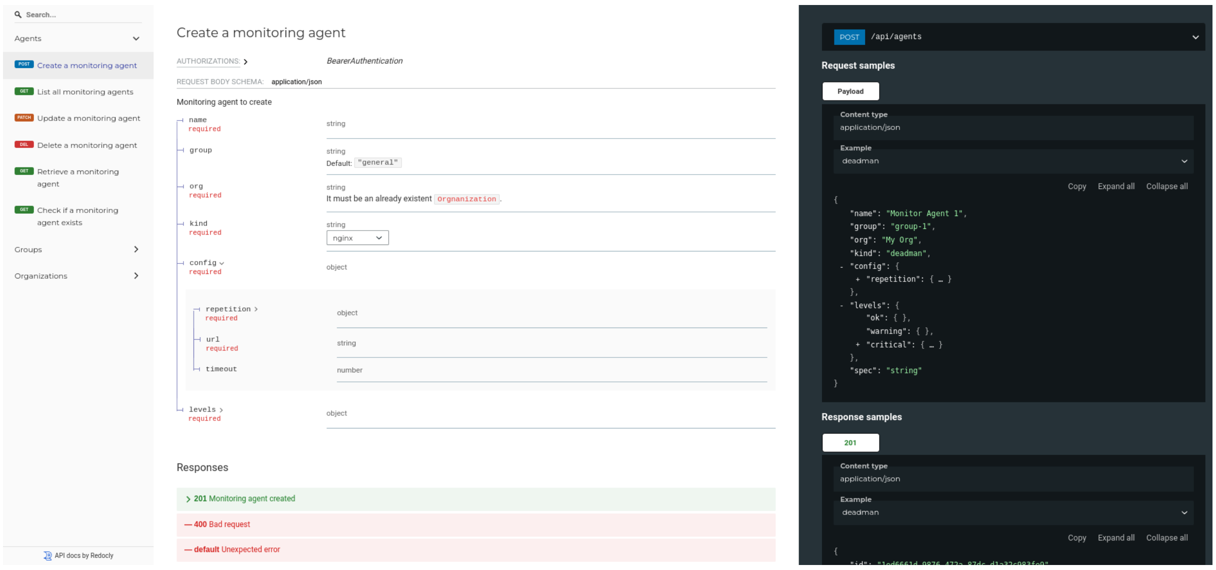
name required	string
group	string Default: "general"
org required	string It must be an already existent Orgnanization.
kind required	string nginx
config required	object
repetition required	object
url required	string
timeout	number
levels required	object

Figure 19: API documentation: NGINX agent

```
/**
 * @swagger
 * /api/agents:
 *   post:
 *     operationId: CreateAgent
 *     tags:
 *       - Agents
 *     summary: Create a monitoring agent
 *     requestBody:
 *       content:
 *         application/json:
 *           schema: $ref: "#/components/schemas/NewMonitor"
 *       description: Monitoring agent to create
 *       required: true
 *     responses:
 *       201:
 *         description: Monitoring agent created
 *         content:
 *           application/json:
 *             schema: $ref: "#/components/schemas/Monitor"
 *       400:
 *         description: Bad request
 * */
router.post("/", (request, response) => { ... });
```

Example 9: YAML-based decorator used to specify the creation endpoint of a monitoring agent

Finally, every microservice has its own API documentation page, all built with the tool and strategies described above. Figures 20, 21 and 22 showcase some examples of the aforementioned documentation pages.


Search...
Agents
POST Create a monitoring agent
GET List all monitoring agents
PATCH Update a monitoring agent
DEL Delete a monitoring agent
GET Retrieve a monitoring agent
GET Check if a monitoring agent exists
Groups
Organizations
API docs by Redocly

Create a monitoring agent

AUTHORIZATIONS: BearerAuthentication

REQUEST BODY SCHEMA: application/json

Monitoring agent to create

name required	string
group	string Default: "general"
org required	string It must be an already existent orgnanization.
kind required	string nginx
config required	object
repetition required	object
url required	string
timeout	number
levels required	object

Responses

201 Monitoring agent created

400 Bad request

default Unexpected error

POST /api/agents

Request samples

Payload

Content type: application/json

Example: deadman

```
{
    "name": "Monitor Agent 1",
    "group": "group-1",
    "org": "My Org",
    "kind": "deadman",
  - "config": {
      + "repetition": { … }
    },
  - "levels": {
        "ok": { },
        "warning": { },
      + "critical": { … }
    },
    "spec": "string"
}
```

Response samples

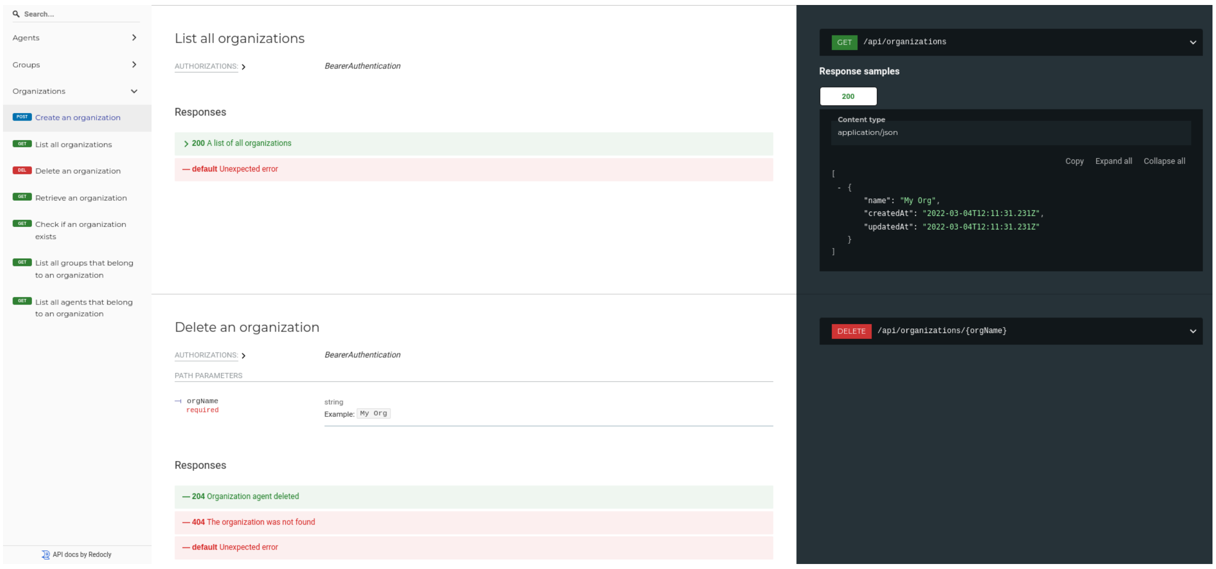
201

Content type: application/json

Example: deadman

```
{
    "id": "1ed6661d-9876-472a-87dc-d1a32c983fe0"
```

Figure 20: Core service API documentation page - monitoring agent creation


Search...
Agents
Groups
Organizations
POST Create an organization
GET List all organizations
DEL Delete an organization
GET Retrieve an organization
GET Check if an organization exists
GET List all groups that belong to an organization
GET List all agents that belong to an organization
API docs by Redocly
Copy
Expand all
Collapse all

List all organizations

AUTHORIZATIONS: BearerAuthentication

Responses

200 A list of all organizations

— default Unexpected error

Delete an organization

AUTHORIZATIONS: BearerAuthentication

PATH PARAMETERS

orgName required — string, Example: My Org

Responses

— 204 Organization agent deleted

— 404 The organization was not found

— default Unexpected error

GET /api/organizations

Response samples

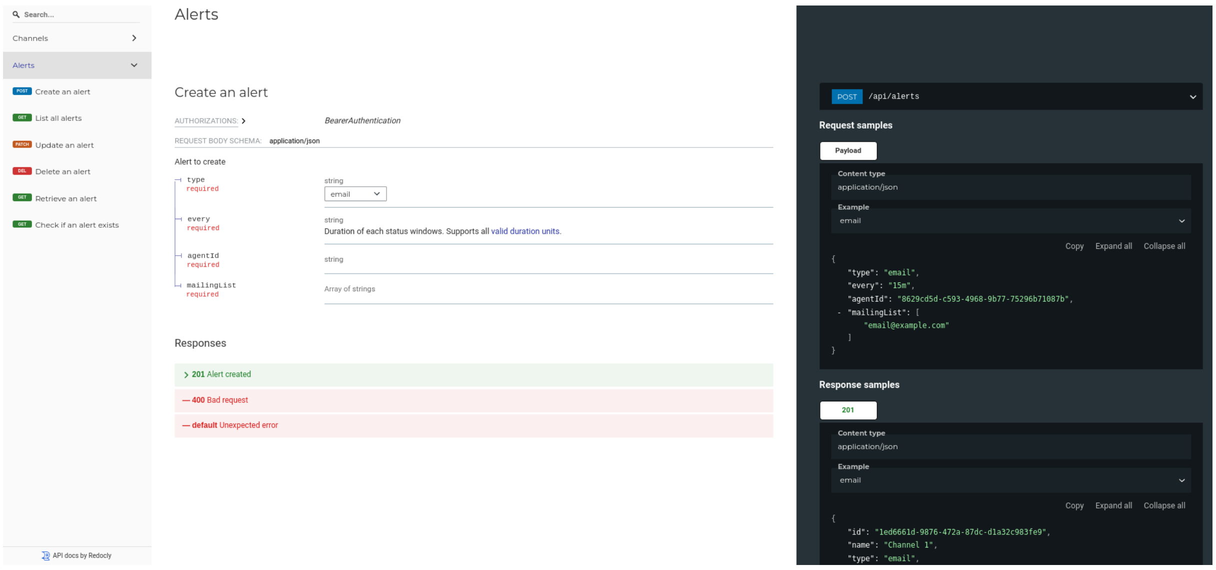
200

Content type: application/json

```
[
 - {
       "name": "My Org",
       "createdAt": "2022-03-04T12:11:31.231Z",
       "updatedAt": "2022-03-04T12:11:31.231Z"
   }
]
```

DELETE /api/organizations/{orgName}

Figure 21: Core service API documentation page - organizations querying



Alerts

Create an alert

AUTHORIZATIONS: BearerAuthentication

REQUEST BODY SCHEMA: application/json

Alert to create

type required — string, email

every required — string, Duration of each status windows. Supports all valid duration units.

agentId required — string

mailingList required — Array of strings

Responses

201 Alert created

— 400 Bad request

— default Unexpected error

POST /api/alerts

Request samples

Payload

Content type: application/json

Example: email

```
{
   "type": "email",
   "every": "15m",
   "agentId": "8629cd5d-c593-4968-9b77-75296b71087b",
 - "mailingList": [
       "email@example.com"
   ]
}
```

Response samples

201

Content type: application/json

Example: email

```
{
   "id": "1ed6661d-9876-472a-87dc-d1a32c983fe9",
   "name": "Channel 1",
   "type": "email",
```

Figure 22: Alert service API documentation page - alert creation

5.6 Deployment

The solution's deployment, as mentioned and detailed in section 4.4.3, follows the microservices architecture and deploys that architecture using Docker containers, one container to each service and infrastructure components, i.e., API gateway and databases. To accomplish this, every one of these components has a Dockerfile, which is the file used to build and create the Docker image for that component. Since all the services are structured and organized similarly and use the same technology stack, the content of each Dockerfile is similar to the one present in example 10. The only exceptions to this strategy are the API gateway and the databases which have official Docker images built and made available at the official Docker registry[12] and are, therefore, not documented in this document.

```
FROM node:lts-alpine3.14 as ts-build

# Setup Node app
WORKDIR /usr/src/app

# Install app dependencies
COPY --chown=node:node ./package*.json ./
COPY --chown=node:node ./tsconfig*.json ./
RUN npm install

# Bundle app source
COPY . ./

# Build dist
RUN npm run build

#########################################
FROM node:lts-alpine3.14
LABEL org.opencontainers.image.authors="Vasco Ramos"

# Setup Node app
WORKDIR /usr/src/app

# Install app dependencies
COPY --chown=node:node ./package*.json ./
RUN npm ci --production

# Bundle app source
COPY --chown=node:node --from=ts-build /usr/src/app/dist ./dist

CMD [ "npm", "start" ]
```

Example 10: Dockerfile - Core service

As it can be seen in the above example, each Dockerfile is composed of a multi-stage build layout which is useful to optimize and improve the security of Dockerfiles (and the subsequent Docker images) while keeping them easy to read and maintain [57]. This works by creating a temporary image to install any necessary software and dependencies, as well as to create and build the application source. After this, the definitive image build uses only the necessary files from the temporary image to create the final image, keeping the final image lightweight and without unnecessary software and dependencies, which reduces the build's vulnerability risk.

