UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Architecture and Deployment of Web Applications with a focus on ensuring Data Privacy

Pedro Manuel Marques Pereira

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Mário João Barata Calha

2025

Agradecimentos

Ter trabalhado nesta tese de mestrado influenciou significativamente o meu desenvolvimento técnico pois permitiu-me adquirir competências em áreas com quais ainda não tinha tido contato. A vontade de querer aprender algo novo foi um forte agente motivador inicial porém, com o passar do tempo, desafios foram surgindo.

Por este motivo quero agradecer ao Professor Doutor Mário Calha que muito contribuiu para ultrapassar estes desafios. Durante a sua orientação de tudo fez para encontrar um caminho quando a solução parecia distante e para motivar à conclusão deste trabalho com sucesso. Da mesma forma, quero agradecer aos meus dois colegas, André Krippahl e Tiago Almeida, que trabalharam em temas paralelos e que sem os quais o trabalho teria sido sem dúvida mais difícil.

Deixo também uma palavra de apreço à equipa BITalino, que proporcionou o tema deste trabalho e à Faculdade de Ciências da Universidade de Lisboa, instituição que me acolheu. A todos, votos de sucesso.

Por fim, quero agradecer à minha família e amigos que sempre me apoiaram durante o meu percurso académico e profissional.

Resumo

As tecnologias de virtualização e os sistemas de orquestração têm-se tornado cada vez mais essenciais na engenharia de software com o decorrer dos anos. Ao se integrarem profundamente no panorama das tecnologias de informação e ao criarem uma estrutura resiliente para a gestão de aplicações em ambientes computacionais heterogéneos, a virtualização encapsula componentes e as suas dependências, garantindo uma execução consistente independentemente da infraestrutura subjacente, enquanto a orquestração introduz a automatização necessária para a alocação de recursos e escalabilidade, assegurando que as aplicações se mantenham eficientes e adaptáveis às exigências em constante mudança.

As unidades de processamento virtualizadas são utilizadas para satisfazer os requisitos funcionais deste desenho arquitetural. Apesar de estas unidades poderem ser implementadas de várias formas, partilham normalmente necessidades fundamentais semelhantes: capacidade para receber informação de entrada, processá-la, persistir os dados resultantes e tornar estes resultados acessíveis para consultas ou processamento subsequente. Considerando que um volume substancial de informação pode fluir entre diferentes programas, é imperativo permitir a colaboração entre serviços distintos e minimizar cálculos redundantes. Alcançar este nível de interoperabilidade requer não apenas interfaces de programação de aplicações (APIs) bem definidas, mas também a implementação de estratégias para manipular os dados consumidos e produzidos de forma segura.

Estas ferramentas de virtualização surgem também como resposta à rápida evolução das exigências do mercado. Muitas empresas adotam metodologias de trabalho como o Agile, em que múltiplas entregas de produto são desenvolvidas simultaneamente. Isto significa que vários programadores, ou até diferentes equipas de desenvolvimento, podem trabalhar no mesmo projeto, mas em serviços web paralelos que necessitam de ser implementados de forma simultânea ou independente. A nível arquitetural, esta abordagem modular apresenta desafios, uma vez que diferentes componentes da aplicação devem poder ser modificados ou substituídos sem comprometer a funcionalidade global do sistema. Para garantir entregas rápidas e contínuas, tornou-se comum a existência de equipas DevOps que facilitam o deployment e asseguram a entrega contínua de software. DevOps representa uma abordagem colaborativa entre desenvolvimento e operações, otimizando o processo de entrega de software. Ao manter estas equipas sincronizadas, é possível

criar ciclos de desenvolvimento mais ágeis, permitindo a integração rápida e segura de novas funcionalidades. É igualmente relevante que a equipa de desenvolvimento cumpra quaisquer restrições e regulamentos da infraestrutura, que podem impor limitações que necessitam de ser respeitadas.

Particularmente em grandes empresas digitais, é comum existirem diferentes aplicações que resolvem problemas específicos mas partilhem os mesmos dados e são geridas pela mesma equipa. Estas aplicações podem ainda possuir arquiteturas de software distintas, e com tantos serviços a trocar informação, torna-se frequente a recolha e armazenamento de dados de utilizadores. Estes dados podem ser usados para análise de tendências ou fins estatísticos. Podem também ser enviados para serviços externos, dificultando a rastreabilidade do seu percurso. Qualquer informação que permita identificar o seu proprietário é considerada dado privado, desde um nome ou morada a um simples endereço IP.

Em abril de 2016, foi criado na União Europeia o Regulamento Geral de Proteção de Dados (RGPD), implementado em Maio de 2018. Este regulamento pretende garantir que os utilizadores de qualquer website ou aplicação online que recolha dados privados mantenham controlo sobre a sua informação pessoal, servindo como norma aplicável em toda a Europa. Software de terceiros fora da União Europeia que consuma serviços de uma aplicação europeia também está ao abrigo dos artigos presentes no RGPD, estando sujeitos a multas de até 20 milhões de euros ou 4% do volume de negócios anual, consoante o valor que for mais elevado. O regulamento inclui diversos artigos que impactam directamente o desenvolvimento de software, como o direito ao esquecimento, permitindo ao utilizador solicitar a eliminação total dos seus dados pessoais. Por estas razões, é importante que as aplicações adotem uma abordagem que priorize a proteção de dados desde a conceção e por padrão, como mencionado no artigo 25.º do RGPD, exigindo que o responsável pelo tratamento incorpore salvaguardas eficazes para processar informação privada. O responsável pelo tratamento deve também implementar medidas para garantir que cada dado privado serve um propósito e só é armazenado quando estritamente necessário. Garantir conformidade com o RGPD pode tornar-se complexo, especialmente em empresas com múltiplas equipas, tornando processos automatizados essenciais. Empresas frequentemente possuem várias aplicações a aceder aos mesmos dados, como um banco que utilize microserviços para crédito habitação e um sistema monolítico para empréstimos pessoais. Tecnologias como Docker e Kubernetes são cruciais para gerir estas aplicações de forma eficiente, sendo a colaboração entre DevOps e equipas de desenvolvimento vital para cumprir os requisitos do RGPD em todos os sistemas.

Neste contexto, o presente trabalho aborda os requisitos provenientes da equipa BITalino. Esta equipa fabrica dispositivos baseados em placas Arduino, hardware capaz de integrar vários tipos de sensores, utilizados para recolher sinais biométricos dos utilizadores. Após uma sessão de medição destes sinais, os dados adquiridos são armazenados num ficheiro CSV dentro do dispositivo BITalino, podendo posteriormente ser transfe-

ridos para um computador e processados localmente através de scripts Python independentes, também desenvolvidos pela equipa. É importante notar que, segundo o RGPD, dados biométricos são considerados informação pessoal sensível, tornando a abordagem anterior incompatível com os requisitos de privacidade. Além disso, não existia uma arquitetura de software escalável. Como prova de conceito, os sinais biométricos e scripts disponíveis são aplicados a um caso específico: eventos de natação.

Propõe-se, portanto, uma framework capaz de melhorar os procedimentos associados à equipa BITalino e aos seus dispositivos. Embora existam várias perspetivas a explorar sobre este tema, a infraestrutura e os dados de saída gerados pelos scripts são as principais preocupações.

Iremos explorar ferramentas capazes de providenciar a infraestrutura de software, permanecendo independentes de qualquer fornecedor de serviços em cloud em particular. Mecanismos a nível aplicacional também serão discutidos para garantir os casos de uso e os requisitos funcionais desta equipa. Temos como ponto de partida os scripts disponíveis já em containers e num repositório de imagens como DockerHub. Estes scrits serão consideradas funções serverless.

Como prova de conceito, foi implementado e aqui descrito em detalhe um demonstrador recorrendo a Terraform, Kubernetes e Helm charts para providenciar a infraestrutura necessária. Aqui terão que ser garantidas as condições para que os scripts da equipa possam ser executados de forma tão isolada quanto possível. Foram considerados cenários para além da natação e assumimos que nem sempre teremos acesso ao código fonte dos scripts, levando a que estes possam no futuro ser utilizados para explorar falhas de segurança do sistema. Já a nível aplicacional, API's auxiliares foram desenvolvidas recorrendo a Spring, uma framework de Java, e Hibernate. Foram ainda utilizados elementos externos à nossa infraestrutura como Auth0 e MinIO para facilitar objetivos recorrentes em aplicações deste carácter tais como gestão de utilizadores e armazenamento de objectos. Todos os elementos adicionais criados são ao longo do trabalho devidamente justificados.

Por fim, o demonstrador é avaliado tendo em conta os requisitos funcionais e não funcionais da equipa. São também apresentados alguns pontos que não foram aqui abordados em detalhe mas que poderão no futuro enriquecer a infraestrutura.

Este estudo é um de três desenvolvidos tendo como contexto as necessidades levantadas pela equipa BITalino, cada um contribuindo para os seus respectivos requisitos e problemas específicos. Estes outros dois trabalhos serão brevemente apresentados uma vez que são relevantes para o desenho da framework aqui apresentada. Serão mencionados apenas quando se justificar.

Palavras-chave: Virtualização, infraestrutura, privacidade, BITalino, segurança

Abstract

This study focuses on enhancing the BITalino ecosystem, a platform for acquiring biometric signals through sensor based devices. Until now, the processing of these biosignals relied on standalone scripts without a supporting infrastructure.

To address these issues, we developed a Kubernetes-based architecture using microservices to enable containerized script execution. For storing the resulting output files, we implemented an object storage solution suitable for handling unstructured data.

The handling of biometric data also introduces regulatory challenges in this virtualization. Since such data is considered sensitive under the General Data Protection Regulation (GDPR), the system incorporates mechanisms for anonymization or full deletion of data, ensuring the privacy of the users. Measures were implemented to mitigate the OWASP Top 10 vulnerabilities, enhancing the overall security and reliability of the system.

As a case study, the solution was validated in the context of swimming, where scripts for processing session files were executed through APIs, with some adaptations, Kubernetes Jobs and CronJobs.

Keywords: Virtualization, infrastructure, privacy, BITalino, security

Contents

List of Figures

List of Tables

Acronyms

APIs Application Programming Interfaces

GDPR General Data Protection Regulation

CSV Comma Separated Values

ECG Electrocardiographic

CLI Command Line Interface

OS Operating System

CI/CD Continuous Integration and Continuous Delivery/Deployment

OWASP Open Web Application Security Project

IaC Infrastructure as Code

JWT JSON Web Tokens

GCP Google Cloud Provider

CA Certificate Authority

Chapter 1

Introduction

1.1 Motivation

Containerization technologies and orchestration systems have become essential to modern software engineering. By forming a deeply embedded presence in the IT landscape and creating a resilient framework for managing applications across heterogeneous computing environments, containerization encapsulates components and their dependencies ensuring consistent execution regardless of the host infrastructure, while orchestration introduces the necessary automation for resource allocation and scaling, ensuring that applications remain efficient and adaptable to ever-changing demands.

The processing units inside the containers are used to achieve the functional requirements of this architectural design. Dispite these units can be implemented in various manners, they typically share similar fundamental needs: the capacity to receive input information, process it, persist the resulting data, and make these results accessible for queries or subsequent processing. Given that a substantial volume of information can flow through this environment from other containers, it is imperative to enable collaboration between different services and minimise redundant computations. Achieving this level of interoperability necessitates not only well-defined application programming interfaces (APIs), but also the implementation of strategies to securely handle the data consumed and produced.

The use of these containers and orchestrators is also a response to the rapidly evolving market demands. Many companies adopt work methodologies like Agile [5], in which multiple product deliveries are developed simultaneously. This means that multiple developers, or even different teams of developers, may be working on the same project, but on parallel web services that may need to be implemented simultaneously or independently. From a high level perspective, this modular approach presents challenges regarding the architectural style an application should adopt, as different application components must be modified or replaced without disrupting the overall system functionality. To ensure the rapid and continuous delivery of numerous applications, it has become common to have

DevOps [21] teams that facilitate deployment and ensure continuous software delivery. DevOps is a collaborative approach between development and operations teams to optimize the software delivery process. By keeping these teams synchronized, it is possible to create a more agile development cycle, enabling continuous delivery and the integration of new features more quickly and securely. It's worth noting that the development team must also comply with any infrastructure restrictions and regulations, as these may impose limitations that must be adhered to.

Especially in large digital market companies, it's possible to find different applications that address specific problems but share the same data and are managed by the same team. These applications may also have distinct software architectures, and with so many services exchanging information, it's common to collect and store user data. This information can be used in a variety of ways, such as trend analysis, statistical purposes, and all types of data mining. The data may also be sent through a third-party service, making it difficult to trace its path. Any information that can identify its owner is considered private data, from a name or address to a simple IP address.

In April 2016, a General Data Protection Regulation (GDPR) [1] was created in the European Union and implemented in May 2018. This regulation aims to ensure that users of any website or online application that collects private data have control over their personal information and serves as a standard regulation for all of Europe. If third-party software outside the European Union consumes services from a web application in Europe, it is also subject to the GDPR, and those that fail to comply with these rules can face fines of up to €20 million or up to 4% of their annual revenue, whichever is greater. Within this regulation, there are many articles that directly impact software development, such as the "Right to be Forgotten", whereby a user can request the deletion of all their personal data at any time. For these reasons, it is important that our applications adopt an approach that prioritizes data protection by design and by default, as also mentioned in the GDPR regulation (Article 25), stipulating that the data controller must incorporate effective safeguards to process all private information. Furthermore, the data controller must implement measures to ensure that each piece of private data serves a purpose and is stored only when necessary. Ensuring GDPR compliance for applications is a complex task, especially in large companies with multiple teams. Analyzing every use case for code or software manually can be impractical, so automated processes are crucial. Companies often have multiple applications accessing the same data. For example, a bank might use microservices for mortgage lending and a traditional monolithic system for personal loans. Technologies like Docker and Kubernetes are essential for efficiently managing different applications. Collaboration between DevOps and development teams is vital to meeting GDPR requirements across all systems.

Given the current context, this work addresses the requirements arising from the BITalino team. This team manufactures Arduino based devices capable of integrating

various types of sensors, which are then used to collect biometric signals from their users. After a signal measurement session, the acquired data is stored in a comma separated values (CSV) file within the BITalino device. This data can then be transferred to a computer and processed for various purposes using independent Python scripts, also developed by this team. It is important to consider that according to the GDPR, biometric data is categorized as private user information, which means that this team's approach directly conflicts with the privacy concerns required by this regulation. It also lacks a scalable software architecture for the reasons already mentioned. As a proof of concept, the biosignals used, as well as the available scripts, will come from a specific case: swimming events.

Therefore, this work proposes a framework capable of improving the procedures involving the BITalino team and its devices. It is worth noting that there are several perspectives to be explored regarding the presented topic, but the infrastructure as a whole and the output data generated by the scripts are the main concerns here. This study is one of three developed on this topic, which will also impact the requirements presented later. Each study will offer its own contribution and address specific problems.

1.2 Goals

This work aims to create a software architecture capable of assisting in the processing of data produced by BITalino that is GDPR-compliant, ensuring the privacy of system users and data owners. Non functional requirements will be a very present concern in this work so the architecture presented is designed to be scalable, secure, efficient, while maintaining compliance with relevant regulations and following best practices.

To achieve this objective, the following tasks should be considered:

- Granting users permission to authenticate into the system and submit data.
- Allowing users to execute scripts in a secure environment.
- Properly hosting the data resulting from script processing and providing access to it.
- Designing this infrastructure to be compatible with a cloud environment but not dependent on any cloud provider in specific.
- Implementing mechanisms for deploying and modifying the entire infrastructure as needed.

1.3 Contributions

The accomplishment of this work was made possible through a partnership between Faculdade de Ciencias da Universidade de Lisboa (FCUL) and the company responsible for

the development of BITalino devices. The contributions of this work are:

- The design of a software infrastructure that complies with GDPR.
- A study on script execution in a containerized environment and data handling.
- The potential deployment of this application in a cloud environment.

1.4 Document Structure

As is usual in this type of work, this document is structured into distinct chapters, each addressing a critical aspect of the proposed architectural structure. The Chapter 2: "Literature Review" offers an in-depth exploration of relevant concepts and prior research. The BITalino device will be comprehensively introduced, and the specific swimming case under study will be duly detailed. This chapter examines topics such as containerization and orchestration technologies, data processing techniques, GDPR compliance, standards for cloud computing, authentication and authorization mechanisms, as well as security best practices.