Moreover, so the services' Docker images can be used to create and execute the actual Docker containers, the images need first to be built using the Dockerfiles and made available on a public registry. As such, were created Continuous Integration (CI)/Continuous Delivery (CD) pipelines using the integrated GitHub CI/CD platform, called GitHub Actions[13] to build a Docker image based on a provided Dockerfile

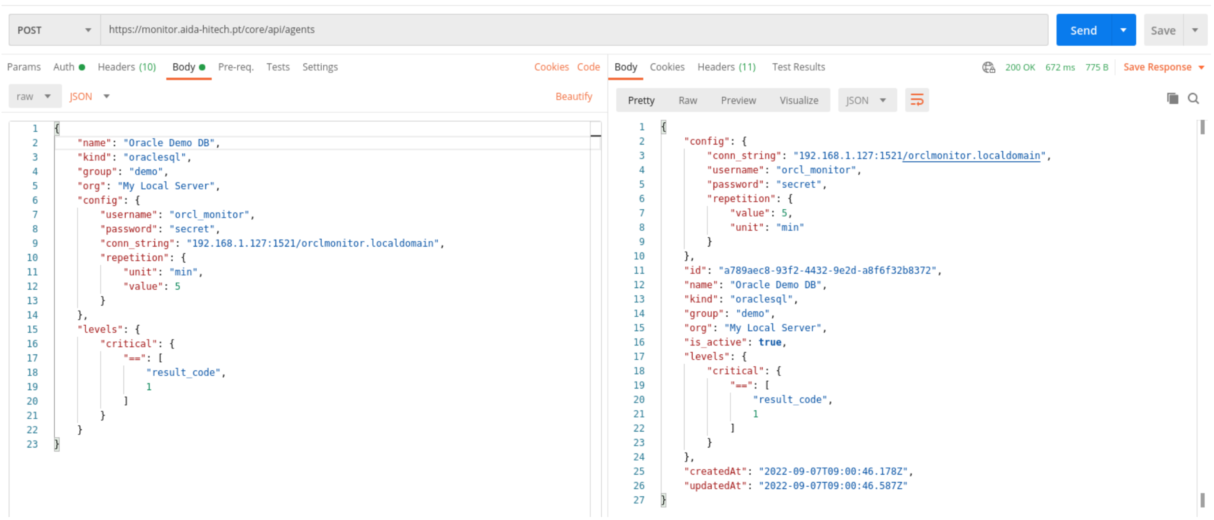
[12]https://hub.docker.com
[13]https://docs.github.com/en/actions

and publish it to the GitHub Packages Container Registry[14] every time a new git tag is pushed to the git repository. The base structure and implementation of this pipeline, or workflow, as is called in the GitHub Actions platform, is available in appendix C.1.

Finally, to deploy or redeploy the solution's servers were created two Docker Compose[15] files, one for the guardian server and the other for the local server, with all the required services for each server, which simplify the task of defining and running multi-container Docker applications through a YAML-based configuration file [58]. Both Guardian Server's and Local Server's Docker Compose files are available in appendixes C.2 and C.3, respectively.

5.7 Core Features Examples

The overall solution has several components, as already explained throughout the document. This section includes some examples of the system's API endpoints to some of the most important features, such as agent creation, alerts creation and monitoring data querying. The examples were executed using an API client named Postman[16]. Figures 23, 24 and 25 show examples of the API calls necessary to the creation of monitoring agents for the OracleSQL, NGINX and Deadman services, respectively.



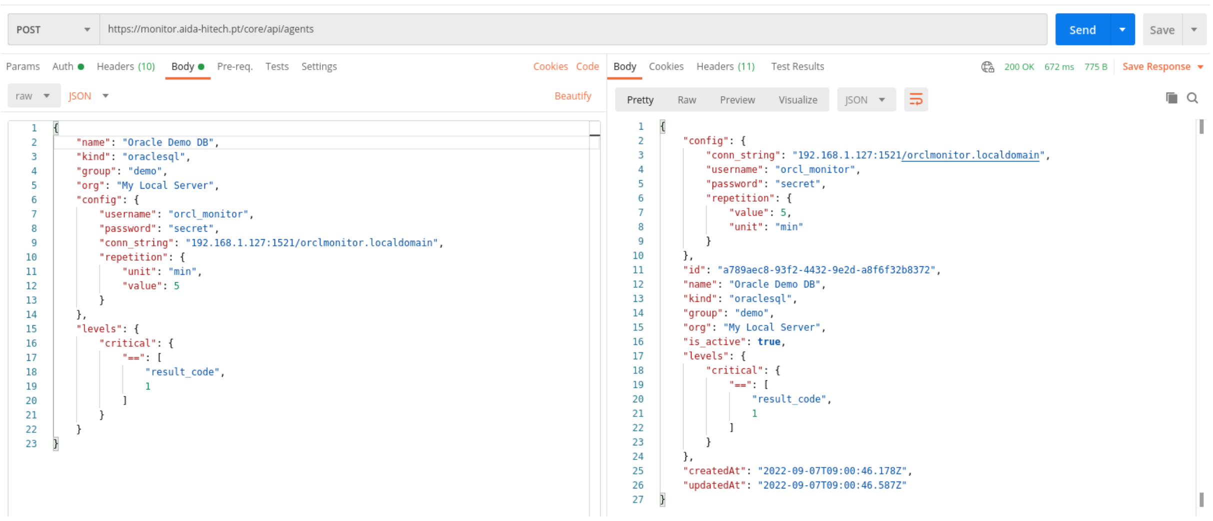
```
POST https://monitor.aida-hitech.pt/core/api/agents
```

```json
{
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "config": {
        "username": "orcl_monitor",
        "password": "secret",
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "repetition": {
            "unit": "min",
            "value": 5
        }
    },
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    }
}
```

```json
{
    "config": {
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "username": "orcl_monitor",
        "password": "secret",
        "repetition": {
            "value": 5,
            "unit": "min"
        }
    },
    "id": "a789aec8-93f2-4432-9e2d-a8f6f32b8372",
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "is_active": true,
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    },
    "createdAt": "2022-09-07T09:00:46.178Z",
    "updatedAt": "2022-09-07T09:00:46.587Z"
}
```

Figure 23: OracleSQL agent creation API call

[14] https://docs.github.com/en/packages
[15] https://docs.docker.com/compose
[16] https://www.postman.com



```
POST https://monitor.aida-hitech.pt/core/api/agents
```

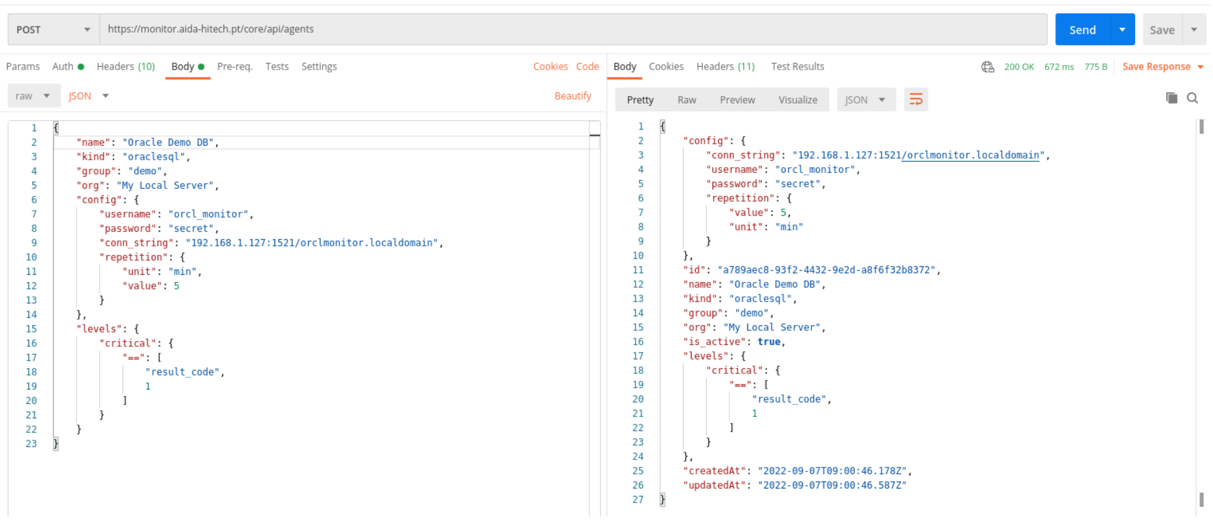
Request body (raw, JSON):

```
{
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "config": {
        "username": "orcl_monitor",
        "password": "secret",
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "repetition": {
            "unit": "min",
            "value": 5
        }
    },
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    }
}
```

Response: 200 OK 672 ms 775 B

```
{
    "config": {
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "username": "orcl_monitor",
        "password": "secret",
        "repetition": {
            "value": 5,
            "unit": "min"
        }
    },
    "id": "a789aec8-93f2-4432-9e2d-a8f6f32b8372",
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "is_active": true,
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    },
    "createdAt": "2022-09-07T09:00:46.178Z",
    "updatedAt": "2022-09-07T09:00:46.587Z"
}
```

Figure 24: NGINX agent creation API call



```
POST https://monitor.aida-hitech.pt/core/api/agents
```

Request body (raw, JSON):

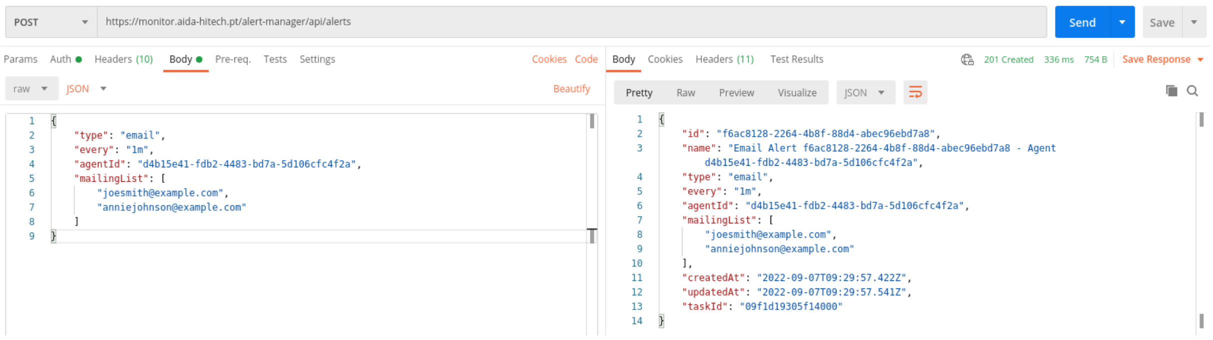
```
{
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "config": {
        "username": "orcl_monitor",
        "password": "secret",
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "repetition": {
            "unit": "min",
            "value": 5
        }
    },
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    }
}
```

Response: 200 OK 672 ms 775 B

```
{
    "config": {
        "conn_string": "192.168.1.127:1521/orclmonitor.localdomain",
        "username": "orcl_monitor",
        "password": "secret",
        "repetition": {
            "value": 5,
            "unit": "min"
        }
    },
    "id": "a789aec8-93f2-4432-9e2d-a8f6f32b8372",
    "name": "Oracle Demo DB",
    "kind": "oraclesql",
    "group": "demo",
    "org": "My Local Server",
    "is_active": true,
    "levels": {
        "critical": {
            "==": [
                "result_code",
                1
            ]
        }
    },
    "createdAt": "2022-09-07T09:00:46.178Z",
    "updatedAt": "2022-09-07T09:00:46.587Z"
}
```