Chapter 3, "Specification and Design," outlines the approach taken to design the proposed architecture. It covers the actors to be considered by the system, use cases and requirements gathering, as well as architectural design decisions and data flow within the system.

Chapter 4, "Deployment and Implementation", will address in greater technical detail the configurations chosen to provide the software infrastructure. It will also describe the supporting APIs implemented in the application layer and how they enable communication between system components.

Finally, Chapters 5 and 6 will present the results and verification of the identified use cases and requirements, along with the corresponding conclusions.

Chapter 2

Literature and related Work

This chapter will address the literature and related work that will serve as a starting point to solve the challenges proposed in the previous chapter. It will explain in detail what the BITalino product is, as well as how this product is currently being used, highlighting the major problem that currently exists with the lack of user privacy and ways to correct it. Since the objective is to create an infrastructure capable of making use of what BITalino offers and that must be designed considering future use cases of this product, standards will be discussed that help with scalable development and security standards that any software system should follow.

2.1 BITalino

2.1.1 What is it?

Measuring biosignals allows us to capture events such as heartbeats or muscle contractions, a process that can be used for several purposes. As a team of researchers sought to measure these biosignals, they encountered a problem: the existing equipment to measure biosignals was too expensive. To try to overcome this obstacle, BITalino was created. As mentioned in its presentation paper, published in 2013 [4], BITalino is a low cost biosignal acquisition system composed of both a hardware and an open-source software component. Its hardware is based on an Arduino board that consists of an electronic component capable of being connected to different types of sensors and has already been successfully used to acquire, for example, Electrocardiographic (ECG) changes. Regarding its software, it consists of firmware to control the Arduino board and an API that controls the data acquisition process. Over the years, new works were developed based on BITalino and new prototypes with different purposes were launched, showing application capacity in different scenarios.

2.1.2 Its applications

As mentioned in the introduction, our case study is set within the context of swimming. However, the software architecture proposed in this work also aims other BITalino scenarios, so it is essential to understand the potential application possibilities facilitated by this device. The integration of the above mentioned sensors allows the detection of biosignals from various sources, providing data to our system. In addition to these sensors, there are also actuators that receive signals and initiate physical actions or movements to control the system. In the article entitled "Bitalino use and applications for health, education, home automation and industry" published in 2017 [16], several sensors and actuators are discussed, along with their potential use cases at the time.

Here is a brief description of them:

- Electrocardiogram (ECG) sensor: allows detecting the electrical differentials of the heart cells. These sensors can be used in heart rate monitoring, physical activity measurements, bionic prosthesis, robotics, biomedical, among others.

- Light sensors (LUX): capable of detecting variations in luminosity. It can be used not only for automatic brightness adjustment in electrical equipment such as LCD or smartphones but also to detect changes in body temperature.

- Electrodermal activity (EDA) sensor: measures electrical conductance of the skin to assess physiological and emotional arousal. It detects changes in sweat gland activity to monitor stress, anxiety, and sympathetic nervous system responses.

- Adaptive cruise control (ACC) sensor: An ACC sensor, equipped with a chip capable of detecting acceleration on the X, Y, and Z axes, enables users to monitor and analyse movement in all three dimensions individually or collectively. Its versatile applications range from activity monitoring, motion and tilt detection, image stabilization, sports equipment, hard disk protection, and more.

- LED actuator: a basic LED, controlled by BITalino, used to provide visual feedback.

- Button actuator: a button for controlling functions on BITalino.

- Buzzer actuator: produces a sound signal when powered. Its main applications involve audio feedback particularly for alarms and other types of notifications.

- Touch sensor actuator: a touch sensor, serving as an alternative to a mechanical button. It is also waterproof.

2.1.3 Swimming context

As just demonstrated, BITalino boasts a broad range of applications so this work will leverage its high modularity and scalability. The solution presented as to be compatible with these scenarios but the main focus is the context of swimming. In 2016, the study "Electrocardiography, electromyography, and accelerometry signals collected with BITalino while swimming: Device assembly and preliminary results" [17] was published, affirming not only the feasibility of collecting biosignals in aquatic environments but also the potential for real-time data acquisition. Taking these facts into account, this work will have as background swimming athletes, using data collected during training sessions with a BITalino device. One of the scenarios considered will involve implementing this infrastructure within the Portuguese Swimming Federation. According to a work also carried out in the context of BITalino, [3], in 2018 there were 15,000 registered athletes in the federation. However, the potential number of athletes who could use this system is even larger, requiring thoughtful system design to handle the increased scale effectively. Additionally, in this federation scenario is essential to take into account the possibility of an athlete having a coach or being part of a larger team. Therefore, data sharing mechanisms must also be taken into consideration.

2.1.4 Data files and Metadata

Once a measurement session using the BITalino device ends, a CSV file is generated which can then be downloaded to a computer for analysis. The CSV content will depend on the type of session and the sensors used. In the swim context, it will contain information such as variation in body positioning over time and angles of rotation, which typically have a significant impact on sports performance. Although this specific CSV example does not contain data that could explicitly identify the athlete who performed the session, we must expect that these sessions may also contain biometric data. Biometric data refers to unique physical or behavioural characteristics and should be treated as private data. These CSV files will be the main input data for the system proposed and will be often mentioned as session data. As the purpose is to analyse session data, complementary data to those in the CSV files will frequently be considered. Information such as weight, height, or age group in a sports context may be relevant and should also be associated with the athlete. The system must also provide means to accommodate this data and treat it as private. This data will commonly be referred to as session metadata. Once the session data is collected, the BITalino team uses programs to generate new output data. This data can be used for various types of analysis, depending on the program that processes it. This means the system should not rely on a specific format for these output files, as they may be generated as PDFs, images, or simply JSON of any size. Since the processing to generate these outputs is not predefined and, in the future, programs with

any kind of purpose may be created, these session outputs also have the potential to be private data.

2.1.5 The processing scripts

As mentioned, after collecting data during the presented sessions, it can be analysed for any desired purpose. The BITalino team develops Python scripts to facilitate data analysis using various algorithms. These scripts are stateless programs that receive the session data and, upon execution, generate the new output data. For this project the BITalino team could only provide one exemplary script but it adds insight into what other aspects should be considered. Although the received script was developed in Python, the system should be capable of integrating and executing scripts written in any programming language. The BITalino team has the objective to produce more of these scripts without defining specific requirements. Since the infrastructure here proposed must be able to execute these scripts in order to persist the output data, some script requirements have to be decided in Chapter 3. Given that Python scripts can serve several purposes, they may require different libraries for their functionality, which can raise security problems. There are tools that can verify these dependencies, although this will not be the focus of this work.

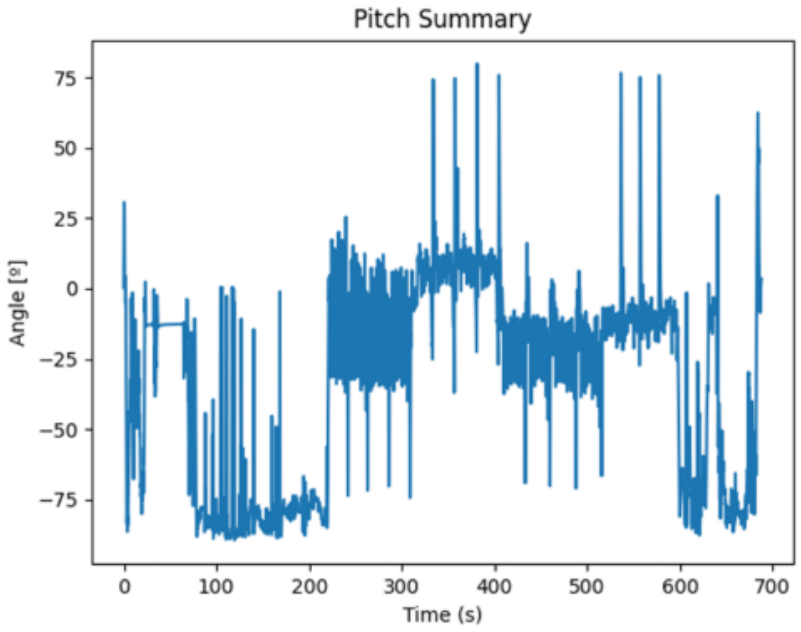

Figure 2.1: Script output example

As an example, in the Figure 2.1 we can observe a picture that was generated by a Python script with the aid of libraries designed for this purpose. Through this image, the BITalino team can even deduce the swimming style the athlete used during the data collection session. This demonstrates the great versatility that scripts enable.

Since this work aims to create a solution that is cloud compatible, it may be required to apply some changes to the current scripts. Cloud communication is often achieved using means such as APIs or Event Streaming and currently the script used as an example is not prepared for this new context but all changes will be properly described and justified.

2.1.6 The current system

Team BITalino manages the CSV data from sessions containing sensitive personal information on their individual machines. Since the team conducts data analysis by executing locally the already mentioned scripts there is also a possibility of data sharing within the team. These scripts are developed in Python, as it stands as the dominant language within the team, however, the objective is to diversify and create scripts in other programming languages as well.

Currently the execution of these scripts is initiated through the command line interface (CLI), and although the scripts are executed manually, the objective is for scripts to be automatically executed at certain times of the day and to be capable of analysing high volumes of data. To function properly, the scripts themselves need to know the precise location of the CSV session data files, enabling them to generate the necessary output. Following the execution, the generated output files are also stored locally on the machine of the individual who initiated the script. Despite the fact this system may be seen as convenient for the team that is developing scripts, there are several serious privacy and security concerns and it is possible to verify that the processes that the team follows are not scalable, a reason that was strongly at the origin of this work.

2.2 The privacy concern

With the presentation of the current system made, we can see that several problems in terms of user privacy may arise. Since the information from the swimming sessions must be in the same place where the scripts are executed, there is no control over who accesses it, nor is it possible to say what type of processing these session files were subjected to, because anyone with the session files and the Python script locally can generate new results.

Now, an analysis will be conducted regarding the best practices on privacy to which any system should be subject.

2.2.1 GDPR

The General Data Protection Regulation (GDPR) is a European law implemented in May 2018 that aims to protect the privacy of all personal data belonging to consumers of IT

applications. According to this regulation, the owners of such applications are fully responsible for preserving the privacy and security of the users. This regulation was created within the European Union but websites or applications outside the EU must also comply with the GDPR if they process the personal data of European citizens, ensuring that a consistent level of data protection is maintained for EU citizens, regardless of where their data are being processed.

For an application to comply with the GDPR, users must explicitly grant authorization regarding the specific purposes for which their data will be used. This means that users should be informed about how the data controller intends to handle their data, ensuring transparency and informed consent. Non-compliance with the GDPR can lead to severe consequences. The regulation empowers supervisory authorities to impose significant fines and penalties, which can be to a percentage of global annual turnover or a fixed monetary value, depending on the nature and severity of the violation. Given that the case study of this work involves the processing of biometric data, it is crucial to underline that this specific type of data is in fact protected by the GDPR. As defined in Article 4(14) of the GDPR, biometric data refers to "personal data resulting from specific technical processing relating to the physical, physiological or behavioural characteristics of a natural person, which allow or confirm the unique identification of that natural person , such as facial images or fingerprint data". Given their sensitive nature, biometric data are also classified in the special categories of personal data in article 9.1, which means they are subject to more stringent processing conditions. This underlines the importance of implementing strict protection measures when processing biometric data to ensure full GDPR compliance.

Main Articles

The GDPR is an extensive regulation with several articles that directly impact software architecture design. In addition to those related to biometric data mentioned earlier, there are key articles that address critical challenges that data controllers, the name given in the GDPR to the data holder, must manage:

- Article 5.1 (f) Data should not be processed without authorization;
- Article 5.2 The data controller shall be responsible for it and shall be able to demonstrate what actions did;
- Article 6 Lawfulness of processing;
- Article 9 Processing of special categories of personal data(e.g. biometric data);
- Article 17 Right to erasure ('right to be forgotten');
- Article 22 Automated individual decision-making, including profiling;

- Article 25 Data protection by design and by default;
- Article 32 Security of processing;

According to Article 5.1(f) of the GDPR, private data can only be processed with explicit authorization. Therefore, a system that shares data must have clear definitions of the specific services for which consent has been granted to use the data. This ensures compliance with the GDPR's requirement of informed and explicit consent for data processing activities.

The article 5.2 implies that every time that a service consumes data, the usage of it shall be registered. This means that each instance of data consumption should be appropriately logged and documented.

Article 6 reinforces that a consent shall be given by the user to the controller of the data. It also mentions the same applies to possible third party services.

Article 9 of the GDPR focuses on the processing of special categories of personal data, including biometric data. Processing biometric data generally requires explicit consent from the data subject or must be justified by specific legal grounds, emphasising the need for heightened protection and careful handling of such sensitive information.

The Article 17 may be one of the most challenging articles to handle on big architectures because multiple services can coexist in the same systems and some of them may collect data in different ways. Ensuring compliance with this article may require careful coordination and implementation of processes to accurately identify and delete personal data across all relevant services and data sources, taking into account the different data formats and storage locations involved.

On article 22 the user shall have the right not to be subject to a decision based solely on automated processing. This means that any significant actions affecting the user should involve human review.

Article 25 is one of the most discussed articles, and it defines that the data controller shall implement appropriate technical and organisational measures to ensure privacy protection. These measures should be based on state-of-the-art safeguards and mechanisms. All measures implemented in the infrastructure will be discussed in the following chapters.

Article 32 reiterates that the state of the art mechanisms of security shall be implemented. Some of these techniques are personal data encryption, proper backups to recover data in case of physical or technical security breach, testing and evaluating the effectiveness of the applied measurements. The complete regulation can be found in [1].

Logging private data

Logging is a common practice in software systems. It can provide evidence of what actions occur, being relevant under GDPR Article 5.2, as well serving purposes such as

troubleshooting, performance monitoring, among others.

Despite GDPR not explicitly mentioning logging it gives relevance for data minimisation, meaning logs should only capture the minimum amount of data necessary for the specific purpose they serve. Logging excess data, particularly sensitive personal information (e.g., names, email addresses, or IP addresses), should be avoided unless it's necessary for legitimate reasons.

Since the current system creates the output files locally, the scripts may require some significant adaptations for being executed in a solution compatible with cloud. One option could be to change the scripts to convert the output file to a Base64 string and log it but, as already mentioned, these output files may also contain private data so better approaches will be followed in Chapter 3.

2.3 Software Infrastructure

Now that the current system used by the BITalino staff is described in 2.1.6, by showing biometric information as personal information and that therefore should be kept as private, is justified the need to plan a software infrastructure capable of handling these subjects. This section will serve as a starting point for the solution design by specifying some of the currently used tools and standards, as well as discussing the situations in which they can be most advantageous.

2.3.1 A definition and its importance

The infrastructure covers all the essential components required to run an application, including hardware, network resources, storage and virtual environments, forming the basis of any system. All of these components that are part of the final product will have to interact with each other in a well-defined and optimised way. For most complex systems, the cost of maintaining infrastructure significantly exceeds development expenses, which justifies the need for careful planning of infrastructure from the start since rash decisions can lead to increased costs or negatively affect typical non-functional software requirements, such as reduced scalability, essential for most products in the current context, or cause other system failures over time. A properly planned infrastructure allows the system to evolve without needing frequent and expensive reviews.

The design of infrastructures and its components has been a subject of study over the years, becoming a well established area in computer science. The Software Engineering Body of Knowledge (SWEBOK) [10] published by the IEEE Computer Society is an internationally recognized guide that provides consensual guidelines of knowledge in software engineering with best practices for designing robust infrastructures. SWEBOK addresses software architecture, system integration, and non-functional requirements like performance, security, maintainability and provides a framework for the development of

efficient and sustainable software systems. SWEBOK has a chapter dedicated to Software Design where it is provided key issues in design, like control and handling events or distribution of components as addresses Software Structure and Architecture. In this guide is quoted that Software Architecture is "the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" also referring that with the high demand of applications were emerging a number of architectural designs that can be applied in building systems such as that of this study.

2.3.2 A world in the cloud

As mentioned in the introduction of this work, the demand for processing capacity has increased significantly. On-premise solutions, where the software owner manages all infrastructure, tend to harm the fast availability and scalability of the system and since this work aims to offer a solution that is scalable to contexts beyond swimming, it becomes challenging to make predictions for the needs that may arise in the future. On the other hand, cloud computing allows the delivery of computing services over the internet, enabling on-demand access to resources like storage, computing power, databases or networking and where the infrastructure providers can allocate resources automatically without the need of human intervention.