Figure 25: Deadman agent creation API call

Figures 26 and 27 show examples of the API calls necessary to the creation of email and slack alerts, respectively.



```
POST https://monitor.aida-hitech.pt/alert-manager/api/alerts
```

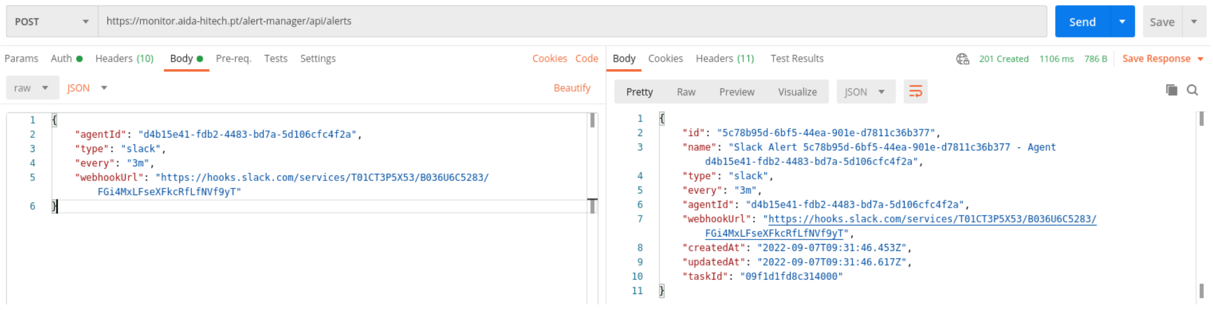
Request body (raw, JSON):

```
{
    "type": "email",
    "every": "1m",
    "agentId": "d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "mailingList": [
        "joesmith@example.com",
        "anniejohnson@example.com"
    ]
}
```

Response: 201 Created 336 ms 754 B

```
{
    "id": "f6ac8128-2264-4b8f-88d4-abec96ebd7a8",
    "name": "Email Alert f6ac8128-2264-4b8f-88d4-abec96ebd7a8 - Agent
        d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "type": "email",
    "every": "1m",
    "agentId": "d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "mailingList": [
        "joesmith@example.com",
        "anniejohnson@example.com"
    ],
    "createdAt": "2022-09-07T09:29:57.422Z",
    "updatedAt": "2022-09-07T09:29:57.541Z",
    "taskId": "09f1d19305f14000"
}
```

Figure 26: Email alert creation API call



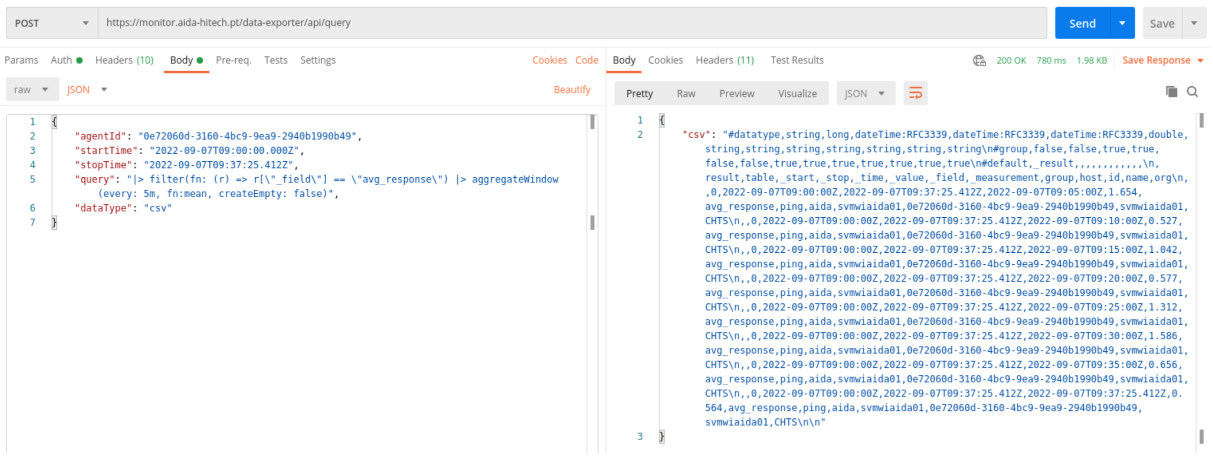
https://monitor.aida-hitech.pt/alert-manager/api/alerts

```
{
    "agentId": "d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "type": "slack",
    "every": "3m",
    "webhookUrl": "https://hooks.slack.com/services/T01CT3P5X53/B036U6C5283/
        FGi4MxLFseXFkcRfLfNVf9yT"
}
```

```
{
    "id": "5c78b95d-6bf5-44ea-901e-d7811c36b377",
    "name": "Slack Alert 5c78b95d-6bf5-44ea-901e-d7811c36b377 - Agent
        d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "type": "slack",
    "every": "3m",
    "agentId": "d4b15e41-fdb2-4483-bd7a-5d106cfc4f2a",
    "webhookUrl": "https://hooks.slack.com/services/T01CT3P5X53/B036U6C5283/
        FGi4MxLFseXFkcRfLfNVf9yT",
    "createdAt": "2022-09-07T09:31:46.453Z",
    "updatedAt": "2022-09-07T09:31:46.617Z",
    "taskId": "09f1d1fd8c314000"
}
```

Figure 27: Slack alert creation API call

And, finally, figure 28 shows an example of an API call to query the monitoring data for a specific agent.



https://monitor.aida-hitech.pt/data-exporter/api/query

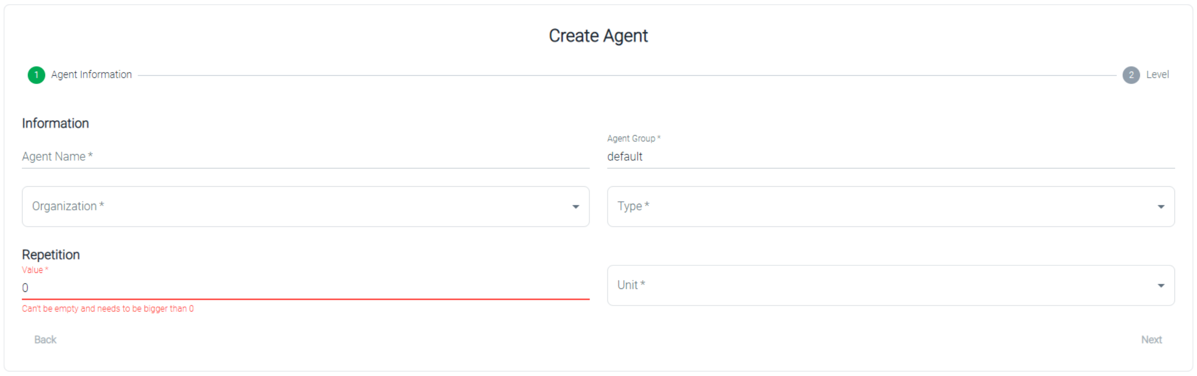
```
{
    "agentId": "0e72060d-3160-4bc9-9ea9-2940b1990b49",
    "startTime": "2022-09-07T09:00:00.000Z",
    "stopTime": "2022-09-07T09:37:25.412Z",
    "query": "|> filter(fn: (r) => r[\"_field\"] == \"avg_response\") |> aggregateWindow
        (every: 5m, fn:mean, createEmpty: false)",
    "dataType": "csv"
}
```

```
{
    "csv": "#datatype,string,long,dateTime:RFC3339,dateTime:RFC3339,dateTime:RFC3339,double,
        string,string,string,string,string,string,string\n#group,false,false,true,true,
        false,false,true,true,true,true,true,true,true\n#default,_result,,,,,,,,,,,,\n,
        result,table,_start,_stop,_time,_value,_field,_measurement,group,host,id,name,org\n,
        ,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:05:00Z,1.654,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:10:00Z,0.527,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:15:00Z,1.042,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:20:00Z,0.577,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:25:00Z,1.312,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:30:00Z,1.586,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:35:00Z,0.656,
        avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,svmwiaida01,
        CHTS\n,,0,2022-09-07T09:00:00Z,2022-09-07T09:37:25.412Z,2022-09-07T09:37:25.412Z,0.
        564,avg_response,ping,aida,svmwiaida01,0e72060d-3160-4bc9-9ea9-2940b1990b49,
        svmwiaida01,CHTS\n\n"
}
```

Figure 28: Monitoring data query API call

5.8 External Results

As stated in chapter 1, this Dissertation is part of a larger project in which was also developed a IT monitoring management and visualization web application that integrates and uses the solution developed in this Dissertation. Thus, although the author of this Dissertation did not develop the web application, it was considered relevant to show some of the referred web application interactions, i.e., portions of the application, as a way of also presenting the results of the backend architecture solution and infrastructure developed in this Dissertation since the web application serves as an interaction and visualization tool to simplify the usage of the developed monitoring solution and help end users more easily take advantage of its full potential. Figures 29, 30, 31, 32 and 33 show some screenshots of the web application displaying functionalities such as overview page with the existing monitoring agents, agents' creation and detail pages, as well as the visualization dashboard functionality that takes advantage of the guardian's server data export service.


Create Agent

1 Agent Information — 2 Level

Information

Agent Name *

Agent Group *
default

Organization *

Type *

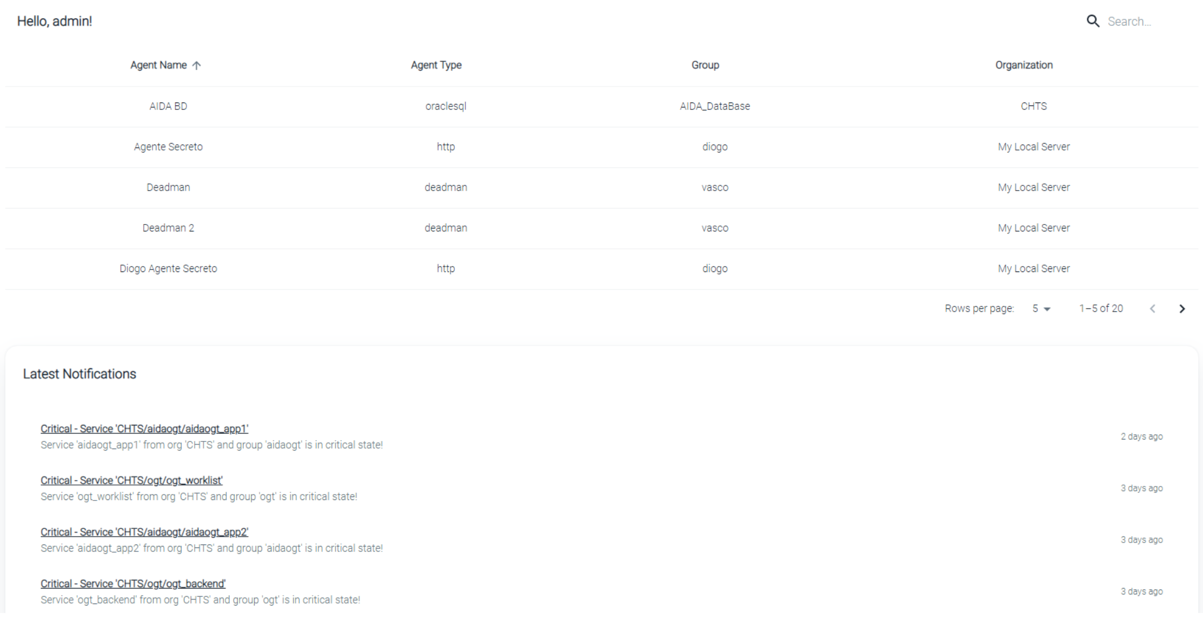
Repetition

Value *
0
Can't be empty and needs to be bigger than 0

Unit *

Back — Next

Figure 29: Agent creation page


Hello, admin!

Search...