The global public cloud services market has grown year after year, as shown in Figure 2.2, which demonstrates that a system designed for the long term must, at the very least, be compatible with this cloud-based world. A Public cloud is one of the deployment models but there are others. In a public cloud the infrastructure is hosted by a third-party provider like AWS, Azure or Google Cloud that own and manage the resources. In a private Cloud the infrastructure is operated exclusively for a single organisation, either on-premise or hosted by a third-party typically offering more control and security but being more costly. It is also worth highlighting the Hybrid cloud deployment model where with a combination of public and private clouds, allows data and applications to move between them, offering, as example, flexibility by implementing solutions where sensitive data are kept in private environments while public cloud resources scale as demanded.

In addition to deployment models, in a cloud environment, cloud service models should also be considered, with the most common being Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS), each offering different levels of resource management. Typically, the way these services are consumed varies between cloud providers, which conflicts with the goal of this solution being cloud-compatible but independent of any specific provider, so these topics will not be explored in depth. There is also Functions as a Service (FaaS), which could be relevant in the context of the previously mentioned Python scripts, however, this solution will not be considered for the same reasons as the other cloud services mentioned.

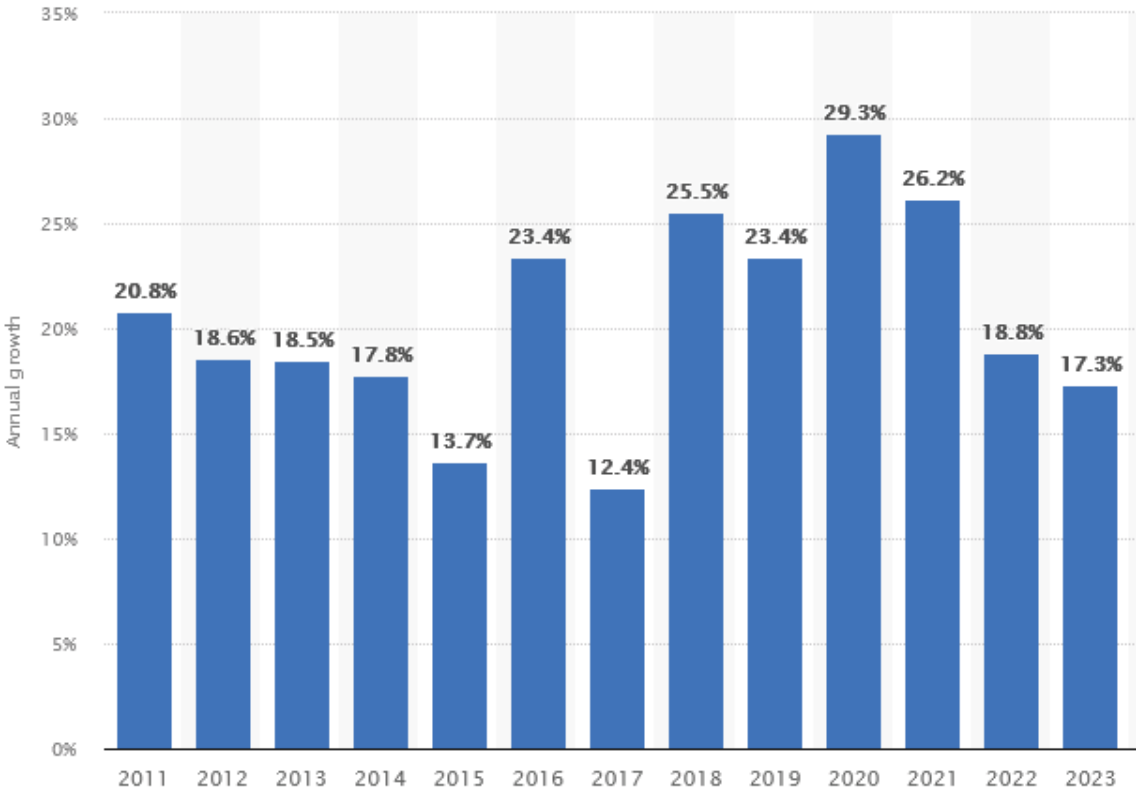

Figure 2.2: Public Cloud Market

2.3.3 Architectural Design Patterns

In software architecture, design patterns provide established solutions to common challenges in system design. Currently, the IT market seems dominated by Microservices specially in cloud environments, as will be mentioned below, but there are more options that can serve as blueprints for designing systems.

Below, some architectural patterns are briefly introduced, examining how they address specific needs and support efficient software development.

Broker pattern

The Broker design pattern is used to mediate communication between a client and a service provider by using an additional component for this purpose, the Broker. By decoupling the communication, the server must register with a broker, while the client accesses the broker directly. This pattern is commonly applied in distributed systems coordination and is also frequently used in other architectures such as Microservices or event-driven architectures.

Advantages:

- scalability because the same broker can intermediate multiple clients and servers;

- the client doesn't need to know the server's location;

Disadvantages

- broker has potential to be a single point of failure;
- consistency of the data may be affected in situations such as in caching;

Layered pattern

The Layered Pattern is a software architectural design that organises an application into distinct layers, where each layer has a specific role. Each layer interacts only with the one directly below it, promoting separation of concerns.

Advantages:

- modularity since one layer don't affect others;
- layers can be reused across different applications;

Disadvantages:

- can introduce performance bottlenecks;

Microservices pattern

Microservices is an architectural style in which an application is broken down into smaller, independent services that each perform a specific function. These services run in their own processes, communicate through lightweight protocols (such as HTTPs or messaging) and since each service is independent it can be developed or deployed without affecting the others.

In 2014 an article was published regarding this subject where it was affirmed that this style "is becoming the default style for building enterprise applications" [6], a fact that can be observed until current days. Microservices architecture is an evolution of Service-oriented architecture (SOA), where the services are installed by a single artefact.

Advantages:

- independence between services being less complex to develop, test and deploy each one;
- faster scaling;

Disadvantages:

- in large distributed systems it can be complex to manage all the services;
- one task can depend on multiple services;

Event driven pattern

In this pattern the system's components communicate through the production and consumption of events. Components emit events when something significant happens, and other components (listeners) react to those events.

Advantages:

- components can scale independently when receiving events;
- events can be processed asynchronously and in parallel by multiple components;

Disadvantages:

- managing and debugging event flows can be challenging;
- event timing and order may not be guaranteed;

2.4 Virtualization

In order for a program as a BITalino Python script to be available in an infrastructure and available in a distributed manner an virtual environment must be created. Virtualization is the process of creating a version of an environment on top of physical hardware such as servers or storage where applications can be executed. Virtual machines were for years the virtualization standard to follow as it would be possible to achieve a resource isolation level even having multiple virtual machines running on the same physical machine. This process was achieved through a Hypervisor that is a software that allocates the physical resources to the virtual environment. Since each machine is a unique virtual environment, each one could be migrated at any moment for a different hardware. Containers are also a virtual environment but instead of running directly on the hardware its virtualization occurs at the host Operation System (Host OS) level, meaning that operation system and the Kernel are between this virtualization and the bare machine. The isolation in containers is done at a process level and each one will be created only with the minimum dependencies needed as libraries or code to execute that specific container. In the similar way to the VMs, multiple containers can run in parallel without being aware of each other's existence but typically each container will require less hardware resources than a virtual machine. Containers become the standard go to in cloud environments where modern applications consist of several containers together providing services.

2.4.1 Containers: Docker vs Podman

Two of the popular containerization tools are Docker and Podman. Docker is a well established tool that relies on a background daemon process to manage containers. When a client interacts with the Docker CLI or API, this client is communicating with this

daemon which runs with root privileges. If the Docker daemon becomes unresponsive the access to the containers can be compromised. Despite Docker's popularity this can raise security problems. In the thesis "A Framework for Supporting Privacy in the Computation of Biosignals" [19], a work also conducted in the BITalino context, the developed solution showed that some manners can be applied to mitigate this problem. Podman is another containerization technology engine that, in contrast to Docker, does not have the daemon process and runs the commands in the system in a rootless manner claiming to be more secure. This open source tool is an alternative to docker but offers the same commands and functionalities. This work will use Podman to handle containers as it is specified in Chapter 4.

2.4.2 Container orchestrator - Kubernetes

Once we have our applications in containers, it will be necessary to deploy them to our infrastructure. Container orchestrators are a component in modern cloud-native environments, serving as the management layer for containerized applications. Their purpose is to automate the deployment, scaling, networking, and lifecycle management of containers, which, when operated manually, quickly becomes unmanageable in complex systems, meaning that the central problems it aims to solve is resource allocation.

In the previously mentioned thesis [19], which was also conducted in the context of the BITalino device and its scripts, a comparison was made between two popular container orchestrators: Kubernetes and Docker Swarm. For this reason, the same comparison will not be carried out here. However, unlike Sebastião's thesis, this work will make use of Kubernetes.

Kubernetes [13] is widely adopted in the market and is an open-source orchestrator created by Google that enables deployment of containers within a cluster. The cluster itself is not created by Kubernetes, but tools for this purpose will be addressed later. The central elements of Kubernetes are the worker nodes and the control plane. The worker nodes run the containers with the applications, and these executions occur within Pods. Pods are defined as the most basic deployment unit in Kubernetes, where one or more containers can run together. The control plane is responsible for managing the state of the cluster and its worker nodes, ensuring that Pods run inside the worker nodes and launching them when necessary.

The control plane is divided into the following components:

- kube-apiserver: A REST API that exposes the control plane to receive requests from clients for cluster management;

- etcd: where all cluster data is stored, allowing the control plane to monitor cluster health and resource availability;

- kube-scheduler: The component responsible for assigning newly created Pods to worker nodes in the cluster. It makes scheduling decisions based on factors such as resource availability;
- kube-controller-manager: runs controllers that continuously monitor the cluster state through the kube-apiserver and make adjustments to ensure the actual state matches the desired state defined in manifests, the configuration files to be developed by us;

It is also necessary to mention the components of the worker nodes:

- kubelet: A daemon responsible for connecting the worker node to the control plane and ensuring that the Pods assigned to the node are running;
- kube-proxy: A network proxy responsible for routing traffic between Pods and maintaining network rules for Services;
- Container Runtime Interface (CRI): This component actually runs the containers inside the Pods and pulls container images from a registry such as Docker Hub or a private registry;

These components are particularly relevant to this work, especially the interaction with the kube-apiserver and manifest files. Further details are provided in Chapters 3 and 4. In Figure 2.3 we can observe the typical structure of a Kubernetes cluster. It is also necessary here to address some of the different types of Kubernetes manifest files.

Although this chapter is more focused on the state of the art to help us design the solution, leaving the technologies for Chapter 4, the use of a tool like Kubernetes by itself brings certain challenges. As an application scales, multiple configuration files need to be created, which can result in some of them having repeated settings, meaning that a simple update to a variable must be propagated across several files. Editing multiple configuration files increases the likelihood of errors. To assist with these tasks, Helm Charts will be used, as they help address these issues, and this is also one of the reasons Kubernetes was chosen for this work.

As already mentioned, the configuration files are used to describe the desired state of resources within the cluster. Typically written in YAML or JSON format, each manifest defines a Kubernetes object, with various types available depending on the intended purpose. This section provides a brief theoretical introduction to some of the manifests used in this work, whose implementation will be detailed in Chapter 4.

Deployment

Deployment manifest is one of the most commonly used resource specifications, designed to manage the lifecycle of applications running within Pods by usually pulling their containers from a image registry. Deployments provide a declarative way to define how many

replicas of a Pod should be running and have the hability to manage updates through rolling updates or rollbacks. This means that when a container image or configuration is changed, Kubernetes gradually replaces old Pods with new ones, ensuring minimal downtime and service continuity. If an error occurs, the Deployment controller can revert to a previous state. This behavior makes Deployments particularly suitable for managing stateless applications similar to BITalino scripts.

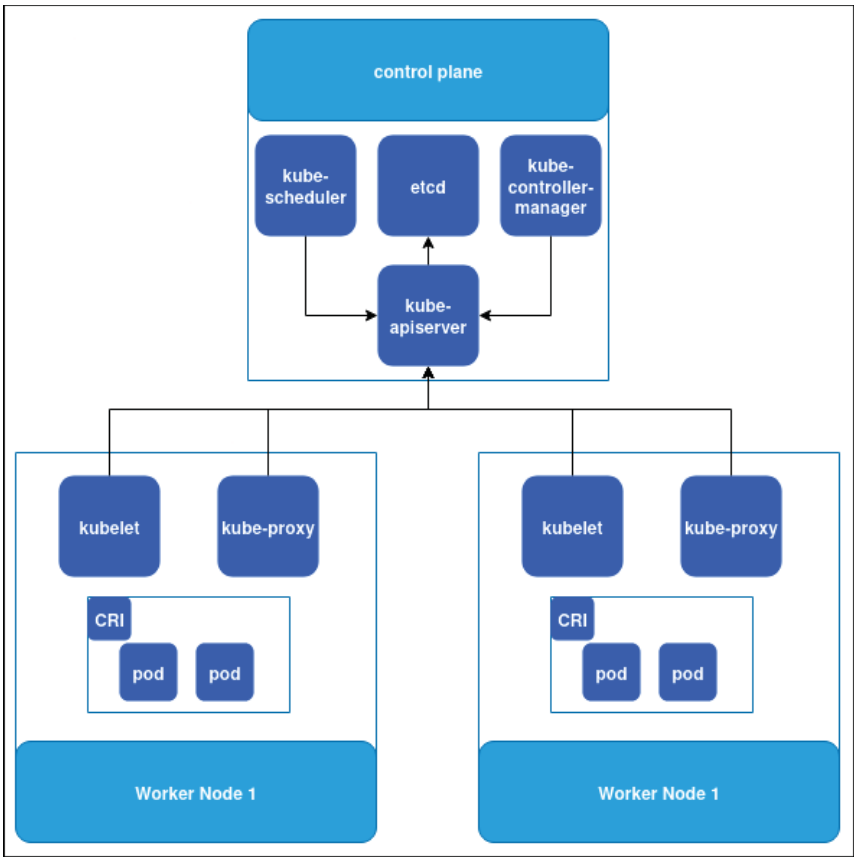

Figure 2.3: Kubernetes Cluster

Service

Since pods are ephemeral due to their lifecycle, a Service object provides a way to access a pod by exposing a virtual IP and an internal DNS name that allows traffic. There are three types of services, depending on how the pod should be available. As ClusterIp, the pod will be availble only for communication inside the cluster. On the other hand, NodePort is used to provide an external IP allowing traffic outside the cluster. At last, LoadBalancer services are used also in external connections but to balance the traffic between internal resources. So each service has its use case.

Ingress

Ingress manages external access to services in the cluster by providing HTTP(S) routing rules for incoming traffic. Unlike a LoadBalancer, which exposes a single service directly to external clients with its own external IP address, an Ingress acts as a single entry point by routing multiple services via a common external address. It requires an Ingress controller installed in the cluster to watch ingress objects.

NetworkPolicy

NetworkPolicies allow create rules for traffic flow between Pods or the outside world. By default, kubernetes allows communication without restrictions, which is particularly problematic in this case since each BITalino scripts may contain external calls. Measures will have to be applied in Chapter 4.

2.4.3 Infrastructure and Application Management as Code

It was previously mentioned that Kubernetes itself does not provide the computing resources on which containers run, which leads to the need for additional tools. In addition, manually configuring these components may result in failures, as multiple elements such as route tables or software installation dependencies must be handled whenever the infrastructure is deployed. Infrastructure as code (IaC) tools allow us to automate the creation of our infrastructure as well as the installation of its dependencies through configuration files. Several tools on the market follow the IaC concept, and this work will use Terraform [8]. This software allows us, through a declarative language, to indicate the objective we want for the installation of our infrastructure instead of explicitly indicating each step to be taken, since Terraform handles these details.

Regarding applications, relying only Kubernetes, which also uses code based configuration files, tends to present challenges as more platform needs arise. The number of configuration files easily increases, and repeated configurations end up being defined in different files, leading to changes having to be propagated in multiple locations. To facilitate this management, Helm Charts will also be used. Helm Charts [9] are a set of files that describe all the Kubernetes resources required to run an application. They maintain a history of installations and allow the creation of declarative templates, in which common parameters for different Kubernetes components can be defined.

2.4.4 Cloud storage

Data is a valuable asset in any application. One of the main points to be addressed in this work is the treatment given to the session output data produced by the python scripts of the BITalino team. In order for the drawn solution to be executed in a cloud environment

we now analyse typical solutions used for cloud storage. These types of cloud storage can be Ephemeral and Persistente. The Ephemeral type is usually to store temporary files because everything saved here is lost once this storage is shut down. A cache system is one of the use cases of this ephemeral storage. In contrast, Persistent storage ensures that the data remains available and can be divided into three types: block storage, file storage and object storage.

In block storage, data is broken into individual blocks that can be retrieved in parallel, allowing faster access. It is typically used for structured data, such as databases, whether SQL or NoSQL.

File storage type uses a hierarchical structure that consists of folders similar to the directories in an operation system. To retrieve data, the full path should be known. As use cases can be file sharing, archiving files or structured storing data;

Object storage is used for Big Data where each object can contain data, metadata and works with buckets. Can store unstructured data with different types, suitable for the output data from BITalino scripts.

2.5 OWASP Top 10

Although the primary focus of this work is not cybersecurity, it is essential for any software infrastructure to meet certain minimum security requirements, especially when dealing with users' private information. The Open Web Application Security Project (OWASP) [15] is a prominent non profit foundation dedicated to enhancing the security of web applications. Consisting of thousands of members worldwide, this community collaborates on open-source projects to address modern security concerns and foster knowledge-sharing among developers and organisations. Among OWASP's most well known projects is the OWASP Top 10, an initiative that identifies the ten most critical risks faced by web applications in today's digital landscape. This project serves as an important standard for designing and implementing secure software infrastructure. For each vulnerability listed, the OWASP Top 10 provides recommended approaches for prevention and mitigation. It is important to note that following these security standards is a way to ensure a response to the main articles mentioned above regarding the GDPR since the OWASP Top 10 is directly aligned with the GDPR. Identifying and addressing common security risks in web applications, ensuring that data processing activities are secure, and preventing unauthorised access to data are just a few examples of this alignment. For this work will consider the OWASP Top 10 released in 2021, and the following topics will provide a concise description of each vulnerability.

2.5.1 Broken Access Control

Access control describes the need, in any interaction with the system, for the user not to exceed any permission granted to him. This prevents unauthorised actions from being carried out and safeguards sensitive data critical for the system's proper functioning. By adhering to access control measures, the system maintains the necessary privacy and security levels. Access control plays a fundamental role in web application security, as it helps protect against various threats such as unauthorised access, data breaches, and privilege escalation. Through the implementation of strong authentication mechanisms, role-based access control, and proper session management, the system can verify the identity of users and grant them appropriate access rights. Some measures to help protect against unauthorised access is to deny access by default, i.e. authorization has to be expressly granted, log access failures or apply rate limits in calls to APIs.

2.5.2 Cryptographic Failures

Cryptographic failures are considered the second most critical vulnerability in application security and arise from weak algorithms or lack of cryptographic protection, potentially exposing sensitive data. This flaw highlights the need to verify where sensitive information is located in the system, as defined by regulations such as GDPR or similar, and to design mechanisms to protect it both where it is stored and in circulation. Here the OWASP foundation recommends that the use of protocols such as HTTPS be forced, as well as never leaving passwords or secrets unprotected or hardcoded in the source code.

2.5.3 Injection

This type of attack occurs when there are no validations for the data that can be introduced into the system. This lack of protection filters can lead to a malicious user being able to directly enter SQL commands or even OS commands. Some of the precautions to be considered to avoid this type of vulnerability are the use of sanitation inside APIs or, in the case of object oriented programming languages, the use of Object Relational Mapping Tools (ORMs) that serve as an extra layer between these programming languages and the data model, decreasing the need to write queries directly.

2.5.4 Insecure Design

It characterizes the failures in specific measures adopted for vulnerability prevention. These failures can occur at both the architectural and application levels. Some examples can be lack validations on users input or failure to implement security policies.

2.5.5 Security Misconfiguration

A vulnerability that is very present in cloud applications, which often reflects a mismatch between rapid development and secure configuration practices. It arises when cloud services, such as storage buckets, databases, or APIs, are left exposed to the public internet due to misconfigured access controls, overly permissive policies, or a lack of authentication mechanisms and leading to data breaches, unauthorized access to sensitive resources compromising cloud environments, making it a persistently high risk issue across industries. Since this work will give a great focus to the infrastructure layer where the applications will be executed, prevention measures such as denying by default to infrastructure elements will be implemented.

2.5.6 Vulnerable and Outdated Components

When we are building an application, especially nowadays, it represents a whole formed by a set of distinct components that work together. Any security breach in one of these components can lead either to the unavailability of this component or, in the worst case scenario, to a major failure of the system itself. A user of this application may not realize that there is an internal division into components, but an attacker will actively look for them and search for outdated or vulnerable libraries that may exist. Therefore, to avoid this type of attack, the system must include only the libraries strictly necessary for its functioning, keep them regularly updated, and ensure that unnecessary or unused features are removed.

Continuous integration and continuous delivery/deployment tools (CI/CD) that can scan the applications during compilation phase and before sending them to the cloud infrastructure typically can help detecting this vulnerability but these tools wont be the focus of this work. Our starting point will take into account containers already created and hosted in an image registry like DockerHub, but since the infrastructure must be able to handle multiple applications, a vulnerability of this type could occur at any time, and therefore tools capable of versioning and scaling will be essential.

2.5.7 Identification and Authentication Failures

Since most systems require identification and authentication as the first step for a regular user, safeguarding against unauthorized access is crucial. A system's authentication can be compromised in a number of ways, including automatic authentication attempts in which the attacker uses legitimate username and password combinations, brute force attacks, and, in the event that the system does not prevent a series of consecutive incorrect logins, the use of "admin" passwords. Implementing mechanisms to expire the validation of session tokens once the user logs out or becomes inactive is also important. Limiting or postponing a subsequent login attempt in the event of several unsuccessful attempts, as

well as avoiding the use of the system's default credentials, are some strategies to prevent these attacks. We must consider that our application layer will have multiple containers running programs developed by the BITalino team, so preventive measures for this type of vulnerability at the infrastructure level will be discussed in more detail in Chapter 4.

2.5.8 Software and Data Integrity Failures

Similar to Outdated and Vulnerable Components, software and data integrity failures occur when applications rely on components that, while not outdated, may still contain vulnerabilities. It is advisable to implement processes using a CI/CD pipelines to automate testing, using tools like SonarQube, or to ensure that all components remain trustworthy by being obtained through reliable sources.

2.5.9 Security Logging and Monitoring Failures

Logging all actions within the infrastructure provides full visibility into system operations, enabling the detection of potential security incidents.

All actions taken within our framework must be properly recorded to address the concerns raised and already presented in the GDPR.

It is also important to remember that private data should not be recorded in these logs, as this would be a significant breach of privacy. All logs of the software to be developed must be properly evaluated. Those who initiated the actions must also be properly registered.

Logging is essential in a production environment. Here we will only log some basic information but we will not use any specific technology for our proof of concept in Chapter 4 such as Elasticsearch or Loki.

2.5.10 Server-Side Request Forgery

Server-Side Request Forgery (SSRF) occurs when a server is lead into making requests to external resources or internal that the attacker should not have access to, acting as a proxy to reach other components of our system. The system should never trust URLs passed as input unless strictly necessary and sanitization and validation measures are applied. In our case, there are two major elements that can potentially increase this vulnerability: session data and BITalino scripts. Although in the swimming context, session files do not have URLs, this may not be the case in the future, either in swimming or in other contexts. Additionally, an isolated environment will need to be provided for script execution.

2.6 Summary

It was possible to confirm that several measures need to be taken to improve the processes for executing the BITalino team's scripts. The great versatility of the device in collecting biometric data across different scenarios emphasizes the need to update the environment surrounding this device, affecting data storage and script execution.

The procedures described directly conflict with regulations such as the GDPR and with recommendations from the OWASP Top 10. It should be reiterated that this is not a security focused study, but some measures will need to be implemented.

We have seen the advantages of tools like Kubernetes and why microservices architecture patterns have dominated the digital market for so long.

For the implementation of our solution, we will build a framework capable of running and creating an infrastructure that improves the BITalino team's procedures. Although we have analyzed Kubernetes, this tool alone is not sufficient to create a cluster in which we can run our microservices. It will be used in combination with Terraform, which ensures infrastructure as code and can provide all the necessary dependencies for Kubernetes objects. These objects will also be managed with Helm Charts.

Chapter 3

Specification and Design

To design an architecture that meets the needs presented so far, this chapter will define how the proposed system will be structured. It will discuss the approach taken to ensure that architecture aligns with both functional and non-functional requirements that were gathered during meetings with the BITalino team. The chapter begins by discussing two parallel works that were carried out alongside this one, which will help justify certain requirements that must be adhered to here. This is followed by a discussion of use case scenarios, illustrating how the system will operate not only in the swimming context previously introduced but also in other possible contexts as well. By the end of this chapter, all decisions regarding the conception of the system should be mentioned.

3.1 Initial requirements in BITalino context

Before detailing the requirements of the problem that this work aims to solve, it is important to introduce the general context in which it was carried out. Although the focus of this work is to provide an infrastructure where the scripts can be executed and their outputs persisted, there are several areas for potential studies that should be conducted before concluding all architectural and implementation choices with objectivity. This introduction is relevant because two other academic studies were conducted in parallel with this one, which led to the addition of certain requirements that must be respected.

The first work that needs to be mentioned has its focus on the integrity and security of user's session data and its storage. Here it is explored the advantages and disadvantages of different types of databases whether SQL, NoSQL, traditional or distributed and only the session data obtained by the BITalino device and some metadata is considered. Since a swimming session data is duly specify but the same does not happen with the script's output, as mentioned on the previous chapter, this results on the need to have at least one major decision: the infrastructure to be planned needs to have a data ingestion service that will receive the session data but won't be the same that will persist the script's output. This work can be seen in [3].

The second work made in parallel with this one aims to use a distributed Ledger network due to one of its main advantages, the immutability of data, which will be used to store privacy contracts, thus responding to one of the topics discussed in the previous chapter: authorization. These privacy contracts will contain the authorization details defined by the user, such as which other users can consult data, and allow the highest possible granularity decisions. The goal is to study strategies to ensure that, despite the immutability of the ledger, the user can at any moment change the authorization level of the data. Since this work focuses on authorization only, another relevant factor still needs to be discussed: authentication. As a result, authentication and authorization will be inside the same service, as it will be explained in more detail in this chapter, but only authentication will be discussed in depth. This work on authorization is still unfinished at the present time, and therefore it will not be cited in the bibliography.

In addition to these two works, it is worth mentioning again another previous one [19] where the scripts to be executed are obtained through a private container registry. Since this topic has already been discussed, for this work it will be assumed that scripts are already within containers and in a container registry that can be accessible by the infrastructure. The containers to be used will be detailed in this chapter and implemented in the following one. Although these decisions must be taken into account to achieve an overall design of the system, they will only be mentioned when relevant, and only the solutions to the problems of this work will be discussed in detail.

3.2 Stakeholders and use cases

With the current procedures, there are no actors with clearly defined roles. Since the scripts and the files containing the swimming sessions only need to be stored locally, any user can run the script and view the generated output, without the need for any access authorization. Therefore, the architecture to be developed will prevent these types of conveniences from being possible, and it will now be necessary to identify the actors who can operate the system.

System users will be the owners of the data not only resulting from measurement sessions with the BITalino device but also those produced after executing the scripts. All users must be able to register and authenticate in the system whenever necessary. During the requirements meeting with the BITalino team, the possibility of a user granting others permission to view and/or manipulate data was raised, provided they were duly authorized.

Additionally, these authorizations can be granted to both individual users and groups. In the context of swimming, which serves as the main example in this work, there is a common relationship between athlete and coach, or even multiple coaches. It is also common for team members to share the same level of permissions. It was decided on the

existence of individual users and groups of users, allowing permissions to be assigned to either individuals or groups, without defining specific roles such as 'Coach' or 'Athlete.' This allows the system to be adapted to other contexts like in Medical care, where the relationship would be between 'Patient' and 'Doctor.'. Figure 3.1 illustrates what is expected of a user to be able to do.

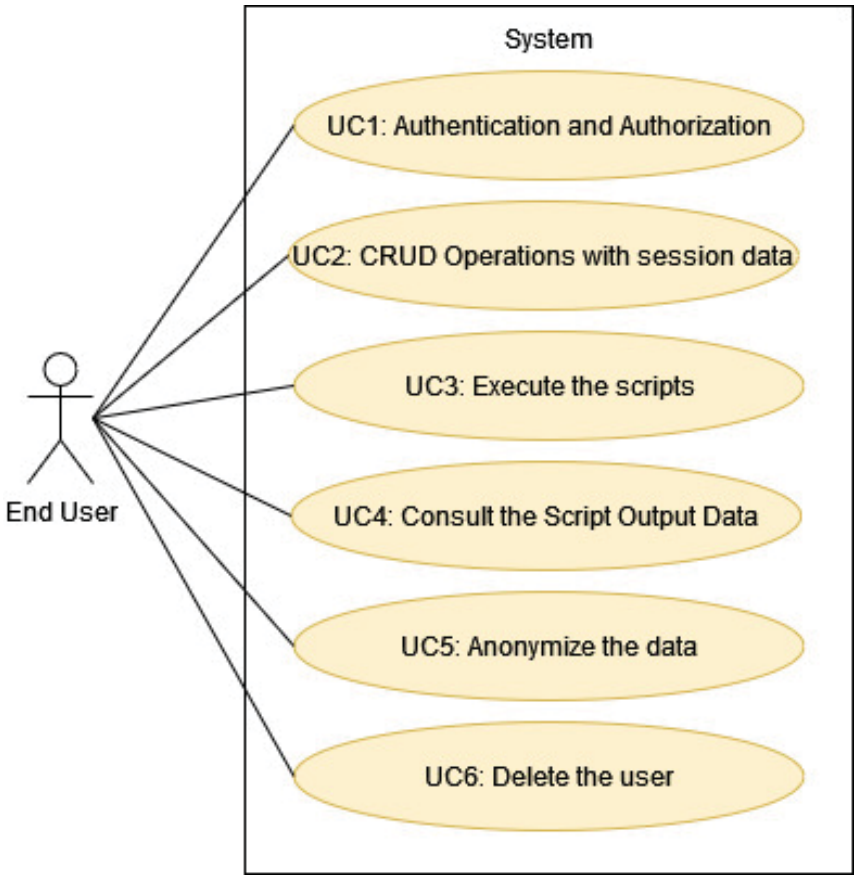

Figure 3.1: Use case diagram

3.3 Requirements gathering and analysis

In addition to use cases, requirements gathering is another essential step in defining the scope of the problem we are solving. To ensure this work supports the BITalino team's needs, functional requirements (FR) and non functional requirements (NFR) will be presented here.

Functional requirements (FR)

FR1 Different types of Session Files: CSVs generated by measurement sessions with the BITalino device can have different columns, depending on the context or sensor used. The system must be able to handle different types of input.

FR2 Different types of Output Data: Depending on the script used, the resulting output can be in any format such as images, PDFs or JSON structures.

FR3 Avoid repeated processing: different executions of a script with the same input will produce the same output. Measures should be implemented to avoid unnecessary execution.

FR4 Different sizes of output data: scripts can produce small or large files.

FR5 Script execution can be synchronous or asynchronous: The system needs to ensure that scripts with different execution times can be executed. In long executions, it is not possible for the user to wait for them to finish, so measures will have to be implemented.

FR6 Script deployment on demand: a rarely used script should only be executed on demand to avoid services running without requests.

FR7 Keep statistical data: Unlike what happens with data deletion, anonymized data should remain in the system for statistical use.

Non functional requirements (NFR)

NFR1 Availability: Both scripts and other supporting applications must always be accessible to calls.

NFR2 Scalability: The system must be capable of handling an increased number of users and, consequently, a larger volume of data. It should also be possible to scale the system to other contexts beyond swimming.

NFR3 Portability: the system must work on premise and in a cloud environment.

NFR4 Performance: the solution must not significantly increase processing time when compering to the current system.

NFR5 Traceability: it must be possible to identify the script that originated the data as well as the user who initiated the request.

NFR6 Maintainability and Operability: The system must be designed so that individual microservices can be deployed and scaled independently. All infrastructure must be defined as code.