Agent Name ↑	Agent Type	Group	Organization
AIDA BD	oraclesql	AIDA_DataBase	CHTS
Agente Secreto	http	diogo	My Local Server
Deadman	deadman	vasco	My Local Server
Deadman 2	deadman	vasco	My Local Server
Diogo Agente Secreto	http	diogo	My Local Server

Rows per page: 5 — 1–5 of 20

Latest Notifications

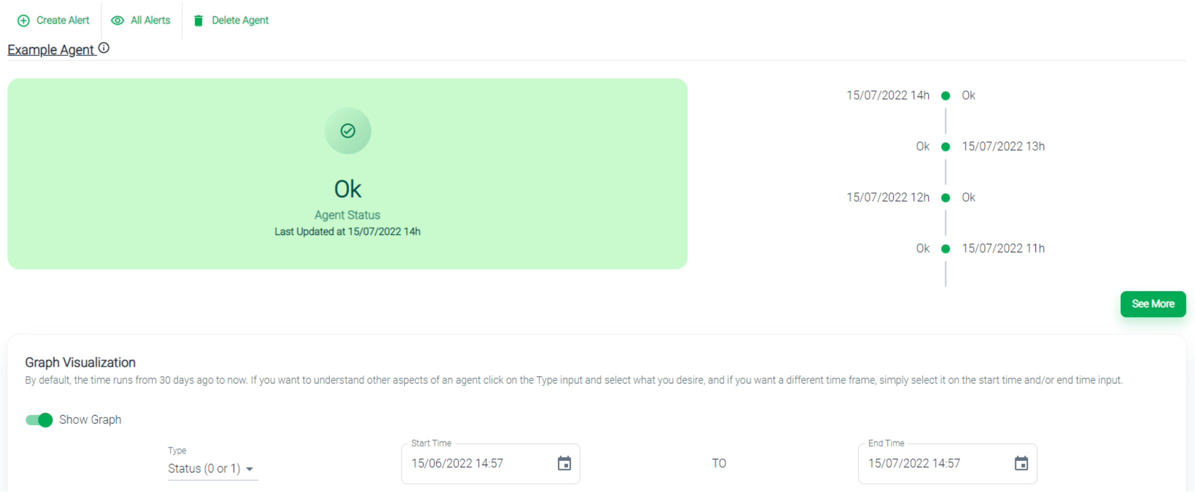
Critical - Service 'CHTS/aidaogt/aidaogt_app1'
Service 'aidaogt_app1' from org 'CHTS' and group 'aidaogt' is in critical state! — 2 days ago

Critical - Service 'CHTS/ogt/ogt_worklist'
Service 'ogt_worklist' from org 'CHTS' and group 'ogt' is in critical state! — 3 days ago

Critical - Service 'CHTS/aidaogt/aidaogt_app2'
Service 'aidaogt_app2' from org 'CHTS' and group 'aidaogt' is in critical state! — 3 days ago

Critical - Service 'CHTS/ogt/ogt_backend'
Service 'ogt_backend' from org 'CHTS' and group 'ogt' is in critical state! — 3 days ago

Figure 30: Agents overview page


Create Alert | All Alerts | Delete Agent

Example Agent

Ok
Agent Status
Last Updated at 15/07/2022 14h

15/07/2022 14h ● Ok
Ok ● 15/07/2022 13h
15/07/2022 12h ● Ok
Ok ● 15/07/2022 11h

See More

Graph Visualization

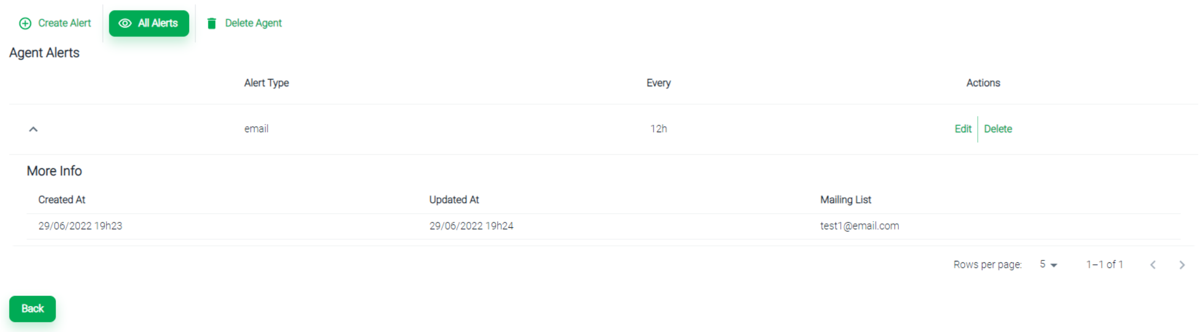
By default, the time runs from 30 days ago to now. If you want to understand other aspects of an agent click on the Type input and select what you desire, and if you want a different time frame, simply select it on the start time and/or end time input.

Show Graph

Type: Status (0 or 1)

Start Time: 15/06/2022 14:57 TO End Time: 15/07/2022 14:57

Figure 31: Agent page



Agent Alerts

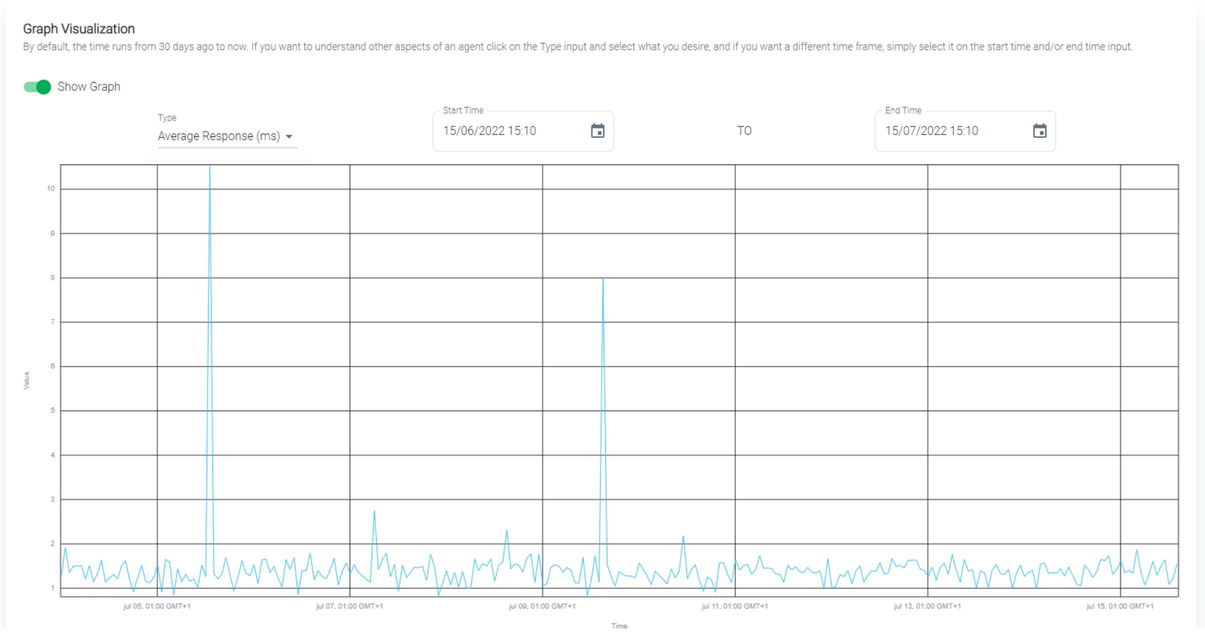
	Alert Type	Every	Actions
^	email	12h	Edit | Delete

More Info

Created At	Updated At	Mailing List
29/06/2022 19h23	29/06/2022 19h24	test1@email.com

Rows per page: 5 1–1 of 1

Figure 32: Agent alerts page


Graph Visualization

By default, the time runs from 30 days ago to now. If you want to understand other aspects of an agent click on the Type input and select what you desire, and if you want a different time frame, simply select it on the start time and/or end time input.

Show Graph

Type: Average Response (ms)

Start Time: 15/06/2022 15:10 TO End Time: 15/07/2022 15:10

Figure 33: Agent's metrics visualization page

5.9 Summary

The base features of the architecture proposed in chapter 4 were successfully implemented. With the current solution, it is possible to monitor and visualize systems' and services' performance and health in a centralized platform that supports multiple target sites and organizations.

The end users and external applications can communicate with the system through HTTP requests to the guardian server, given valid authentication or credentials. Through these requests, it is possible to create and manage monitoring agents as well as its health status levels and alerts, customize the solution with pre-configured notification channels to serve as alerts' base configurations and consume monitoring metrics and information to actively monitor and inspect the monitored services' current and past status and performance. All the available requests are documented and exemplified through openly accessible API documentation.

Each and every service is packaged in a Docker image, which allows and streamlines the distribution and installation of a production environment as well as its maintenance. Finally, the monitoring solution is integrated with an external web application, demonstrating the functionality and potential of the developed monitoring backend and infrastructure solution.

Chapter 6

Validation and Discussion

After the platform has been developed, it was necessary to validate it in a real-world context and environment. Subjective and formal assessment and validation processes were used to demonstrate that the produced technology is capable of adoption and usage in the IT departments of healthcare organizations, as initially planned.

To that effect, first, it was conducted a user acceptance and validation questionnaire that focused on the participants' experience and perspectives regarding the solution's quality and usefulness. Then, it was carried out a two-part formal validation process by conducting a SWOT analysis of the project followed by a systematic process of risk assessment and mitigation.

6.1 Usefulness and Acceptance Validation

The system's testing and validation were carried out through real case studies that were developed in cooperation with the same two healthcare institutions of the initial questionnaire, i.e., CHTS and USL-Guarda. This evaluation focused on whether the target audience, after a familiarization period, accepted and embraced the new system and whether it was useful and had a positive impact on their daily monitoring duties. Thus, a second questionnaire was developed which, like the first one, was directed to the IT and systems administrators and distributed in the form of an anonymous Google form, the questions of which are available in appendix B. The number of participants was inferior when compared to the first questionnaire, with a final result set of 13 responses.

Moreover, as in the first questionnaire, this one is also constituted of two sections: the first section with closed questions and a second with more open ones. The first section focuses on gathering a perspective regarding the usage context (in which platforms, with what regularity, etc) and the responder's perception of the platform's quality (if there was any bug, issue, wrong/unexpected interaction/behavior, etc). On the other hand, the second section focuses on the responder's perception of the system's usefulness, effectiveness and applicability.

Regarding the data preparation and analysis, the same strategies and techniques from the first questionnaire were used. The data from the closed questions were treated as being of the nominal type, allowing a statistical representation of the data, followed by the respective analysis and interpretation. The answers to the open questions were treated individually, manually extracting the topics and suggestions considered most important and relevant for the evaluation and validation of the developed solution.

The first two closed questions are aimed at providing a behavioral and usage context for the system. Figure 34 presents the usage context of the system, that is, the software used to access and interact with the developed system, showing that more than half of the participants engaged with the system through the web application developed by Carolina Marques in the scope of the overall project and that almost 20 percent interacted with the system using terminal calls or programmatic clients (such as Python scripts or Java programs). On the other hand, the second question focuses on how often the participants used the system. As figure 35 shows, a large majority of the participants used the system at least once a day, with a second-highest frequency of usage several times a day, which most possibly means the system was well accepted and perceived by the participants.

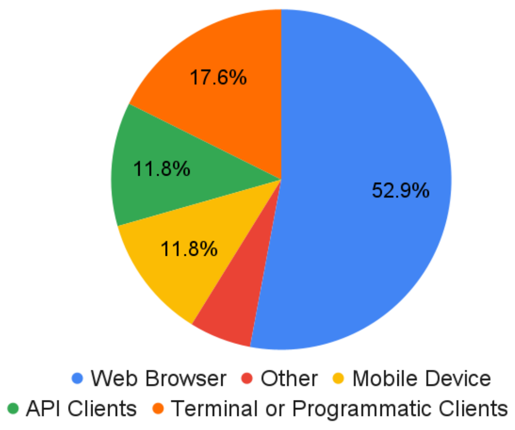


Figure 34: Distribution of the devices/services from which the solution was used and tested

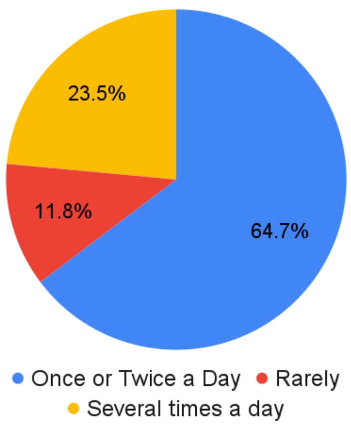


Figure 35: Distribution of the regularity with which the solution was accessed and used

The questionnaire's next three questions focused on quality control and system performance by gathering information on whether the participants had encountered any kind of unexpected or troubling behavior while testing the platform. Figure 36 presents the answers' distribution to the three questions which reveal the system is indeed working as expected, that is, respecting the data isolation policies and the privacy of each organization, while operating consistently and without any kind of error or wrong behavior. Furthermore, the system's performance and responsiveness are more than satisfactory, since from all the participants only one reported having once experienced delays or lags while using the platform. As such, from a quality and performance perspective, it is possible to conclude the system is capable, reliable and meets end-user usage demand and privacy concerns.

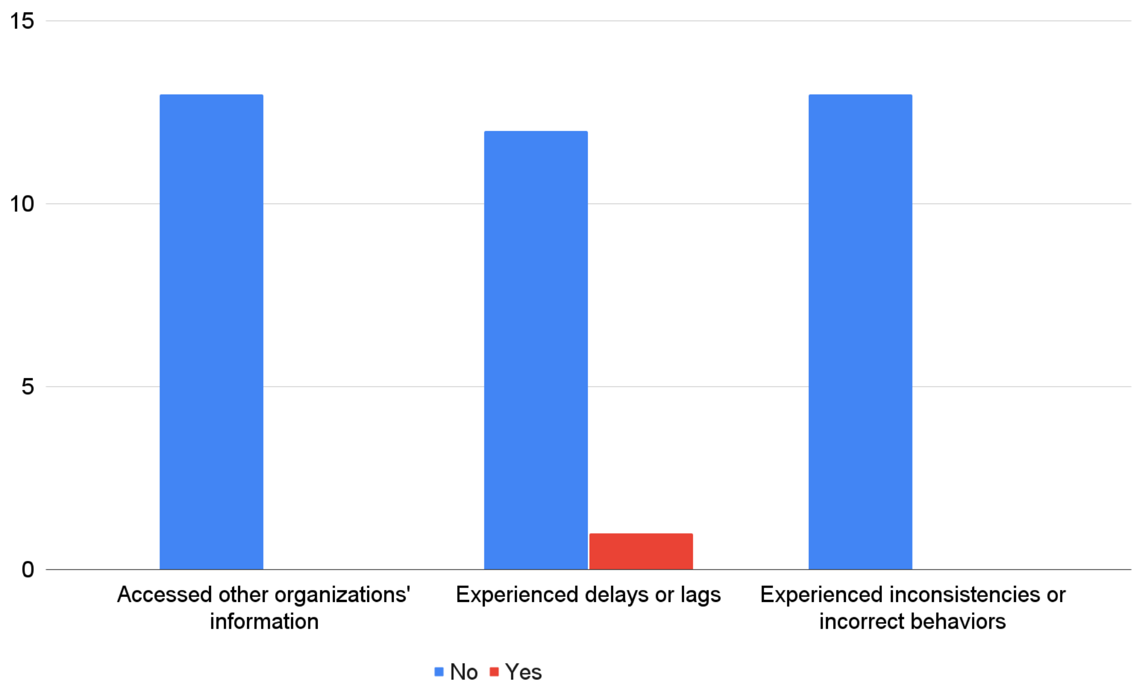


Figure 36: Distribution of responses to the third, fourth and fifth closed questions (see appendix B)

The remaining closed question focused on the system's functionality, whether the system met the participant's expectations and needs and whether was better or more useful than the monitoring solution previously used. This question uses the method of providing statements and measuring the participants' level of agreement, as was in the first questionnaire. The statements presented in the questionnaire are as follows (further detail on appendix B):

1. The monitoring agents' variety is appropriate/sufficient for your organization's monitoring needs.
2. Every time a monitored service failed you were warned per the configurations you specified.
3. The alerting delivery mechanisms are appropriate/sufficient for your organization's setup and needs.
4. The notification alerts provided enough context and information to successfully inform you of the problem at hand.
5. The deployment of the local server (i.e., the data collection clients) was simple and straightforward and occurred with zero or minor installation problems.
6. The monitoring platform, as a whole, simplified the monitoring management and consumption in your organization.
7. The monitoring platform, as a whole, fulfilled your organization's needs better than the previously used monitoring solution.

Figure 37 presents the answers' distribution to this question statements. The responses were treated with the same labeling and scoring mechanism used in the first questionnaire, represented in table 2.

The overall results indicate that the system presented a satisfiable set of functionalities with some shortcomings, namely the diversity of the monitoring agents' types available (statement one). According to the participant's responses, the system's current strongest advantages are the deployment simplicity and good alerting capabilities (statements three, four and five). Finally, given the good scores on statements six and seven, which referred to the system's capability of meeting the needs of users and outperforming previous solutions, it is possible to conclude that the system successfully addresses the problems and necessities that were central to this Dissertation and because feedback on its capabilities and performance, when compared to other monitoring solutions, was also positive, it is also possible to conclude that the system has a good chance of being adopted by more healthcare facilities which is also a good indicator of its usefulness and quality.

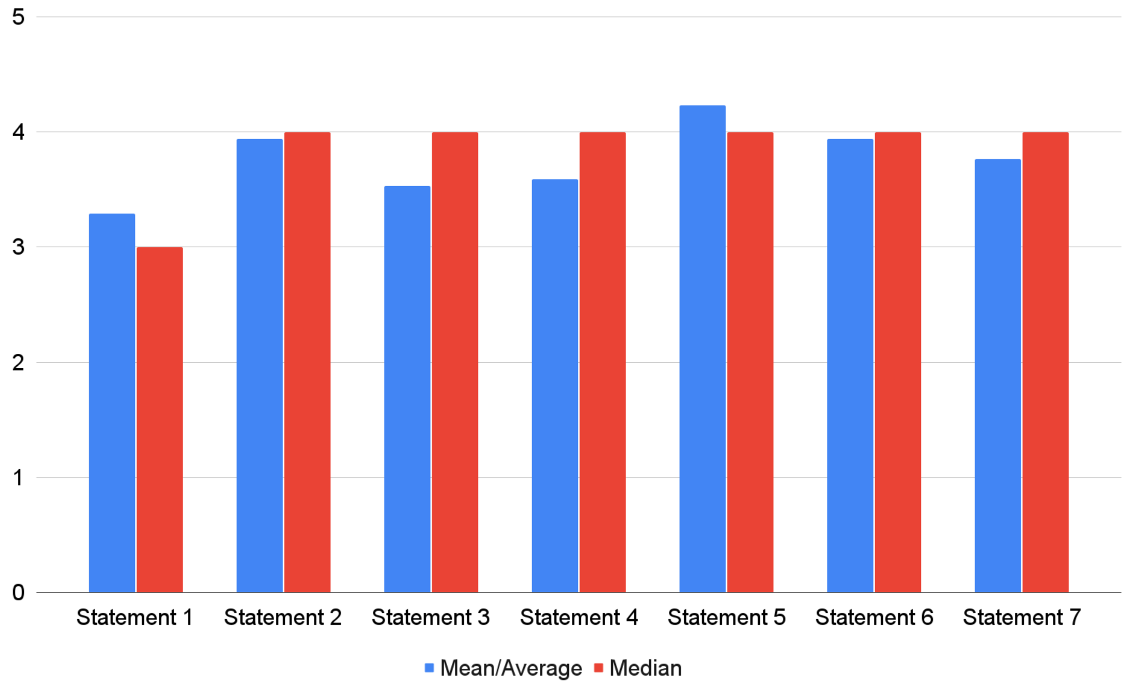


Figure 37: Distribution of responses to the statements of the questionnaire's sixth closed question (see appendix B)

The final two questions, which are open-ended, were focused on understanding which functionalities the participants think the system is missing and also how the current functionalities can be improved and expanded. Most of the answers were related to increasing the diversity of the monitoring agents and alerting delivery mechanisms available. This was already expected and planned to be improved in future work outside of this Dissertation scope and execution, as it is going to be detailed in the next chapter.

The feedback from the participants was generally very positive, as all of the indicators resulting from the questionnaire's responses pointed to the system's success, both in terms of its ability to meet the problems and concerns that originated this Dissertation, as well as the ability to meet the needs and expectations of the IT administrators who tested it, and also in terms of its positive behavior when compared to other solutions already on the market.

As a final observation, these results allow an important conclusion, which is the answer to the guiding research question that originated this Dissertation's work. The question intended to understand if a multi-device/site and multi-organization monitoring solution would help improve the availability and, consequently, quality of the highly heterogeneous IT infrastructures of the healthcare industry. All results indicate a positive answer and, as such, corroborate this Dissertation's initial hypotheses as well as the success of this solution's architectural design and implementation.

6.2 Formal Evaluation

While validation by assessing the feedback of participants on real case studies is very insightful and helpful in assessing the project's outcome and success, it is also important to conduct a formal evaluation of the project and developed system using standard and well-tested project evaluation and analysis frameworks. This section presents the results of the two formal evaluation frameworks used: SWOT Analysis and Risk Assessment.

6.2.1 SWOT Analysis

The SWOT analysis is a strategic planning framework used when performing a critical evaluation of an organization, a project, a system or a business activity. SWOT analysis has two dimensions: internal and external. The internal dimension is concerned with internal and organizational factors, i.e., the system's **Strengths** and **Weaknesses**, and the external dimension is concerned with the environmental factors, i.e., the market's **Opportunities** and possible **Threats** [59]. This framework is frequently represented as a 2-by-2 matrix similar to the one in figure 38.

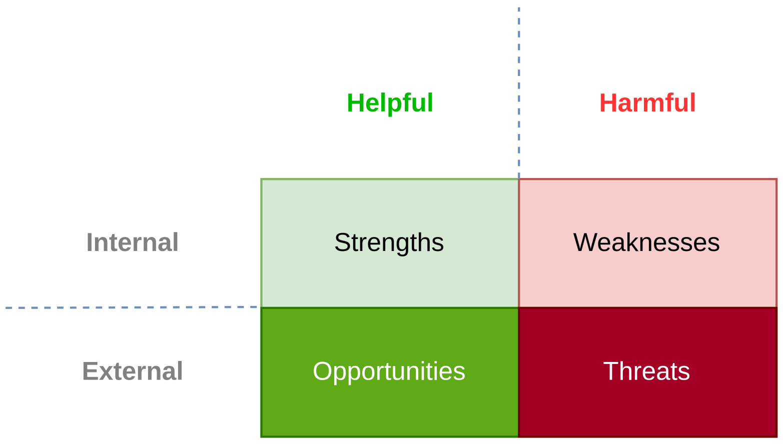


Figure 38: SWOT Matrix. Adapted from: [60]

It is clear, then, that each SWOT analysis is composed of these four categories:

- **Strengths** is what an organization or system excels at and what separates it from the competition.
- **Weaknesses** is what stops an organization or system from achieving its full potential and, therefore, are the areas that need to be worked on and improved.
- **Opportunities** are beneficial external factors that could give an organization or system a competitive advantage.
- **Threats** are external factors that have a negative impact and the potential to harm or prejudicate a system or an organization.

Regarding the case study at hand, taking into account the theoretical and practical components of this evaluation framework and also using the information obtained from the platform's usage in the cooperating healthcare facilities, it was carried out a SWOT analysis of the developed monitoring platform. The results of this analysis are listed below.

Strengths

- High scalability and availability of the platform as well as simple deployment process;
- Information security through authentication and data isolation mechanisms;
- Support for multi-site/facility and multi-organization monitoring with a unified and centralized management system as well as the possibility to have a unified view (through external dashboard clients) of everything related to monitoring (agents and alerts' configuration, services' status, notifications, visualization dashboards, etc);
- Modular system, which can be continuously improved, based on the structure initially created.

Weaknesses

- The current variety in terms of agent types and/or alerting delivery mechanisms available may be limited or insufficient for certain usage scenarios;
- The platform's use is entirely dependent on a network/internet connection;

Opportunities

- The demand in healthcare to have a simpler but capable and efficient monitoring system, possible to be configured and managed entirely over a web page or HTTP requests;
- The democratization of monitoring systems in healthcare can lead to broader and extensive adoption of monitoring techniques, improving the organizations' IT systems and healthcare services' quality, resulting in a better experience for both the patients and the healthcare professionals.