NFR7 Fault tolerance: The system must continue to operate correctly even if some services fail.

NFR8 Comply with GDPR: the articles mentioned in 2.2.1 should provide guidelines to protect users data.

NFR9 Comply with OWASP Top 10: mechanisms to avoid the discussed vulnerabilities in 2.5 must be implemented.

3.4 Architecture Design

With the use cases and system requirements presented, it is now necessary to design an architecture that helps meet these needs. The BITalino team's scripts already offer several functionalities to be included in the system, but the focus of these scripts is always the processing of swimming sessions, or other contexts as already mentioned, and the generation of new data.

There are therefore several key elements that must be designed and implemented, and that are common to all interactions between system users and scripts. These processing units will be part of the Application Layer. It is important to mention that some of the applications included in this layer will need to use external services, which will be duly indicated. On the other hand, the applications to be described in this chapter require an infrastructure where they can be executed and which guarantees the non-functional requirements already mentioned. This infrastructure will also have several key components and will be part of the Infrastructure Layer. Although the frontend of this system is not part of this work, a Presentation Layer will also be indicated as a representation of the end user. Figure 3.2 provides a high level representation of the architecture, which will now be detailed. This will follow the same Generic Architecture already proposed in [19] but here each layer will have different components.

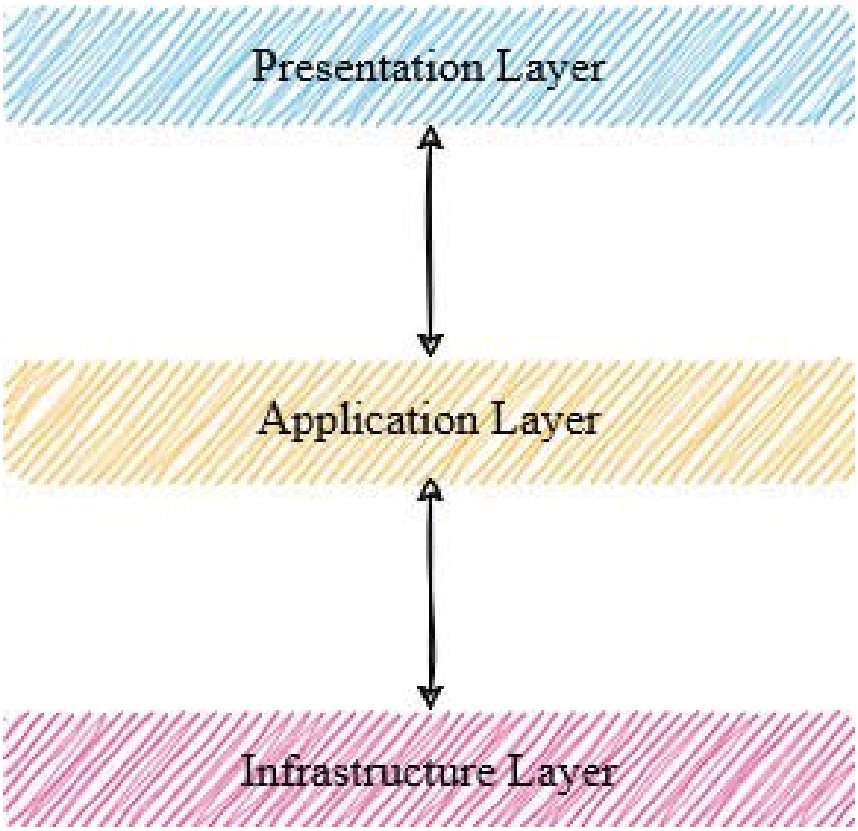

Figure 3.2: Architecture Overview

3.5 Application Layer

The Application Layer serves as the core of the system, where the business logic and essential services are implemented. It follows a Microservices pattern. As mentioned in Chapter 2, this pattern is a standard in cloud environments and provides many advantages for this work, especially in scalability. The communication between the services is done through RESTful API calls.

Although the goal of this chapter is to propose a solution at the architectural level, leaving the choice of technologies and implementation details for Chapter 4, it is important to emphasise that the proposed solution should be as modular as possible. This ensures that any future changes in technology or applications will have minimal impact on existing elements.

Considering the information provided in 3.1 and software requirements 3.3, it was found that the Application Layer is actually divided into three quite distinct areas. The first one represents the need to handle user authentication and authorization within the system. Secondly, there is the need for session data ingestion. Lastly, it must allow scripts to be executed and their output stored and made accessible. For these reasons, it is proposed that this Application Layer consist of three distinct modules: a Security Module, a Session Module, and a Script Module. This division aligns with the system's requirements and ensures that each aspect of the system is handled independently. Each module has an API that will be the entry point for the services that a module offers. Additionally, this layer interacts with external services, such as databases and storage systems, while adhering to security best practices to ensure that data remains protected.

As an entry point for this layer there will be an ingress controller, already discussed in chapter 2. An API gateway is also used in conjunction as it offers routing between services and more options such as authentication and authorization.

3.5.1 Security Module

The Security Module is responsible for managing user authentication and access control across the system and it was created with the goal of addressing use cases UC1 "Authentication and authorization" and UC6 "Delete the user" mentioned in Figure 3.1. Figure 3.3 is an overview of this module.

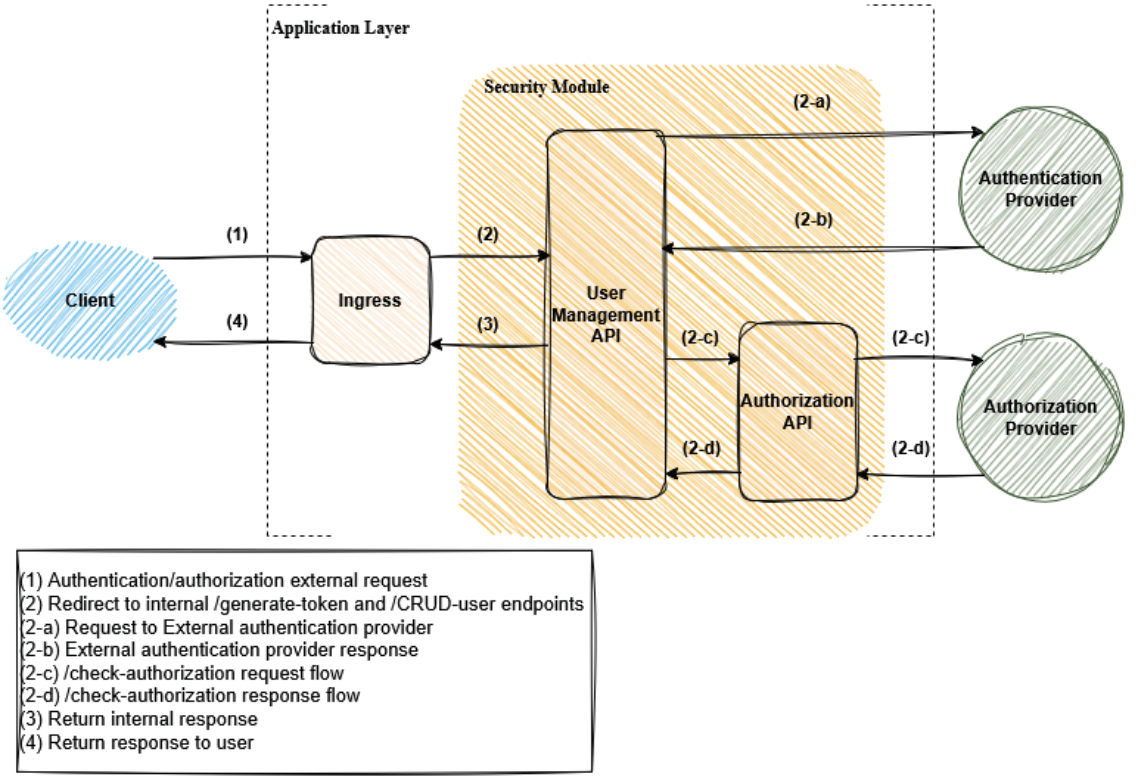

Figure 3.3: Security Module operation's flow

Security components description

The user-management API is the entry point to this module. All services that require authentication/authorization will obtain it through this API and never directly from external providers.

The authentication provider, to be determined in Chapter 4, will be where users' login data and profile information will be stored. This provider will not be implemented in this work, but research has been done to use reliable market tools for this purpose. It will also be responsible for generating JSON Web Tokens (JWT) that will serve as a login system.

It is worth noting that authorization is not implemented in this work, however, it is important to include it here because connections will be made between the main APIs for user management.

3.5.2 Session Module

It is in this module that users will be able to register the data coming from the BITalino device. The potential use cases of the resources API will not be explored here, as it is part of another study, as indicated at the beginning of this chapter. However, there are two factors relevant to the need for this module in our work: access to session data and metadata, and the necessary infrastructure level configurations.

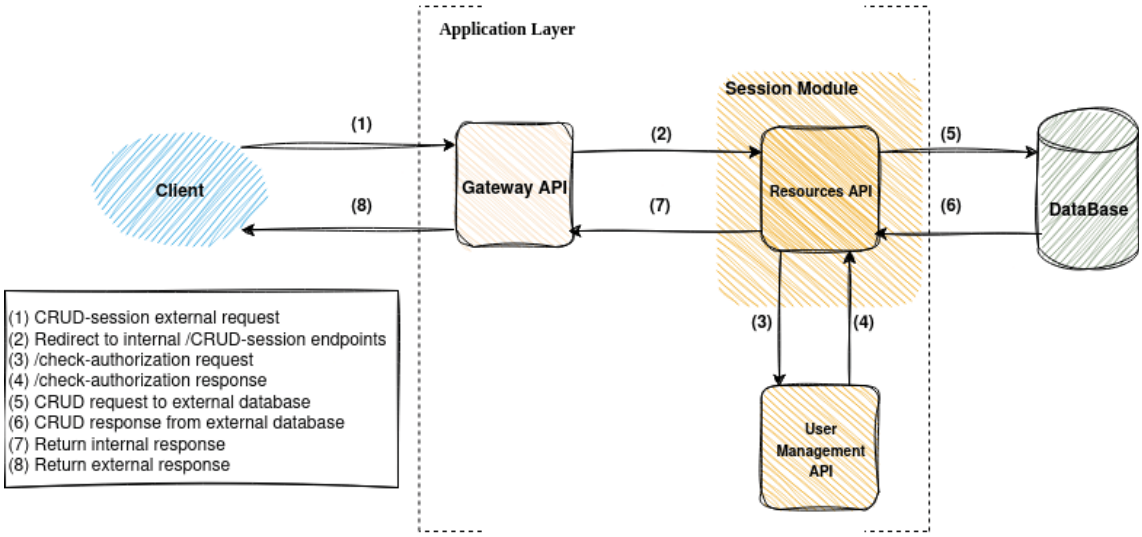

Figure 3.4: Session Module operation's flow

Session Module components description

All data that needs to enter the system first passes through this module. This is where data sanitization will be applied, when necessary, for subsequent processing by the scripts. The BITalino device produces a CSV file, which, after being converted to JSON at the presentation layer, will be stored in the external database outside the cluster via the Resources API. For the proof of concept here, it will only serve to store session data without additional analysis.

It is also important to mention that the Resources API needs to be able to verify authorization through the User-Management API and access to an external database.

3.5.3 Script Module

Although authentication is relevant to this work, this module is the main component implemented at the application layer. Within this module, mechanisms must be implemented to enforce the use cases and functional requirements related to the scripts and the output data of the sessions. Since the types of scripts that may be developed in the future are unknown, it is advisable to approach the problem as if access to the scripts' source code is never available, necessitating more generic and adaptable implementations suitable for multiple scenarios.

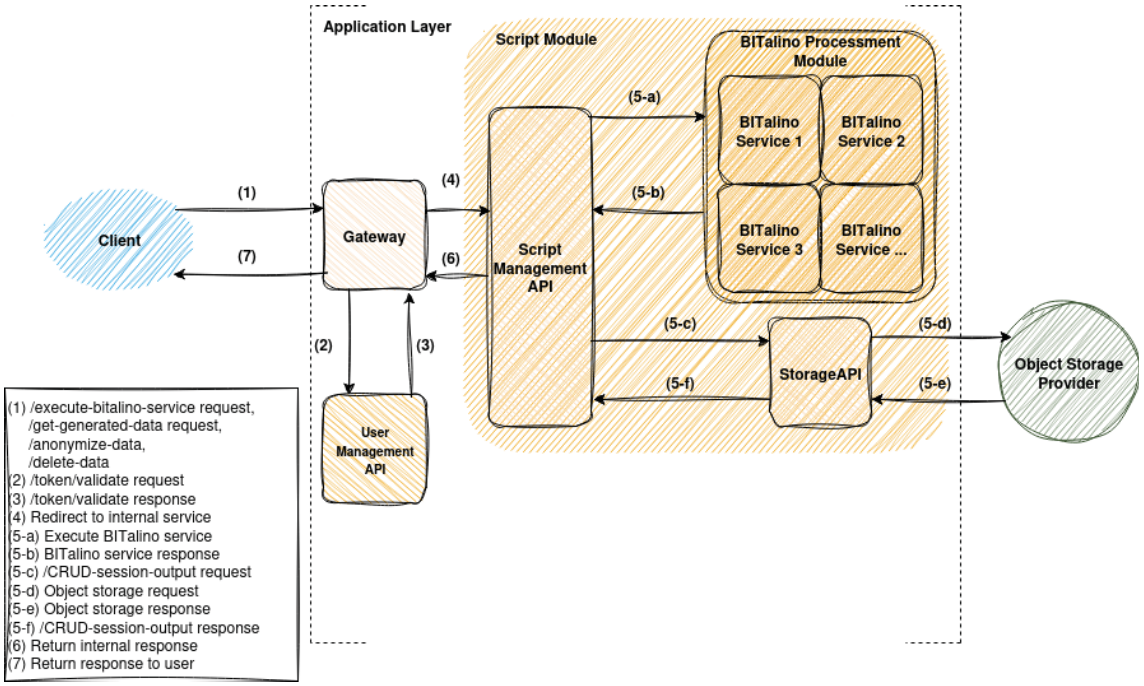

Figure 3.5: Script Module operation's flow

Script Management API

Similar to User Management API and Resources API, this one will serve as the entry point for functionalities related to the scripts developed by the BITalino team. It will be responsible for enabling users to not only execute the scripts but also access the outputs generated by them.

It is in this service that the logic will be implemented to ensure that the output of the scripts is generated only once, as indicated in FR3. This need arises because it was not possible to define the context in which the scripts receive requests to be executed. Execution requests may come directly from the presentation layer or be initiated by internal automated processes that can be scheduled or triggered sporadically. For these reasons, some internal mechanism will need to be implemented to avoid repeated processing.

To make this mechanism possible, two characteristics of the scripts, not yet mentioned, were considered: they are deterministic and idempotent. A deterministic service means that when the service receives a certain input, it will always produce the same output. On the other hand, an idempotent service guarantees that when executed, the consequences of that execution on the system will be the same, e.g., typical GET endpoints in an API. It can be argued that in the case of the script provided for analyzing swimming sessions, described in 2.1.5, its execution locally will not be idempotent because simultaneous executions of the same script, using the same input, will produce multiple files despite being identical. However, it is precisely this problem of repeated outputs that we want to prevent.

Based on these two principles, it is possible to define a hash function that will serve as a processing ID resulting from a certain script having processed certain sessions as input. There are still important considerations that need to be made for this solution to be viable, and they will be addressed in more detail in Chapter 4. However, this will be an important element of this API.

This API will have another important role but it will be necessary to address BITalino Processment Module first.

BITalino Processment Module

The essential role of this BITalino Processing Module is to fulfill Use Case 3, "Execute the scripts". It is not possible to guarantee that all BITalino scripts will be executed within an acceptable timeframe without affecting the system's usability, since the system must be designed to handle a high volume of input sessions. As a direct consequence, there is a need for both synchronous and asynchronous scripts. The Script Management API will therefore have to provide endpoints for this purpose, which leads to another important implication for implementation: BITalino scripts must be able to make REST requests to the Script Management API so that the generated results can be persisted. Since these calls are completely common in a cloud environment, it is again necessary to remember that BITalino scripts are a potential source of vulnerabilities, given that it is not possible to anticipate the types of scripts that may need to be hosted. Therefore, this module must provide an environment that is as much isolated as possible, where the scripts are executed and can communicate only with the Script Management API.