Threats

- Although the user validation questionnaire results showed a good acceptance and understanding of the platform, when expanding to other healthcare facilities new problems and unexpected reactions from users can lead to unforeseen problems or complications;
- Healthcare organizations, as well as its IT teams, may be hesitant to adopt a newer platform to detriment of their current IT monitoring platform or even other monitoring tools more well-established in the market, leading to an adoption problem.

Table 3 presents a summary of the aforementioned results in the format of a SWOT matrix.

Table 3: SWOT analysis - results summary matrix

Strengths	**Weaknesses**
- Scalability and availability	- Limited variety of agent types
- Simple deployment	- Limited variety of alerting delivery mechanisms
- Information security	- Network connection
- Multi-site and multi-organization support	
- Modularity	
Opportunities	**Threats**
- Product demand	- Acceptance/usability problems
- Improve healthcare services' quality	- Lack of adoption

6.2.2 Risk Assessment

A risk assessment analysis is a systematic procedure for finding, assessing, and controlling hazards and risks. This systematic process has a set of phases. First, enumerate all possible risks and problems that may occur. Second, when the risks are identified, evaluate each risk according to the likelihood of occurrence, its potential impact and the severity of the risk, which is the product of the two previous indicators, i.e., likelihood and consequence. And finally, prepare a mitigation action plan to minimize the impact of each risk [61].

To simplify the risk assessment process, the three evaluation indicators will be represented with a number scale system. For the likelihood of occurrence, the following scale will be used:

1. **Rare:** it's possible to occur, although extremely unlikely;
2. **Unlikely:** it could occur, but it is unlikely;
3. **Possible:** there is a real possibility of the risk happening;
4. **Probable:** a risk occurrence is likely to occur more than once;

5. **Almost certain:** a risk occurrence is very (maybe extremely) likely to occur;

For the consequence of the risk, the following scale will be used:

1. **Insignificant:** none or near-zero impact;
2. **Minor:** short-term disruption;
3. **Moderate:** significant (mid-term) disruption;
4. **Major:** highly disruptive (mid to long-term);
5. **Catastrophic:** possibly irreversible (long-term);

Since the severity indicator is the product of the previous two, it can take values from one (1) to twenty-five (25).

The results of this assessment are represented in the table 4. The likelihood indicator of each risk is represented by the column **L**, the consequence indicator by the column **C** and the severity one by the column **S**. There is also an extra column indicating if each risk has indeed occurred during the course of this work.

Table 4: Risk assessment results

Risk	**L**	**C**	**S**	**Occurred?**	**Mitigation Action Plan**
Lack of responses to the research questionnaire.	4	4	16	No	If possible, ask friends and colleagues to answer them. Otherwise, try to find other people who can fill the role and confirm their availability.
The obtained results were unsatisfactory.	3	5	15	No	Understand why this happened and try to fix it. If extra time is needed, ask to postpone the artifact's expected delivery date.
Lack of experience or knowledge with the research topics.	4	3	12	Yes	Research documentation, articles, tutorials, books and other learning materials that could aid in understanding and improving the overall development. Ask for suggestions from the advisor and/or colleagues.

Table 4: Risk assessment results (continued)

Risk	**L**	**C**	**S**	**Occurred?**	**Mitigation Action Plan**
Issues or bugs with the solution.	4	3	12	Yes	Issues or bugs on a production installation although undesired, are very common. Identify the problem and analyze its impact. If needed, roll back to a previous stable version while the problem is being fixed.
The findings and/or questionnaire results did not support the user feedback.	3	3	9	No	Hypotheses are sometimes proposed but they are not always proven. This is still a result, so why it didn't work should be discussed.
The research topics are extremely complex or have not been comprehensively studied.	3	3	9	No	Divide a complex topic into less complicated subtopics. Investigate them and try to connect them at the end.
The chosen methodologies and/or technologies were inadequate.	2	4	8	No	Look for other available methodologies/technologies and select one that is appropriate for the project. Consult with the advisor.
The deployment provisioning was insufficient.	3	2	6	No	Try to understand what are the current bottlenecks and address them by updating the underlying infrastructure.
Difficulties communicating with the adviser.	1	4	4	No	Encourage weekly meetings with the advisor and make use of appropriate communication channels.
Inadequate planning.	1	4	4	No	Rethink a more realistic plan to ensure it meets expectations.

6.3 Summary

This chapter explored the evaluation and validation of the system as well as a subsequent discussion of the acquired results. The case study had the collaboration of the healthcare organizations, CHTS and USLGuarda, and aimed to understand how the developed solution was perceived and used by IT administrators in real healthcare environments to validate whether the system successfully addresses the problems, concerns and requirements discussed in the early phases of this Dissertation.

The system was validated and evaluated through two types of analysis, formal and subjective. The subjective analysis consisted of an anonymous online questionnaire. And the formal evaluation consisted of a SWOT analysis followed by a risk assessment and mitigation analysis. Both the questionnaire and formal methods produced positive results with a big emphasis on the system's ability to overcome the problems and difficulties related to multi-organization and multi-site IT monitoring that were at the center of this Dissertation's exploration topics and the overall participant's acceptance of the system and willingness to use it in detriment of the organizations' previously adopted monitoring solutions.

While the feedback and subsequent evaluation results were positive, some shortcomings and limitations were identified, mainly related to the limited variety of types of monitoring agents and alert delivery mechanisms available. These limitations were already known by the author of this Dissertation and are addressed through future work planning discussed in more detail in the next chapter.

Chapter 7

Conclusion

This Dissertation is concluded with some closing notes on the work presented throughout this document. A summary of this Dissertation's key technical and scientific contributions is also presented, followed by a highlight of suggestions for future improvements.

7.1 Final Remarks

This Dissertation introduced a novel IT monitoring backend solution tailored specifically for the healthcare industry. Based on the existing solutions explored, four main limitations were found:

- Only a small number of the solutions simultaneously support the two existing data collection paradigms (pull-based and push-based);
- Most solutions do not support API-based monitoring management;
- Only one of the analyzed solutions supported multi-organization and multi-site monitoring schemas, albeit not fully;
- All the explored solutions had at least one of the three aforementioned limitations and, therefore, none of them completely suppressed the capabilities needed to overcome the problem addressed in this Dissertation.

The solution, and respective architecture, that was designed and validated in this work overcomes those limitations and offers full API-based IT monitoring management and consumption, with a fully compliant multi-site and multi-organization integration schema and a hybrid data collection mechanism that supports both paradigms. To accomplish this, it was selected a microservices-based approach where the main components of the solution are made available from an off-site installation, named guardian server, and an on-site local server responsible for the organization's monitoring data collection. The final approach and design were heavily influenced by the first questionnaire's results and suggestions, which was a fundamental instrument of technical and conceptual feedback.

To validate the solution, different validation and evaluation strategies were applied. First, the solution was tested in real-life scenarios, namely two regional hospitals, CHTS and USLGuarda. Following the testing period, the IT administrators that participated in the testing were allowed to evaluate the solution through an anonymous online form. The results of the evaluation were positive and the solution was accepted as a viable monitoring option that overcomes the referred limitations found in existing solutions. One relevant finding was that the majority of participants, when asked, stated that the developed solution simplified their day-to-day monitoring tasks and better fulfilled their organization's monitoring needs than their previous monitoring solution, which means this Dissertation's work is an improvement both in terms of functionality and simplicity when compared to the solutions each organization was using. In addition, were also conducted two formal evaluations, a SWOT analysis and a systematic process of risk assessment and mitigation. Both were useful to understand more pragmatically the solution's positive aspects and limitations, as well as identify possible risks and a plan to mitigate them.

Some challenges had to be overcome during this Dissertation to arrive at this final architecture. The overall architecture design took several iterations to arrive at the presented one mainly due to network and security constrictions that needed to be overcome so the solution could be used in real-life healthcare organizations. Also, the deployment and documentation efforts were surmountable, since the goal was to make the installation, adoption and usage processes as simple and seamless as possible. Because of this, and also the concern to make the solution's implementation as clean and understandable as possible, much time was dedicated to refactoring and refining details and achieving the current result.

It is also important to note that, although the solution was designed to focus on the healthcare sector, its architecture and implementation are generic enough to allow the system to be used by other industries that experience the same problems and difficulties explored and addressed by this work.

Overall, it can be concluded that it was possible to address and overcome the limitations and problems discussed in chapters 1 and 2. It was also possible to understand, from the validation process and results, that a multi-device/site and multi-organization monitoring solution can, indeed, improve the availability and quality of healthcare IT infrastructure and, consequently, their healthcare service's quality, which was this Dissertation's guiding research question.

7.2 Contributions

This Dissertation's work contributes with:

- A literary and state-of-the-art review of IT monitoring solutions with application to the healthcare sector and its landscape regarding support for multi-site and multi-organization functionalities.

- A microservices-based IT monitoring backend that fully supports all the defined and specified functional and non-functional requirements with particular emphasis on multi-site and multi-organization functionalities, as well as web and API-based monitoring and real-time alerts configuration and consumption.

- The implementation of the designed IT monitoring backend architecture using open-source technologies and frameworks.

- The backend solution's full documentation, including usage examples, and deployment guides, so it can be easily extended, improved and deployed in countless healthcare organizations, no matter their size, complexity and production environment.

Additionally, this Dissertation's work also has scientific outcomes, namely a conference publication and the submission of a detailed implementation and case study results article to an International Conference on Information Technology & Systems with the following references, respectively:

- C. Marques, V. Ramos, H. Peixoto, and J. Machado, "Pervasive Monitoring System for Services and Servers in Healthcare Environment", Procedia Computer Science, vol. 201, pp. 720–725, 2022, The 5th International Conference on Emerging Data and Industry 4.0 (EDI40). More details on appendix D.1.

- V. Ramos, C. Marques, H. Peixoto, and J. Machado, "Information Technology Monitoring in Healthcare: A Case Study", in The 6th International Conference on Information Technology & Systems (ICITS23), Springer International Publishing (under review). More details on appendix D.2.

7.3 Future Work

Since this work had to address significantly different challenges to develop an integrated solution, there was no opportunity to further refine and improve all elements, resulting in implementation prioritization. The resulting solution may and should be used to leverage additional functionalities that this Dissertation could not cover. Thus, taking into consideration the ideas that were regarded interesting but not a priority, as well as the results and comments from the validation questionnaire, the following features and improvements were considered relevant for future work:

1. *Extend monitoring agents support*. The implemented solution makes available a core set of monitoring agent types. To enable the solution to serve a larger variety of organizations, healthcare or not, it would be interesting to explore and implement additional monitoring agent types, as well as improve the current agent type model to further simplify the process of adding new agent types to the solution.

2. *Extend alerting delivery mechanisms support*. Every organization has a different set of communication and technological tool stacks. Thus, although the current solution already provides a large option set of alerting delivery mechanisms, it would be interesting to further expand it and allow even more integrations, such as SMS, PagerDuty, WhatsApp, Telegram, and Signal, among others.

3. *Extend data export standards support*. The solution's data export functionality is aimed at simplifying the integration of this system with other external monitoring systems or visualization dashboards. Since integration and interoperability is key in healthcare, it should be interesting to understand how this functionality could be expanded to support additional data metrics standards, namely OpenMetrics and OpenTelemetry, which are the two fastest-growing projects in this area of expertise [62].

4. *Improve real-time alerting*. Real-time alerting is an existing functionality of the developed solution. However, its implementation was based on a Socket.IO web server configuration, which, despite allowing quick implementation of this type of mechanism, is not the most reliable, fault-tolerant or efficient option. As such, it should be investigated the possibility to replace the Socket.IO implementation with a combination of a message queue, e.g. RabbitMQ, ZeroMQ, and others, with WebSocket clients [63], [64].

5. *Improve deployability and scalability*. Considerable efforts were put into developing a production installation process that would be simple and reliable enough to follow without major operational issues. Nevertheless, since more and more companies are migrating to the cloud and focusing on cloud-native solutions, it would be interesting to explore cloud and container-based orchestration tools such as Kubernetes and Helm[1].