In addition to execution times and isolation, it is also necessary to consider that there is no well defined periodicity for the potential use of each BITalino script. As a result, these services might be running and consuming resources unnecessarily, since some services may receive multiple calls per minute while others may receive only a few calls per week. This factor intersects directly with a critical aspect of cloud environments: the financial aspect.

Thus, a generalization was made, resulting in three categories of script based on their availability:

Permanent scripts: These must always be ready to receive calls and should be available as soon as all other system components are ready. Since these scripts now receive session input through REST calls, the existing code will need to be adapted, as up until now the data was passed as arguments via the local CLI.

Scheduled Scripts: Scripts that remain inactive until a pre established moment for their execution. Instead of being triggered by a client call through the Script Management API, the system itself must handle the launching. These scripts do not need to be able to receive calls but will need to obtain session data for processing, which requires adapting the existing code.

On-Demand scripts: without a specific schedule, whose processing request volume does not justify being always available. Like the scheduled, these are launched by the system but are triggered by a user request. Adapting the code for these can be more specific, as they may either receive input data directly or have to obtain it themselves through REST calls.

This categorization of scripts is essential because each category raises distinct challenges relevant to implementation. All of these challenges will be discussed in the following chapter.

Storage API

This component will be an internal element of the cluster and will be responsible for storing the data generated by BITalino scripts in an external object storage provider. In Chapter 2, different approaches to handling storage in the cloud were discussed, and from that analysis, the decision was made to adopt Object Storage. This solution addresses the need for the system to handle large volumes of unstructured data, due to the unpredictability of BITalino scripts, with interaction carried out through APIs. It also has the advantage of allowing files to be stored together with a set of metadata, which will be useful as indicated below.

Since data growth can easily be exponential and this solution is organized in a file system like structure, it becomes essential to specify the internal organization to be adopted in the object provider. First, it is necessary to define the number of buckets to be created as well as their use cases. One intuitive approach could be creating one bucket per user, however, better alternatives exist. Although some object providers do not impose a limit on the number of buckets, it is recommended that this number should not depend directly on a factor that can be easily increased as the number of users in a system. The creation of three buckets is suggested: one for user data, a second for statistical data, and a third for internal records.

The first bucket will contain user data, where each user will have their own directory. Even within a bucket, the internal structure remains essential to facilitate operations on the data, since an excessive number of subdirectories may degrade system performance. As the execution of each BITalino scripts can generate a variable number of files, it is convenient to group the data of a given generation in its own directory. For this, the existence of something similar to a generation ID is relevant, so that this becomes the name of the directory. Since all requests reaching this API originate from the script-management API, it is then possible to use the generated hash, described in 3.5.3, as the directory to access the data produced by the corresponding execution. This approach also addresses FR3, because before the script-management API executes a script, it will first query this storage API to verify whether that hash already exists. This procedure is feasible due to the deterministic and idempotent characteristics of the script. If the

hash already exists, the previously generated data will be returned, and the script in the BITalino processing module will not be executed.

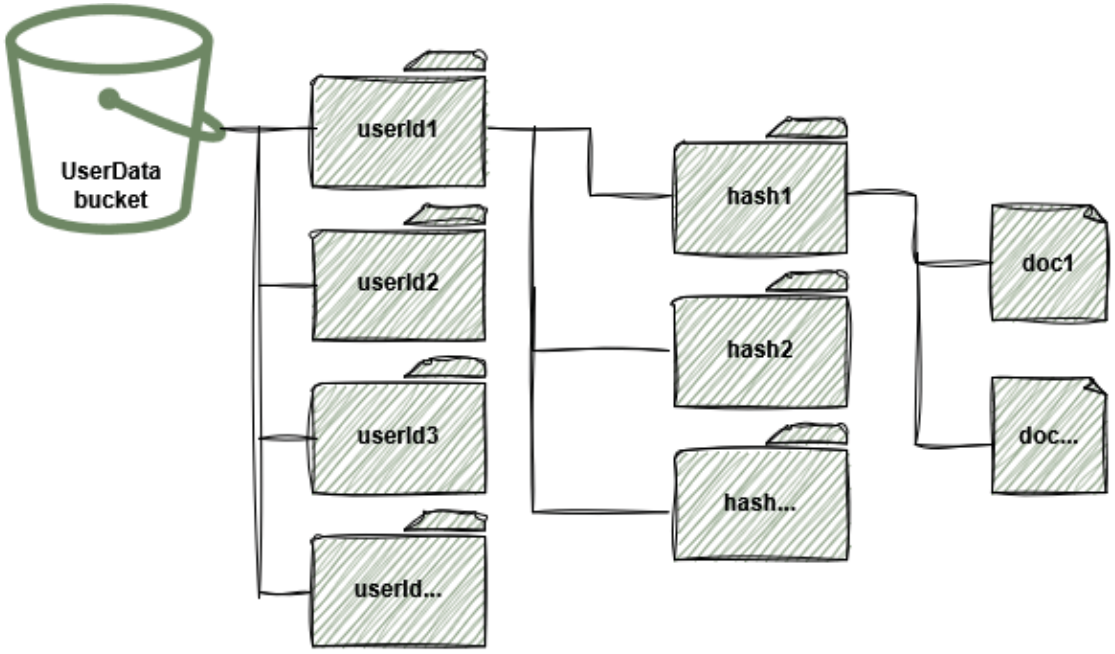

Figure 3.6: User's Bucket

The second bucket will also contain data resulting from BITalino script generations, but which are not associated with any user. Due to the versatility of the device for data collection, it is possible that some data will not contain information capable of identifying its owner and can therefore be used for statistical processes. This bucket will also store data when a user requests anonymization, as indicated in UC5. To enable this, complementary information must be stored together with the generated files, serving as a flag indicating whether this data is sensitive according to the GDPR.

The context in which this flag is introduced into the system has not been defined, but for the purposes of this work we will simply assume that it arrives at the script-management API together with the data itself and will be stored as file metadata. The correct categorization of whether input files contain sensitive information is not something that can be easily automated. It was assumed that when this flag enters the system, it has been properly evaluated either by the user who provided it, although this is not a reliable solution, or by BITalino team.

When users wishes to disassociate themselves from their files, they have the option of deleting them or, if they authorize, allowing them to be used for statistical purposes. The metadata is then verified, and if possible, the files are migrated from the first bucket to this one. Since this data is not associated with users, this bucket will be organized into subdirectories based on the script that generated them.

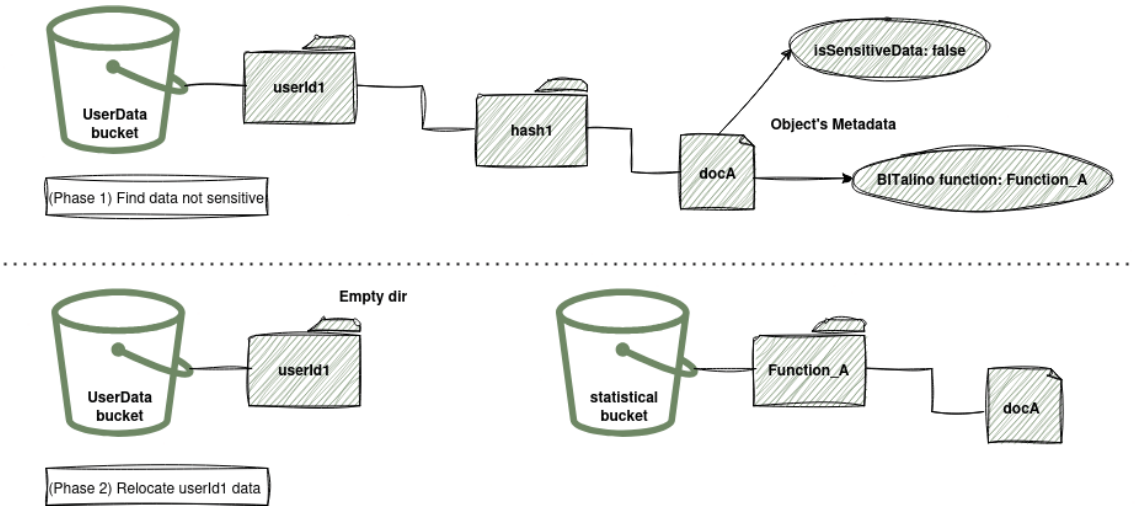

Figure 3.7: Statistical Bucket

The third bucket, will not have functionalities that affect system usability like the previous ones but will serve as a utility bucket where execution logs can be stored for auditing or troubleshooting purposes.

3.6 Infrastructure Layer

This layer is responsible for allocating resources so that the application layer can be executed. As mentioned in Chapter 2, it will be orchestrated through Kubernetes components, however, here a conceptual division is proposed, similar to what was done for the application layer, since there are differences in the permissions granted to certain services.

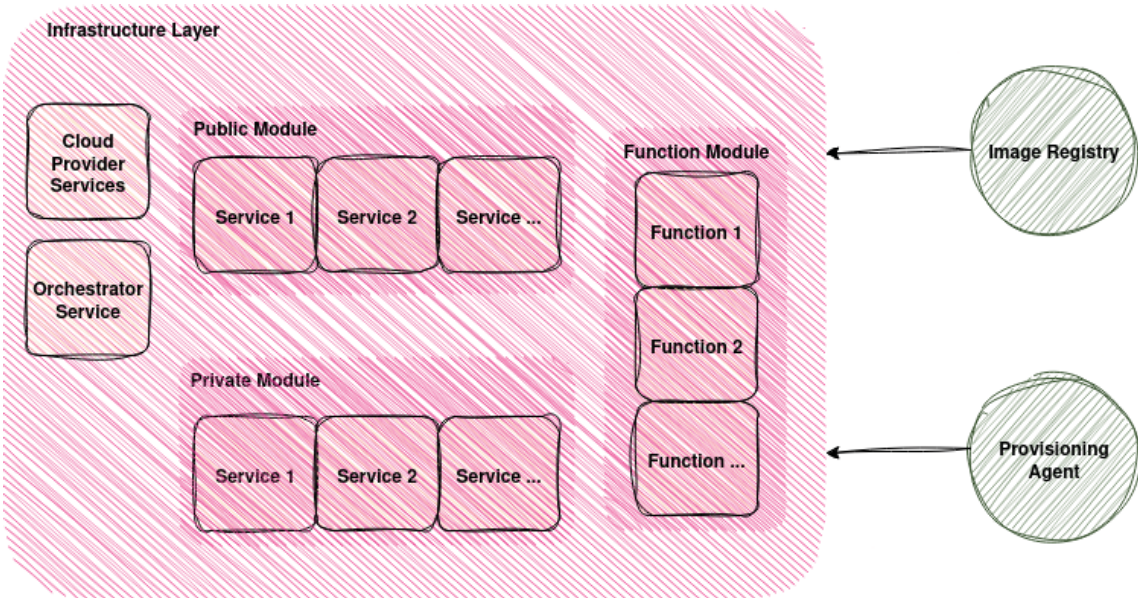

Figure 3.8: Infrastructure Layer

3.6.1 Function Module

This module represents the environment created for running scripts. Here the resources to execute, launch, or terminate code will be provided. Strict networking policies will be applied so that the only possible communication is with the Script-Management API.

All script execution components are isolated in a dedicated environment, ensuring that they can interact only with authorized services. Incoming and outgoing communications are tightly controlled, allowing scripts to perform their tasks while preventing unauthorized access or data exposure. This design enforces a zero-trust model, ensuring that code execution is secure and does not interfere with other parts of the platform.

It was considered that it is not possible to have access to the source code to be deployed here.

3.6.2 Private Module

This module will run services that cannot receive calls from outside the cluster. This division aims to separate sensitive components such as the storage API, which directly accesses the object storage where the output data is located. This should be the only API that accesses object storage.

For this work, only the object storage access API was developed, however, future APIs that should also not be exposed should belong to this module. An example of one such service is the authorization service, a gateway to the ledger, as mentioned in section 3.1.

3.6.3 Public Module

Here are the central services described in the application layer as the entry points to the modules. These services provide the endpoints to be consumed by the clients (presentation layer). Each service will have an IP address for access, however, it should remain internal, requiring all requests to pass through the Ingress controller.

3.7 Summary

With the use cases and requirements analyzed, we were able to make decisions regarding the design of the solution. The solution will follow a cloud-based microservices architecture, with three of them providing direct functionalities to the client: user management, session data management, and script management. We maintained the approach of keeping the scripts in an environment as isolated as possible due to the unpredictability of their functionalities. However, a communication flow had to be planned in order to enable data to be sent to and retrieved from them.

The Figure 3.9 presents a draft of the application and infrastructure layers to be implemented.

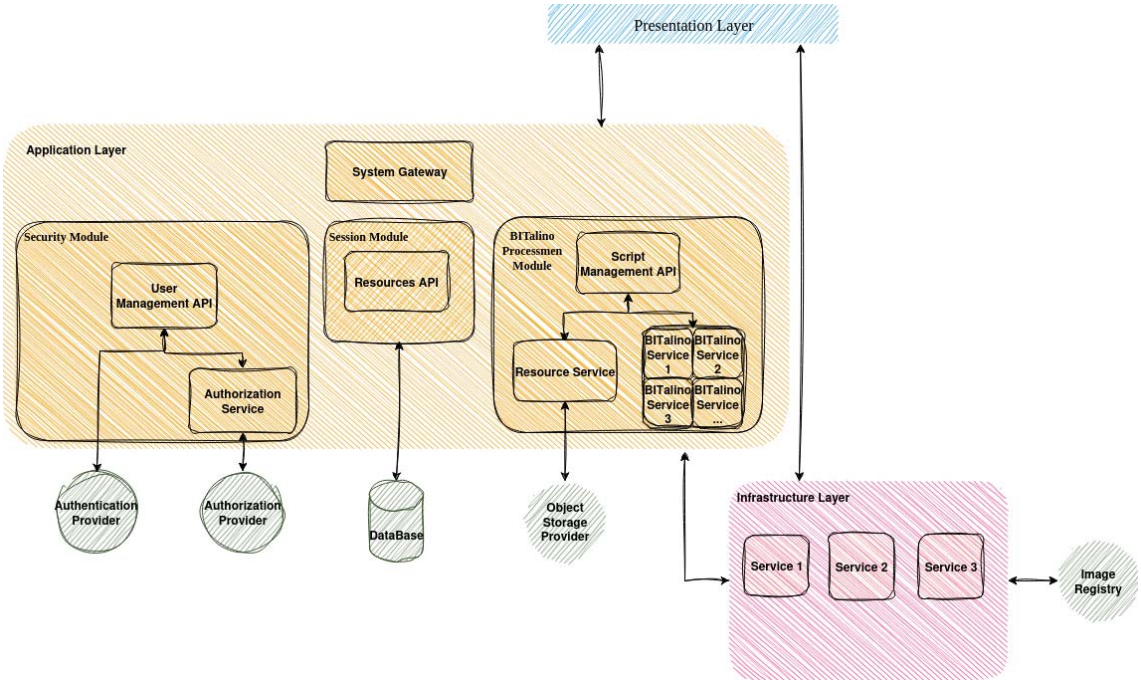

Figure 3.9: Complete Infrastructure

Chapter 4

Implementation and deployment

With the design of our solution defined, we will now turn to the detailed technical components of its implementation. This work aims not only at the application level implementation but also at the deployment of the infrastructure itself, which requires several configuration files and supporting software. Some configurations will be detailed here, but we will analyze only the relevant elements of these files. All configuration files are provided in detail in the Appendix section.

A Kubernetes cluster will be created, and the applications already presented will be deployed and executed there, including the BITalino scripts. This implementation will serve as a proof of concept, meaning there are limitations that would not apply in a real production environment, such as restricted scalability due to a minimal node pool configuration or the use of self-signed certificates instead of certificates from a recognized authority. Additionally, for this demonstrator, a trial account on Google Cloud will be used, which means a project was created in Google Cloud Platform (GCP), resulting in a projectId. This projectId will be required to provision the cluster with Terraform and was obtained through the Google web control panel. Different cloud providers may work differently, but most of the software presented here will run with minimal changes, even if executed on-premises.

In the previous chapter, we introduced the application layer followed by the infrastructure layer. However, it now makes sense to reverse this sequence, starting with a cluster where the applications can be executed.

4.1 Infrastructure Layer / Cluster

Starting with the cluster creation, we will now provision the core infrastructure resources required for the platform.

From the start of this project, it has been stated that the objective is to provide a cloud-compatible architecture without being dependent on any specific cloud provider. The following tools will assist in this task.