6. *Include troubleshooting/recovery support*. Integrated troubleshooting or recovering suggestions is a major asset to every monitoring solution since it allows the IT administrators to easily understand what is the problem, by looking into the monitoring metrics, while also being given some insight on what might be needed to do to troubleshoot the problem or, how to recover the system from the problem occurrence. This could be implemented by starting with generic insights or command executions via a command-like experience, either by web interface or chat-based (e.g. WhatsApp, Slack, etc) integrations, that enables actions like restarting or resetting the service and, afterward, evolve this functionality into a fully functional troubleshooting and recovery mechanism.

[1]Helm is a Kubernetes deployment tool that helps automate the design, packaging, configuration, and deployment of Kubernetes applications and services [65].

Bibliography

[1] E. A. for Digital Transition, *The Digital Transition in Healthcare: An Urgent Need - Manifesto*, [Accessed 14. Dec. 2021]. [Online]. Available: https://digitalforeurope.eu/the-digital-transition-in-healthcare-an-urgent-need (cit. on p. 1).

[2] J. Øvretveit, T. Scott, T. G. Rundall, S. M. Shortell, and M. Brommels, "Improving quality through effective implementation of information technology in healthcare", *International Journal for Quality in Health Care*, vol. 19, no. 5, pp. 259–266, Aug. 2007, issn: 1353-4505. doi: 10.1093/intqhc/mzm031. [Online]. Available: https://doi.org/10.1093/intqhc/mzm031 (cit. on p. 1).

[3] W. H. Organization, *Toolkit on monitoring health systems strengthening*, [Accessed 14. Dec. 2021], 2008. [Online]. Available: https://www.who.int/healthinfo/statistics/toolkit_hss/EN_PDF_Toolkit_HSS_InformationSystems.pdf (cit. on p. 1).

[4] *5 reasons to build a health monitoring system for a hospital*, https://cprimestudios.com/blog/5-reasons-build-health-monitoring-system-hospital, [Accessed 8. Oct. 2021] (cit. on p. 2).

[5] A. Olson, *Avi Systems – Why Health Care Organizations Need a Monitoring and Reporting System*, https://www.avisystems.com/blog/why-health-care-organizations-need-a-monitoring-and-reporting-system, [Accessed 8. Oct. 2021] (cit. on p. 2).

[6] E. Pettersson, "A Comparison of Pull- and Push- based Network Monitoring Solutions: Examining Bandwidth and System Resource Usage", M.S. thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2021 (cit. on p. 6).

[7] G. Statkeviius, *Push Vs. Pull In Monitoring Systems*, https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems, [Accessed 19. Oct. 2021] (cit. on p. 6).

[8] M. Julian, *Practical Monitoring: Effective Strategies for the Real World*. O'Reilly Media, 2018, isbn: 978-1-491-95735-6 (cit. on pp. 6, 7).

[9] Y. I. Binev, "Centralised Monitoring and Alerting Solution for Complex Information Management Infrastructure", M.S. thesis, NOVA Information Management School (NIMS), Jan. 2020 (cit. on pp. 6, 10, 12).

[10] J. Turnbull, *The Art of Monitoring*. James Turnbull & Turnbull Press, 2016, isbn: 978-0988820241 (cit. on pp. 6, 7).

[11] *Start page – collectd – The system statistics collection daemon*, [Accessed 20. Oct. 2021]. [Online]. Available: https://collectd.org (cit. on p. 7).

[12] H. Huang and L. Wang, "P&P: A Combined Push-Pull Model for Resource Monitoring in Cloud Computing Environment", in *2010 IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 260–267. doi: 10.1109/CLOUD.2010.85 (cit. on p. 7).

[13] A. Srivastava, *Elasticsearch 7 quick start guide: get up and running with the distributed search and analytics capabilities of Elasticsearch*. Packt Publishing, 2019, isbn: 9781789803327 (cit. on p. 8).

[14] *What are Beats? | Beats Platform Reference | Elastic*, [Accessed 20. Oct. 2021]. [Online]. Available: https://www.elastic.co/guide/en/beats/libbeat/current/beats-reference.html (cit. on p. 8).

[15] *Elastic Sack Customers and Success Stories*, [Accessed 1. Dec. 2021]. [Online]. Available: https://www.elastic.co/customers (cit. on p. 9).

[16] *TICK Stack | Technology Radar | Thoughtworks*, [Accessed 20. Oct. 2021]. [Online]. Available: https://www.thoughtworks.com/radar/platforms/tick-stack (cit. on p. 9).

[17] P. Dix, *Monitoring with Push vs. Pull: InfluxDB Adds Pull Support with Kapacitor*, https://www.influxdata.com/blog/monitoring-with-push-vs-pull-influxdb-adds-pull-support-with-kapacitor, [Accessed 20. Oct. 2021] (cit. on p. 9).

[18] *Open Source Time Series Platform - The TICK Stack*, [Accessed 20. Oct. 2021]. [Online]. Available: https://www.influxdata.com/time-series-platform (cit. on p. 10).

[19] *Telegraf plugins | Telegraf 1.20 Documentation*, [Accessed 21. Oct. 2021]. [Online]. Available: https://docs.influxdata.com/telegraf/v1.20/plugins (cit. on p. 10).

[20] *InfluxDB OSS 1.8 Documentation*, [Accessed 21. Oct. 2021]. [Online]. Available: https://docs.influxdata.com/influxdb/v1.8 (cit. on p. 10).

[21] I. Data, *An InfluxData Case Study: How Allscripts Uses InfluxDB to Monitor Its Healthcare IT Platform*, [Accessed 12. Dec. 2021], 2021. [Online]. Available: https://get.influxdata.com/rs/972-GDU-533/images/Customer_Case_Study_Allscripts.pdf (cit. on p. 11).

[22] *Who uses TICK Stack?*, [Accessed 28. Nov. 2021]. [Online]. Available: https://stackshare.io/influxdb (cit. on p. 11).

[23] Ł. Kugel, "Tools for distributed systems monitoring", *Foundations of Computing and Decision Sciences*, vol. 41, Dec. 2016. doi: 10.1515/fcds-2016-0014 (cit. on p. 11).

[24] A. Castanheira, "Implementação de Sistema de Monitorização e Controlo de Interoperabilidade Clínica", M.S. thesis, University of Minho (UM), Mar. 2020 (cit. on p. 11).

[25] *Nagios Solutions For Healthcare and Medicine*, [Accessed 1. Nov. 2021]. [Online]. Available: https://www.nagios.com/solutions/healthcare-medicine (cit. on p. 11).

[26] *Who uses Nagios?*, [Accessed 2. Dec. 2021]. [Online]. Available: https://stackshare.io/nagios (cit. on p. 11).

[27] *Nagios — Architecture*, [Accessed 26. Oct. 2021]. [Online]. Available: https://www.tutorialspoint.com/nagios/nagios_architecture.htm (cit. on p. 11).

[28] *Overview | Prometheus*, [Accessed 21. Oct. 2021]. [Online]. Available: https://prometheus.io/docs/introduction/overview (cit. on pp. 12, 13).

[29] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*, First edition. O'Reilly Media, 2018, isbn: 9781492034148 (cit. on p. 12).

[30] *Who uses Prometheus?*, [Accessed 4. Dec. 2021]. [Online]. Available: https://stackshare.io/prometheus (cit. on p. 13).

[31] U. o. O. Network Startup Resource Center, *Scalable monitoring tools – a mile-high view*, https://nsrc.org/workshops/2019/btnog6/nmm/netmgmt/en/futures/scalable-monitoring-tools.pdf, [Accessed 20. Oct. 2021], 2019 (cit. on p. 14).

[32] *Monitoring a multi-cluster environment using Prometheus federation and Grafana*, [Accessed 26. Oct. 2021]. [Online]. Available: https://mattermost.com/blog/monitoring-a-multi-cluster-environment-using-prometheus-federation-and-grafana (cit. on p. 15).

[33] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture", Apr. 2020, pp. 150–153. doi: 10.1109/MEMSTECH49584.2020.9109514 (cit. on pp. 16, 17).

[34] G. F. F. da Cunha, "Data Analysis and Recommender System Architecture for E-Commerce Platforms", M.S. thesis, University of Minho (UM), Jan. 2021 (cit. on pp. 16, 17).

[35] *Microservices vs Monolithic architecture | MuleSoft*, [Accessed 28. Oct. 2021]. [Online]. Available: https://www.mulesoft.com/resources/api/microservices-vs-monolithic (cit. on p. 16).

[36] A. Kharenko, *Monolithic vs. Microservices Architecture*, [Accessed 28. Oct. 2021]. [Online]. Available: http://www.antonkharenko.com/2015/09/monolithic-vs-microservices-architecture.html (cit. on pp. 16, 17).

[37] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures", in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 000 149–000 154. doi: 10.1109/CINTI.2018.8928192 (cit. on p. 17).

[38] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study", *Software: Practice and Experience*, vol. 50, no. 9, pp. 1651–1681, 2020. doi: 10.1002/spe.2869 (cit. on p. 18).

[39] P. Siriwardena and N. Dias, *Microservices Security in Action*. Manning, 2020, isbn: 9781617295959 (cit. on p. 18).

[40] *What is API security?*, [Accessed 2. Nov. 2021]. [Online]. Available: https://www.redhat.com/en/topics/security/api-security (cit. on p. 18).

[41] *JSON Web Tokens —Intro*, [Accessed 3. Nov. 2021]. [Online]. Available: https://jwt.io/introduction (cit. on p. 18).

[42] N. Madden, *API Security in Action*. Manning, 2020, isbn: 9781617296024 (cit. on pp. 18, 19).

[43] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research", *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007. doi: 10.2753/MIS0742-1222240302 (cit. on pp. 22, 23, 26).

[44] S. Mcleod, *Questionnaire: : Definition, Examples, Design and Types*, [Accessed 9. Jun. 2022], Feb. 2018. [Online]. Available: https://www.simplypsychology.org/questionnaires.html (cit. on p. 24).

[45] S. Khan, *What Makes JavaScript So Popular*, [Accessed 1. Dec. 2021]. [Online]. Available: https://generalassemb.ly/blog/what-makes-javascript-so-popular (cit. on p. 25).

[46] *About | Node.js*, [Accessed 1. Dec. 2021]. [Online]. Available: https://nodejs.org/en/about (cit. on p. 25).

[47] E. Brown, *Web development with Node and Express*. Sebastopol, CA: O'Reilly Media, 2014, isbn: 978-1491949306 (cit. on p. 25).

[48] A. Sharma, *Why You Should Use TypeScript for Developing Web Applications*, [Accessed 1. Dec. 2021]. [Online]. Available: https://dzone.com/articles/what-is-typescript-and-why-use-it (cit. on p. 25).

[49] *Why Use MongoDB and When to Use It?*, [Accessed 2. Dec. 2021]. [Online]. Available: https://www.mongodb.com/why-use-mongodb (cit. on p. 25).

[50] S. McLeod, *Likert Scale Definition, Examples and Analysis*, [Accessed 26. Dec. 2021], 2019. [Online]. Available: https://www.simplypsychology.org/likert-scale.html (cit. on p. 30).

[51] *What is Non-Functional Requirement in Software Engineering? Types and Examples*, [Accessed 29. Nov. 2021]. [Online]. Available: https://www.guru99.com/non-functional-requirement-type-example.html (cit. on p. 33).

[52] M. Monsalve, *Lecture notes in Object-Oriented Software Develpment: The Domain Model*, [Accessed 6. Jul. 2022], Apr. 2015. [Online]. Available: https://homepage.cs.uiowa.edu/~tinelli/classes/022/Spring15/Notes/chap9.pdf (cit. on p. 34).

[53] C. Richardson, *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019, isbn: 9781617294549 (cit. on p. 36).

[54] *What is a reverse proxy? | Proxy servers explained*, [Accessed 29. Nov. 2021]. [Online]. Available: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy (cit. on p. 36).

[55] *What is Docker? | IBM*, [Accessed 13. Aug. 2022]. [Online]. Available: https://www.ibm.com/ae-en/cloud/learn/docker (cit. on p. 38).

[56] *What is Cron Job? - Cron Jobs and Scheduled Tasks - Hivelocity Hosting*, [Accessed 11. Aug. 2022]. [Online]. Available: https://www.hivelocity.net/kb/what-is-cron-job (cit. on p. 51).