Since we are using a GCP we need to install gcloud, the command-line interface tool for interacting with Google Cloud services. After that we can configure our local CLI and use terraform and our helm charts to create the cluster.

4.1.1 Terraform

Terraform will be the tool to create our cluster in GPC.

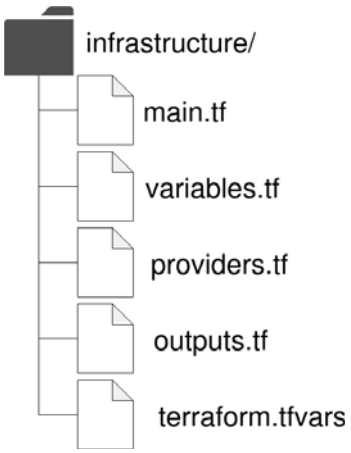

Figure 4.1: Terraform work directory

The Figure 4.1 shows the typical structure of a terraform working directory. The main.tf is the core of the infrastructure definition and contains information as the number of nodes and the server region. In our case will also contain important resources for our work that are detailed below. The variables file contains the variable declaration required as parameters for our cluster. May also have some default values but usually project specific variables are in terraform.tfvars. The output.tf expose important information so it can be used in modules or just as print it. Is with providers.tf that terraform will download the correct provider plugin, in this case Google, and connect with it by an API. In this case it also contains Kubernetes and Helm providers to allow terraform to prepare the cluster with what we need.

Terraform has three main commands to create the cluster. `terraform init` prepares the working directory. `terraform plan` generates an execution plan showing what Terraform will perform without any change. `terraform apply` applies the plan.

Terraform Resources

One of the advantages of Terraform is that it allows us, once our infrastructure is created, to automatically add resources that we know are necessary for the operation of our applications, eliminating the need to add them manually. For this work, three resources were considered:

- `ingress_nginx`
- `cert_manager`
- `calico`

Ingress_nginx deploys an Ingress controller that serves as the entry point to our cluster, controlling traffic coming from outside the Kubernetes cluster to internal services. All services to be deployed will be blocked from direct external access unless they pass through this controller. These communications will enforce the use of HTTPS, handling SSL/TLS certificate termination, which is essential for compliance with GDPR and the OWASP Top 10.

Cert_manager allows automated certificate management. It provides the certificate authority (CA) and certificate lifecycle needed for Ingress SSL/TLS and for internal encrypted communication.

By default, Kubernetes allows all pods to communicate with each other, which is why NetworkPolicies are required, as discussed in Chapter 2. Some cloud providers, such as GKE, do not enforce network segmentation between pods, meaning that even when NetworkPolicies are applied within the cluster, they are not actually enforced. For this reason, we need to install Calico in our cluster. While Calico supports a wide range of configurations, in this context it will be used specifically to enforce Kubernetes NetworkPolicies in our cluster. All Terraform files are in Appendix A.

4.1.2 Helm Charts and Kubernetes

Since we need to handle a considerable number of kubernetes objects our solution uses custom Helm charts. Each service specified in Chapter 3 has particularities that are now to be analyzed. One of the advantages of using Helm charts is that they allow grouping common properties into configuration files, which can then be used by different services. If necessary, more specific properties can be overridden in the Helm chart of the respective service. For this reason, it is beneficial to carefully reflect on the main Helm charts to be developed.

The organization proposed is shown in Figure 4.2, where each of the main services in our application layer will have a corresponding Helm chart. There is also a chart in the common/directory that will contain some of the common properties for all services and will have dependencies on all the others. At the time of deployment, simply executing this chart will ensure that all other services are deployed with just one command. Examples of this execution can be found in Appendix C.

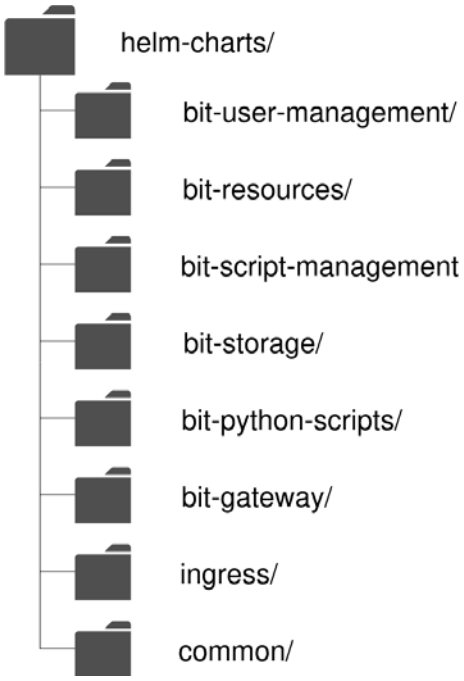

Figure 4.2: Helm work directory

Each directory contains a Chart.yaml file, which is a mandatory Helm file required to install its properties into the cluster. Then, each directory contains a /templates subdirectory, where several files are located to configure Kubernetes objects, as described in Chapter 2 2.4.3. Since there are many files, here we will discuss how each of the objects was implemented at the infrastructure level.

For the purpose of standardization, the prefix 'bit-' has been applied to all developed services.

Kubernetes objects overview

In Table 4.1, we can observe the objects resulting from the application of the Helm charts from our work.

Kind	Names	Context
Namespace	bit-platform-public bit-platform-private bit-python-scripts ingress-nginx cert-manager	Internal organization proposal, different traffic rules will be applied.

Kind	Names	Context
Deployment	bit-gateway bit-user-management bit-resources bit-script-management bit-storage scriptflaskone scriptflasktwo	All the services implemented
Service	ClusterIP	Used for Deployments be available only inside the cluster
ConfigMap	app-config scriptcron-config scriptjob-config	Used to pass variables into deployments
Secret	minio-credentials	bit-storage do not have hardcode the user and password
Ingress	bit-ingress	First layer between the cluster and client. It will handle routing and ensures https
ServiceAccount	job-launcher-sa	To allow bit-script-management to interact with K8 API and launch Jobs
Role/RoleBinding	job-launcher-role job-launcher-binding	To set the possible operations of job-launcher-sa
CronJob	scriptcron	The scripts that will be executed periodically

Table 4.1: Kubernetes objects

Public Pods

External access to our Kubernetes cluster will be possible through two components: an Ingress controller and an API gateway. The Ingress controller handles incoming HTTP/S traffic and routes it to the appropriate services inside the cluster, while the API gateway provides additional features such as authentication, rate limiting, and request aggregation. Together, they ensure secure and controlled access to the cluster's applications.

For the bit-user-management, bit-resources, and bit-script-management services to be accessible outside the cluster, they need to be exposed through the Ingress and API gateway. By using a corresponding URL prefix (e.g., /bit-user-management), the gateway is able to route the requests first to bit-gateway for token validation and then to one of the available pods of that microservice. This is the only possible external communication path to these pods, since in their `service.yaml` files they were configured with ClusterIP, which allows only internal direct calls within the cluster.

As TLS is enabled via cert-manager, all external connections are secured with HTTPS. Any attempt to use HTTP will be redirected to HTTPS.

Private Pods

With the exception of the BITalino scripts, the bit-storage service is the only private service, providing core functionalities to our system. Its `service.yaml` is also configured as ClusterIP, but it is deployed in the bit-platform-private namespace. A NetworkPolicy called deny-all-to-private was created to block all connections to the bit-platform-private namespace by default.

Despite this, we still need to allow bit-script-management to interact with bit-storage, so a strict network policy can be seen in helm-charts/common/values.yaml.

```
networkPolicies:
 enabled:  true
 rules:
 - name:  allow-script-mgmt-to-storage
 source:  bit-script-management
 target:  bit-storage
```

Script Environment

The original BITalino team script was converted to represent the Permanent BITalino scripts already discussed in the previous chapter. In addition, three more Python scripts were developed to serve as proof of concept.

The original BITalino script already produced the output shown in Figure 2.1, generating an image file. Another Python script was created to generate two PDF files. These scripts are referenced in the configuration files as scriptflaskone and scriptflasktwo and are both Permanent BITalino scripts.

Again, since scripts are a common source of vulnerabilities, their environment must be as isolated as possible. Only the bit-script-management API is allowed to interact with these pods. To enforce this, a NetworkPolicy was created in /bit-python-scripts/templates/networkpolicy-flask.yaml, which ensures that only requests from the bit-platform-public namespace and pods labeled bit-script-management are allowed.

Furthermore, the same NetworkPolicy also includes an `egress:{}` property, preventing scripts of this category from making requests to any other services. All requests to this namespace must be started exclusively by bit-script-management.

There was also a need to deploy a script representing the Scheduled jobs. A script called scriptcron was created for this purpose. Unlike the Permanent scripts, these are launched by Kubernetes CronJobs, and therefore must obtain input data and deliver output to bit-script-management. In the file bit-python-scripts/templates/networkpolicy-cron.yaml, the spec.egress was configured so that requests can only be sent to bit-script-management, which is the only allowed connection.

Finally, the On-Demand Scripts are launched on demand through the bit-script-management API. A third script called scriptjob was created for this purpose. This API makes a direct call to the Kubernetes API to launch a pod running the scriptjob. A corresponding file, bit-python-scripts/templates/networkpolicy-job.yaml, was also created, which, like networkpolicy-cron.yaml, only allows calls to the bit-script-management API.

It is important to note that direct calls to the Kubernetes API can introduce security risks, so additional measures must be implemented. A ServiceAccount was configured and associated with the deployment.yaml of bit-script-management so that this API is the only one authorized to make calls to the Kubernetes API.

Additional Configurations

The deployment files also include security properties such as:

```
securityContext
runAsNonRoot:  true
runAsUser:  1000
readOnlyRootFilesystem:  true
allowPrivilegeEscalation:  false
```

In the deployment manifests, a securityContext is defined for each container to enhance security and follow best practices for running applications in Kubernetes. runAsNonRoot ensures that the container does not run as the root user. Running as a non root user minimizes the impact of potential security vulnerabilities within the application, as an attacker would not gain root privileges if the container is compromised.

runAsNonRoot: 1000 explicitly sets the user ID under which the container runs. By specifying a fixed non-root UID, the container operates with restricted permissions, reducing the risk of unauthorized access to the host filesystem or other containers.

The third property, readOnlyRootFilesystem, mounts the root filesystem of the container as read-only. By preventing write access to the container's root filesystem, it mitigates the risk of malware or an attacker modifying system files, ensuring the container remains immutable at runtime.

The last one, allowPrivilegeEscalation, prevents processes in the container from gaining additional privileges, such as via setuid binaries. Even if an application attempts to escalate its permissions, this setting blocks it, protecting the container and the host from privilege escalation attacks. All Helm files can be found in Appendix B.

Still regarding configuration it is important to mention one more detail. The Kubernetes objects used here, by themselves, do not ensure an important factor highlighted not only in the GDPR but also in the OWASP Top 10: the protection of data during transit.

Although we enforce HTTPS for external access to the cluster, internal communication within the cluster remains unprotected. Using tools like Wireshark, a network protocol analyzer that allows monitoring and inspecting the content of network traffic, it is possible to capture and examine messages exchanged between pods.

To address this issue, we attempted to use Istio [11]. Istio is a service mesh, a dedicated infrastructure layer that manages service-to-service communication within a cluster. It provides features such as secure communication, traffic management, observability, and policy enforcement.

By deploying Istio, an additional container (sidecar) is injected into each pod in the infrastructure, ensuring that all communications to the pods pass through this sidecar. This setup enforces mutual TLS (mTLS), guaranteeing that internal data is encrypted during transmission.

However, during implementation, we encountered an issue with Istio and CronJobs. When a CronJob completes its execution, its container terminates as expected, but the Istio sidecar within the same pod remains active. This prevents the pod itself from shutting down automatically. Since this solution must support multiple CronJobs, this could result in several pods staying active due to lingering sidecars, without being able to process new requests.

Mechanisms to overcome this problem should be implemented, but since this work is not focused on security, the demonstrator does not use a service mesh to guarantee encrypted communication between pods. In future work, it is highly recommended to adopt a service mesh like Istio to secure internal traffic effectively.

4.2 Application Layer

4.2.1 BITalino Scripts

With the infrastructure provided, the BITalino scripts can now stop being executed locally. For this work, the Python programs developed by the team will need to be added to containers so that they can be deployed through Helm charts.

This procedure was carried out locally using Podman, the open-source container management tool mentioned in Chapter 2. As a proof of concept, the containers were deployed to DockerHub, a public image registry, and no dependency checks were added for the scripts' requirements.

It is also necessary to specify the adaptations required for each of the three categories mentioned in 3.5.3:

- Permanent Scripts: The Python program needs to be converted into a web service, and an application server must be added. As a proof of concept, we used Flask, a lightweight, open-source web framework for Python.

- Scheduled Scripts / On-Demand Scripts: For both cases, the necessary adaptations will be similar, as what really differs is the timing and method of invocation at the infrastructure level, as shown in the previous section. Here, it will be necessary to modify the source code to allow REST calls, retrieve session data, and also deliver output files through the bit-script-management API.

4.2.2 Core Module APIs

As already mentioned, there are only three APIs that provide services to clients external to the Kubernetes cluster. These APIs were developed using Spring Boot, a Java framework that comes with embedded servers such as Tomcat, which simplifies deployment and makes it ideal for developing web applications. Spring Boot enables developers to create production-ready applications with minimal configuration, making it especially useful for microservices and cloud-based architectures. Similar to the BITalino scripts, these APIs were packaged into containers and deployed to DockerHub.

Bit-user-management

There are two essential functionalities in the system: authentication and authorization. As previously explained, this work will focus only on authentication. For this purpose, we used an external provider that offers typical user management operations. The chosen provider was Auth0, which implements security protocols such as OAuth 2.0 and OpenID Connect, and, after authenticating a user, generates session JWTs and provides the necessary operations on them.

Bit-resources

Since the session data is stored through this API, and since this API is also the focus of another study as indicated in 3.1, the potential services that could be implemented here were not explored. However, within the scope of this work, several essential features were implemented.

This service handled data ingestion into the system, and for each file entering the system, an internal ID was generated. This ID was used by the bit-script-management API for hash generation, as explained earlier.

Additionally, this API stored metadata about the BITalino sessions. One of the metadata elements was a flag indicating whether the sessions contained sensitive data that could not be anonymized, one of the intended use cases.

As a proof of concept, the database with which this API communicates was MongoDB, and it was hosted externally to the cluster.

Bit-script-management

This API's main objectives are to interact with the BITalino scripts and forward the output data to bit-storage after creating a hash. Regarding the hash code, in 3.5.3, we suggested this as a preventive method on multiple requests using the same session data from being processed by the same scripts. Several factors had to be considered when choosing the data for hash creation. The selected data cannot be easily editable, as this would lead to inconsistencies. Since it was decided that no changes would be made to the session files once they are in the system, the hash would be secure if it used the contents of these files. However, we have already observed that these files may contain private information, which conflicts with GDPR regulations. It was also determined that all session data would receive a sessionId as soon as it was received by the bit-resources API. So it is suggested that this hash script be implemented with the following elements:

```
userId +
sessionId_File1 + ...  + sessionId_FileN +
scriptName
```

This hash is feasible because none of the indicated fields will be altered in the system. If, under exceptional circumstances, it was necessary to modify, the session files would have to be sent again to the BITalino scripts to generate new outputs, which would then be properly stored in the object storage with the new hash.

The use of this hash also requires one final consideration regarding scriptName. If the source code is altered and redeployed in the infrastructure, it is highly likely that the generated output will be different, breaking the two previously mentioned principles: determinism and idempotence.

For this work, only one Python script was provided, which did not have any changes. However, in a production environment, besides scriptName, the deployment version could also be included in the hash along with the other fields. Each time a new version of the BITalino script is deployed, a different hash would also be generated.

Furthermore, the API was required to support the execution of On-Demand Scripts. This requirement means that the API communicates directly with the kube-apiserver, the exposed API of Kubernetes in the Control Plane, as explained in 2.4.3. This call will launch a Kubernetes Job, which will perform the processing and then terminate. To facilitate this task, Kubernetes provides a Maven dependency that can be added to the Java code, allowing it to make requests to the kube-apiserver.

4.2.3 Storage

The bit-storage API, which connects with object storage, will serve as the bridge between the object storage provider and our cluster. It was also developed using Spring Boot,

and its main functions include the creation of buckets and endpoints for managing the storage file system. Additionally, it will verify the output session data that has already been generated by querying the hash created by bit-script-management.