[57] *Use multi-stage builds*, [Accessed 13. Aug. 2022]. [Online]. Available: https://docs.docker.com/develop/develop-images/multistage-build (cit. on p. 55).

[58] *Overview of Docker Compose*, [Accessed 13. Aug. 2022]. [Online]. Available: https://docs.docker.com/compose (cit. on p. 56).

[59] E. Gürel, "SWOT Analysis: A Therotical Review", *Journal of International Social Research*, vol. 10, pp. 994–1006, Aug. 2017. doi: 10.17719/jisr.2017.1832 (cit. on p. 66).

[60] L. G. Fine, *The SWOT Analysis: Using your Strength to overcome Weaknesses, Using Opportunities to overcome Threats*. CreateSpace Independent Publishing Platform, Oct. 2009, isbn: 978-1-44954675-5 (cit. on p. 66).

[61] M. Mayernik, K. Breseman, R. Downs, R. Duerr, A. Garretson, and C.-Y. Hou, "Risk Assessment for Scientific Data", *Data Science Journal*, vol. 19, Mar. 2020. doi: 10.5334/dsj-2020-010 (cit. on p. 68).

[62] B. Varshney, *OpenMetrics vs OpenTelemetry - A guide on understanding these two specifications | SigNoz*, [Accessed 19. Aug. 2022], May 2022. [Online]. Available: https://signoz.io/blog/openmetrics-vs-opentelemetry (cit. on p. 75).

[63] M. O'Riordan, "Message queues: the right way to process and transform realtime messages", *Ably Blog*, Jul. 2022, [Accessed 19. Aug. 2022]. [Online]. Available: https://ably.com/blog/message-queues-the-right-way (cit. on p. 75).

[64] R. Tandon, "System Design: Lessons From Netflix's Notification Service Design", *Ravi's System Design Newsletter*, Nov. 2021, [Accessed 19. Aug. 2022]. [Online]. Available: https://ravisystemdesign.substack.com/p/system-design-lessons-from-netflixs (cit. on p. 75).

[65] *What is Helm | Helm Documentation*, [Accessed 19. Aug. 2022]. [Online]. Available: https://helm.sh/docs (cit. on p. 75).

Appendix A

Initial Questionnaire

Monitoring System for Services and Servers in Healthcare Environment - Survey

This survey comes in the course of research work in the context of a master's dissertation related to monitoring of services and infrastructures in healthcare.

Its objective is to try to understand the current status of monitoring services and their use, particularly in the healthcare environment, as well as to understand to what extent monitoring services can simplify systems/infrastructure management processes and also in what way can this happen.

* Required

1. Do you, or your IT department, actively use monitoring systems for your services? *

   ☐ Yes ☐ No

2. **Please, classify the following statements.** Number each statement according to your perspective on it, from 0 to 4. * [0 - Strongly Disagree 1 - Disagree 2 - Neutral 3 - Agree 4 - Strongly Agree]

   a. Currently, whether there is a failure or not, you can observe and control the state of your system.

0	1	2	3	4

Strongly Disagree Strongly Agree

   b. When there is a failure in the system, that failure is noticeable.

0	1	2	3	4

Strongly Disagree Strongly Agree

   c. When a failure in a system is detected, you can determine what the implications are and what other services are impacted.

0	1	2	3	4

Strongly Disagree Strongly Agree

   d. A monitoring system is important for systems management.

0	1	2	3	4

Strongly Disagree Strongly Agree

e. A good monitoring system makes it possible to improve the availability of systems.

0	1	2	3	4

Strongly Disagree Strongly Agree

f. A monitoring system capable of analyzing and displaying, in an integrated way, the status of different targets and organizations, streamlines the systems management process.

0	1	2	3	4

Strongly Disagree Strongly Agree

3. Are you familiar with the detection time in the event of a failure? * (If yes, proceed to the next two questions; otherwise, skip ahead to question 6.)

   ☐ Yes ☐ No

4. In your current infrastructure, what is the average failure detection time, in minutes?

5. In your current infrastructure, what is the average response time to a detected failure, in minutes?

6. In your opinion, what constitutes a good monitoring system? *

Thank you.

Appendix B

Final Questionnaire

Monitoring System for Services and Servers in Healthcare Environment - User Validation

The objective of this questionnaire is to validate and assess if the developed platform was indeed used to great extent by the professionals it was designed for and if it was useful for the task of IT monitoring management and visualization. It is requested the collaboration of all professionals approached in order to allow future improvement and evolution of the developed platform.

* Required

1. From which devices and/or services did you use the platform the most? *
   - ☐ Web Browser (via the Web Application)
   - ☐ Mobile Device (via the Web Application)
   - ☐ Terminal or Programmatic Clients (via the REST APIs)
   - ☐ Postman, Insomnia, or other API Client (via the REST APIs)
   - ☐ Other

2. How often did you use the monitoring solution?
   - ☐ Several times a day
   - ☐ Once or twice a day
   - ☐ Rarely
   - ☐ Other

3. Were you ever able to access information regarding monitoring facilities or organizations you didn't have access to?
   - ☐ Yes
   - ☐ No

4. Did you ever experience some type of delay or lag while interacting with the system?
   - ☐ Yes
   - ☐ No

   a. If yes, please elaborate on which occasion/situation(s) it happened.

5. Did you ever experience any type of inconsistency while interacting with the system (e.g. incorrect monitoring data, miss-labelled health status, failures in alerts triggering, …)?

   ☐ Yes
   ☐ No

   a. If yes, please elaborate on which occasion/situation(s) it happened.

6. **Please, classify the following statements.** Number each statement according to your perspective on it, from 0 to 4. * [0 - Strongly Disagree 1 - Disagree 2 - Neutral 3 - Agree 4 - Strongly Agree]

   a. The monitoring agents' variety is appropriate/sufficient for your organization's monitoring needs.

0	1	2	3	4

Strongly Disagree Strongly Agree

   b. Every time a monitored service failed you were warned in accordance with the configurations you specified.

0	1	2	3	4

Strongly Disagree Strongly Agree

   c. The alerting delivery mechanisms are appropriate/sufficient for your organization's setup and needs.

0	1	2	3	4

Strongly Disagree Strongly Agree

   d. The notification alerts provided enough context and information to successfully inform you of the problem at hand.

0	1	2	3	4

Strongly Disagree Strongly Agree

e. The deployment of the local server (i.e. the data collection clients) was simple and straightforward and occurred with zero or minor installation problems.

0	1	2	3	4

Strongly Disagree Strongly Agree

f. The monitoring platform, as a whole, simplified the monitoring management and consumption in your organization.

0	1	2	3	4

Strongly Disagree

g. The monitoring platform, as a whole, fulfilled your organization's needs better than the previously used monitoring solution.

0	1	2	3	4

Strongly Disagree Strongly Agree

7. In your opinion, what are the functionalities that should be added to the solution? *

8. In your opinion, is there any current functionality that could be improved and/or expanded? If so, please elaborate on the improvement. *

Thank you.

Appendix C

Reference Examples

C.1 GitHub Action to Create and Publish Docker Image

```yaml
name: Create and publish a Docker image

on:
  push:
    tags:
      - '*'

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Log in to the Container registry
        uses: docker/login-action@v1
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v2
        with:
          context: .
          push: true
```

Example 11: GitHub action workflow to create and publish Docker image

C.2 Guardian Server's Partial Docker Compose

```yaml
version: "3.8"
services:
  kong:
    image: ghcr.io/aida-monitor/kong:latest
    restart: always
    ports:
      - "80:8000"
      - "443:8443"
    environment:
      ...
  influxdb:
    image: influxdb:2.1
    restart: always
    ports:
      - "8086"
    environment:
      ...
    volumes:
      ...
  mongodb:
    image: mongo:5.0
    restart: always
    ports:
      - "27017"
    environment:
      ...
    volumes:
      ...
  auth:
    image: ghcr.io/aida-monitor/auth:latest
    restart: always
    environment:
      ...
  core:
    image: ghcr.io/aida-monitor/core:latest
    restart: always
    environment:
      ...
  alert-manager:
    image: ghcr.io/aida-monitor/alert-manager:latest
    restart: always
    environment:
      ...
  data-exporter:
    image: ghcr.io/aida-monitor/data-exporter:latest
    restart: always
    environment:
      ...
```

Example 12: Docker Compose - Guardian server

C.3 Local Server's Docker Compose

```
version: "2"

services:
  local-pull-executor:
    image: ghcr.io/aida-monitor/local-pull-executor:latest
    restart: always
    container_name: local-pull-executor
    network_mode: bridge
    environment:
      ORGANIZATION_NAME: ${ORGANIZATION_NAME}
      CORE_URL: ${CORE_URL}
      INFLUXDB_URL: ${INFLUXDB_URL}
      INFLUXDB_ORG: ${INFLUXDB_ORG}
      INFLUXDB_BUCKET: ${INFLUXDB_BUCKET}
      INFLUXDB_TOKEN: ${INFLUXDB_TOKEN}
      AUTH_TOKEN: ${AUTH_TOKEN}
      NODE_ENV: ${NODE_ENV}
      HTTP_PROXY: ${HTTP_PROXY}
      HTTPS_PROXY: ${HTTPS_PROXY}
      http_proxy: ${HTTP_PROXY}
      https_proxy: ${HTTPS_PROXY}

  local-push-receiver:
    image: ghcr.io/aida-monitor/local-push-receiver:latest
    restart: always
    container_name: local-push-receiver
    network_mode: bridge
    ports:
      - "6000:3000"
    environment:
      CORE_URL: ${CORE_URL}
      INFLUXDB_URL: ${INFLUXDB_URL}
      INFLUXDB_ORG: ${INFLUXDB_ORG}
      INFLUXDB_BUCKET: ${INFLUXDB_BUCKET}
      INFLUXDB_TOKEN: ${INFLUXDB_TOKEN}
      AUTH_TOKEN: ${AUTH_TOKEN}
      NODE_ENV: ${NODE_ENV}
      HTTP_PROXY: ${HTTP_PROXY}
      HTTPS_PROXY: ${HTTPS_PROXY}
      http_proxy: ${HTTP_PROXY}
      https_proxy: ${HTTPS_PROXY}
```

Example 13: Docker Compose - Loval server

Appendix D

Publications

D.1 Pervasive Monitoring System for Services and Servers in Healthcare Environment

Authors: Carolina Marques, Vasco Ramos, Hugo Peixoto, and José Machado

Title: Pervasive Monitoring System for Services and Servers in Healthcare Environment

Conference: The 5th International Conference on Emerging Data and Industry 4.0

Year of Publication: 2022

Abstract: Information systems are continuously evolving in nature and complexity. In the healthcare environment, information and information exchange are critical for providing health care at all levels. Hence, the healthcare environment is particularly relevant when discussing IT infrastructure monitoring and disaster prevention since availability and communication are vital for the proper functioning of healthcare units, whether acting in isolation or on a network.

This work focuses on understanding what comprises a good monitoring solution, analyzing the monitoring solutions currently available in the optics of a multi-location healthcare environment, and, finally, proposing a pervasive and comprehensive conceptual architecture for a monitoring system that is capable of handling such environments.

Keywords: Health Information Systems, IT Monitoring, Systems Microservices

State of Publication: Published

D.2 Information Technology Monitoring in Healthcare: A Case Study

Authors: Vasco Ramos, Carolina Marques, Hugo Peixoto, and José Machado
Title: Information Technology Monitoring in Healthcare: A Case Study
Conference: The 6th International Conference on Information Technology & Systems
Year of Publication: 2023 (estimated)

Abstract: The healthcare environment is particularly relevant when discussing information technology infrastructure monitoring since availability and communication are vital for the proper functioning of healthcare units. It is important to be able to easily monitor and observe each unit from a single point of access so that actions can be swiftly taken when there is a problem.

This paper proposes a multi-site and multi-organization web and microservices-based information technology infrastructure monitoring solution. In addition to exploring the developed system and its architecture, it presents a case study resulting from the system's implementation in an organization and holds a discussion about the obtained results to determine whether a multi-platform monitoring system improves information technology availability in the healthcare industry.

Keywords: IT Infrastructure Monitoring, Microservices, Health Information Systems, Healthcare

State of Publication: Under Review