As the object storage, this work uses MinIO, an open-source, high-performance object storage system designed for large-scale data storage. Like other object storage systems, MinIO is well-suited to store unstructured data, specifically the data produced by the BITalino functions. Similarly to other external elements of our cluster, no measures have been taken to increase the security of MinIO. For our test implementation, MinIO was deployed on a separate virtual machine in Google Compute Engine (GCE), with different IPs from our cluster.

The interaction with MinIO from Java is done by using a Maven dependency developed by MinIO that enables importing classes to facilitate communication with remote storage.

4.2.4 Operations Implemented

The following is a list of endpoints developed for each of the API presented

1. bit-user-management:
    - **POST**: create user
    - **PATCH**: update user
    - **DELETE**: delete user
    - **GET**: generate authentication token
    - **GET**: validate authentication token (authorization not implemented)
2. bit-resources:
    - **POST**: save session
    - **GET**: get session
    - **DELETE**: delete session
3. bit-script-management:
    - **POST**: call script
    - **GET**: get output data
    - **DELETE**: delete output data
    - **POST**: launch script job
    - **GET**: input data (used for async scripts)

- **POST**: save output data (used for async scripts)
- **POST**: anonymize data

4. bit-storage:

- **POST**: create bucket
- **POST**: store output data
- **GET**: get output data
- **DELETE**: delete output data
- **POST**: anonymize data

Chapter 5

Results

In this chapter, we will validate the results of our implementation. We will confirm whether the scripts were indeed executed in an isolated environment and whether their outputs are properly accessible.

5.1 Infrastructure Deployment

The first step to test the use cases requires the deployment of our infrastructure. As mentioned in the previous chapter, for proof of concept, the cluster was created on GCP, which means that only a user with access to a Google account will be able to replicate these steps. However, only a few changes in the Terraform files will be necessary to alter the cloud provider.

After local installation and logging in with gcloud, it is possible to verify that the CLI is connected to GCP. Then, the command terraform init can be executed in our bitplatform/terraform/ folder, which will check for any immediate errors in the .tf files.

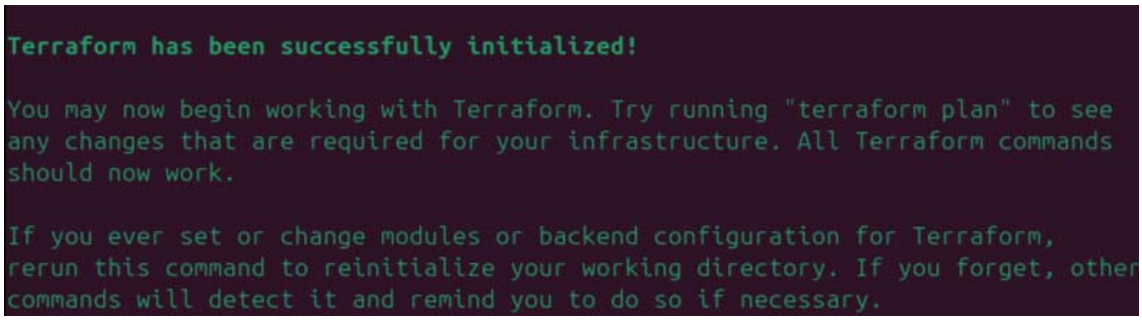

Figure 5.1: Terraform Init

Next, we can apply the Terraform plan. After a few minutes, we can confirm that our Kubernetes cluster has been created with the dependencies mentioned in Chapter 4, as shown by the successful execution of the terraform apply command in Figure 5.2.

```
Apply complete! Resources: 11 added, 0 changed, 0 destroyed.

Outputs:

cluster_name = "bit-cluster"
cluster_region = "europe-southwest1-a"
```

Figure 5.2: Terraform Apply

As the final step to have our infrastructure created, the installation of our custom Helm files will be required. All the Helm files are under the responsibility of a main Helm chart, and to deploy it, simply run the command helm install bit-platform ./common in the /bitplatform/helm-charts folder.

```
user@usercomp:~/Documents/workdir/bitplatform/helm-charts$ kubectl get pods --all-namespaces
NAMESPACE              NAME                                              READY   STATUS    RESTARTS   AGE
bit-platform-private   bit-platform-bit-storage-655b85f675-wdmkp          1/1     Running   0          71s
bit-platform-private   bit-platform-bit-storage-655b85f675-xq4f7          1/1     Running   0          71s
bit-platform-public    bit-platform-bit-gateway-74d78db887-mpmnp          1/1     Running   0          72s
bit-platform-public    bit-platform-bit-resources-5bbffbcfdb-4zcbp        1/1     Running   0          72s
bit-platform-public    bit-platform-bit-resources-5bbffbcfdb-sjnd2        1/1     Running   0          72s
bit-platform-public    bit-platform-bit-script-management-645597fcdc-jsgvx   1/1     Running   0          71s
bit-platform-public    bit-platform-bit-script-management-645597fcdc-m26jm   1/1     Running   0          71s
bit-platform-public    bit-platform-bit-user-management-76c89ffbf4-5k6dz  1/1     Running   0          71s
bit-platform-public    bit-platform-bit-user-management-76c89ffbf4-8dgpt  1/1     Running   0          72s
bit-python-scripts     scriptflaskone-79cc4c5b54-9wzmr                   1/1     Running   0          72s
bit-python-scripts     scriptflasktest-7f455b9c96-vmh5m                  1/1     Running   0          71s
bit-python-scripts     scriptflasktwo-59f6659c6-fzrtz                    1/1     Running   0          72s
```

Figure 5.3: Helm Install

A few minutes later, we can confirm that all the services have been successfully created in our cluster.

5.2 Use cases validation

Next, we will examine in more detail the main services developed at the application level. Several endpoints were created throughout this work, but we will only present input and output examples for those endpoints consumed by users, in order to validate the use cases. The bit-storage API will not be detailed here, as its purpose is to facilitate interaction with our object storage.

5.2.1 User operations

The following are the operations allowed by the bit-user-management API, which is responsible for user management.

Table 5.1: Create/Update user

Kind	Content

Request	{
	"userId": "userTest",
	"password": "password123",
	"email": "user@test.com"
	}

Response Success	200 OK
Response Error	400 Bad Request | 401 Unauthorized

This is the first endpoint in the user flow, after which tokens can be generated. Since both the calls to register session data and the calls for its processing are routed for token validation, this is where the user must register.

Table 5.2: Delete user

Kind	Content

Request	{
	"userId": "userTest"
	}

Response Success	200 OK
Response Error	400 Bad Request | 401 Unauthorized

This is only a service for deleting the user from Auth0, and session or output data will not be removed here. The corresponding services will also need to be invoked.

Table 5.3: Generate token

Kind	Content

Request	{
	"userId" : "userTest",
	"password" : "password123"
	}

Response Success	eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtvL...
Response Error	400 Bad Request | 401 Unauthorized

Service for token generation. This service is not protected and can be accessed without any authentication. All calls to other services require a JWT generated here.

Table 5.4: Validate token

Kind	Content
Request Authorization	`Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtvL...`
Response Success	200 OK
Response Error	400 Bad Request | 401 Unauthorized

The bit-gateway will forward to this endpoint all requests arriving at our cluster, with the exception of two: user creation and token generation.

5.2.2 Session operations

Table 5.5: Insert session

Kind	Content
Request	`{ "userId" : "userTest", "gender" : "M", "ageGroup" : "Teen", "weight" : "80", "height" : "170", "poolSize" : "30", "canAnonymize" : true, "sessionData" : "[0.15,-3.24,-0.42,-0.15...]", }`
Response Success	`{ "sessionId": "68d65b651f5b9915f82358ee" }`
Response Error	400 Bad Request | 401 Unauthorized

This will be the sole entry point for session data into the system. As mentioned, many useful features could be integrated into this service, however, it was developed in the context of another work. Is relevant here is to verify that a sessionId is always generated when data is inserted.

This service will also serve as entry point for the canAnonymize field. This field will act as a flag to indicate whether the inserted session contains data considered private according to GDPR criteria.

Both sessionId and canAnonymize are important for the services presented next.

Table 5.6: Get sessions

Kind	Content

Request	{
	"sessionsIds": ["68d65b651f5b9915f82358ee"]
	}

Response Success	{
	"inputDataList": [
	{
	"sessionId": "68d65b651f5b9915f82358ee",
	"fields": {
	"time": [1364.00, 1374.00, ...],
	"pitch" : [0.36, 0.76, ...],
	"ax" : [-0.42, -0.41, ...]
	}
	}
	]
	}

Response Error	400 Bad Request | 401 Unauthorized

In Table 5.6, we can observe in the response that data has been processed in relation to the input from the previous service. This requirement was raised with the aim of facilitating the execution of scripts and providing greater detail on the fields being handled. We can see that names were assigned to the fields here, since the input structure is known.

Table 5.7: Delete session

Kind	Content

Request	{
	"sessionId" : "68d65b651f5b9915f82358ee"
	}

Response Success	200 OK
Response Error	400 Bad Request | 401 Unauthorized

The user is given the option to delete their session data regardless of whether it is private or not. No action is taken here with respect to output data. A user may wish to delete their sessions, but this will not affect outputs that have already been generated. Specific services for deleting output data will be presented next. This service can also be used to delete session data belonging to other users, provided that the previously established permissions are respected.

5.2.3 Script operations

Table 5.8: Execute script

Kind	Content

Request	{
	"userId": "userTest",
	"scriptName": "scriptflaskone",
	"inputDataList": [
	{
	"sessionId": "68d65b651f5b9915f82358ee",
	"fields": {
	"time": [1364.00, 1374.00, ...],
	"pitch" : [0.36, 0.76, ...],
	"ax" : [-0.42, -0.41, ...]
	}
	}
	]
	}

Response Success	{
	"outputId": "cNC4riSSf38jYMurGZA3DzYN_lxg6AyQTaht6ceGIZg"
	"outputList": [
	{
	"objectName": "file0.png",
	"objectContent": "iVBORw0KGgoAAAANSUhEUgAAAoAA..."
	}
	]
	}

Response Error	400 Bad Request | 401 Unauthorized

Here, the user can pass session data to the script of their choice. The name of the script is treated as its identifier. For this service to execute successfully, the script name must correspond to a pod deployed in the bit-python-scripts namespace and must also be listed in the `networkpolicy-flask.yaml` configuration file. Otherwise, the call will return an error.

If this is the first execution, a new record must be verified in MinIO. If it is a repeated processing, the data will be loaded from MinIO using the hash, and the script will not be executed.

To validate the result of this service, we are using the sessionId resulting from 5.5. The script created a file with the name file0.png that is now stored in MinIO, more specifically in

`userdata/`
`userTest/`
`cNC4riSSf38jYMurGZA3DzYN_lxg6AyQTaht6ceGIZg/`
`file0.png`

Table 5.9: Get output script

Kind	Content

Request	{
	"userId": "userTest",
	"outputId": "cNC4riSSf38jYMurGZA3DzYN_lxg6AyQTaht6ceGIZg"
	}

Response Success	{
	"outputId": "cNC4riSSf38jYMurGZA3DzYN_lxg6AyQTaht6ceGIZg"
	"outputList": [
	{
	"objectName": "file0.png",
	"objectContent": "iVBORw0KGgoAAAANSUhEUgAAAoAA..."
	}
	]
	}

Response Error	400 Bad Request | 401 Unauthorized

Only a query to MinIO is performed. No scripts are involved.

Table 5.10: Delete output script

Kind	Content

Request	{
	"userId": "userTest",
	"outputId": "cNC4riSSf38jYMurGZA3DzYN_lxg6AyQTaht6ceGIZg"
	}

Response Success	200 OK
Response Error	400 Bad Request | 401 Unauthorized

Service for deleting data by receiving the outputId (hash). Since each script can generate multiple outputs, all files and their corresponding directories in MinIO are deleted.

This service applies only to user data, that is, in the userdata bucket in MinIO. Users are not given the option to delete statistical data, as multiple automatic executions may depend on this data. Statistical data in our system is considered private under the GDPR and therefore cannot be used without prior consent.

Table 5.11: Anonymize data

Kind	Content

Request	{
	"userId": "userTest"
	}

Response Success	200 OK

continued on next page

Table 5.11 – *continued from previous page*

Kind	Content
Response Error	400 Bad Request | 401 Unauthorized

The purpose of this service is to address use case UC5 in Figure 3.1 and as explained in 3.7. When scripts generate output data that is stored in MinIO, the canAnonymize field is also saved alongside the generated objects. In this service, all data for the userId provided in the request body is checked, and any data that can be anonymized is moved from the user's bucket to the statistical bucket.

For this work, we implemented only the option to anonymize all data based on the userId but more versatility could be considered in the future, such as anonymizing data partially.

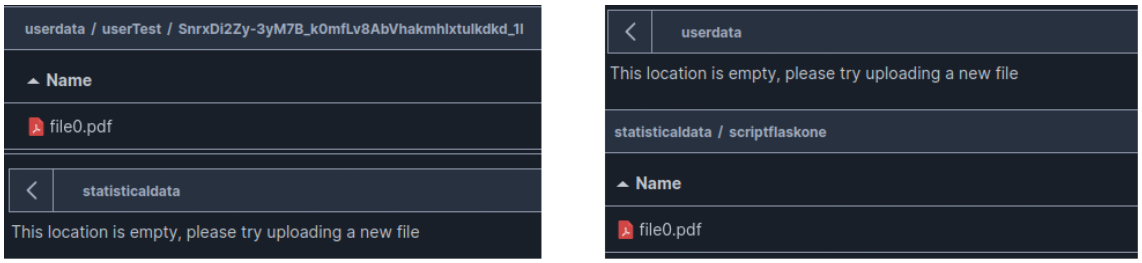

Figure 5.4: Before anonymization | After anonymization

In Figure 5.4, we can see on the left that the statisticaldata directory is empty and that userTest has a file file0.pdf inside a folder with a previously generated hash. In this case, file0.pdf has two metadata fields:

```
"scriptName": "scriptflaskone"
"canAnonymize": "true"
```

The boolean value is stored as a string because MinIO does not support boolean variables as object metadata. It represents the flag that indicates whether or not data can be anonymized.

When the service is executed, the transfer described earlier occurs. The object file0.pdf is moved to the statistical bucket and associated with the script where it was generated, carrying this information from its metadata. The folder in userdata is then deleted and the file is no longer associated to that user.

Hash validation

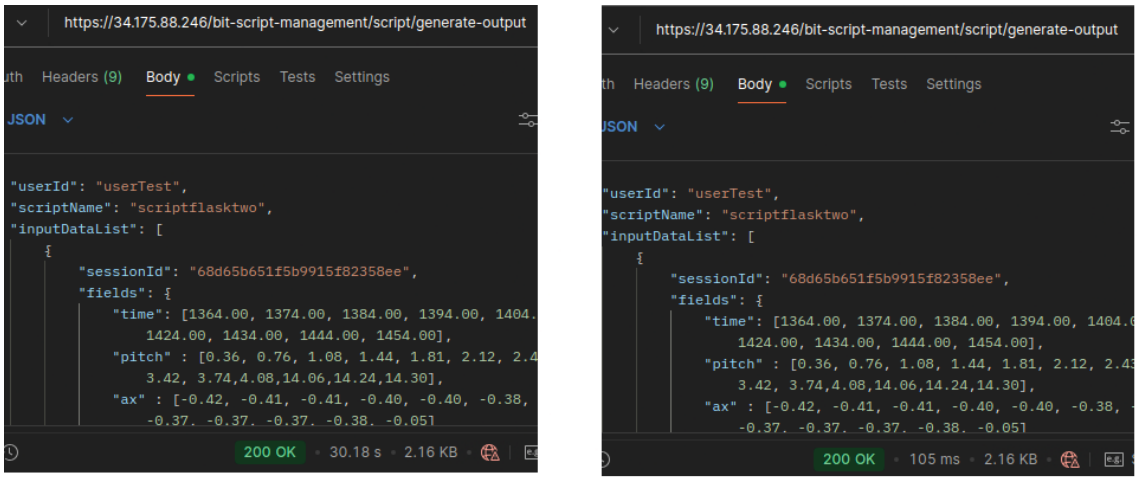

Figure 5.5: Hash comparison

The use of the hash was intended to satisfy the functional requirement FR3: Avoid repeated processing. We have already explained the hash, but it is necessary to confirm that it is actually being used.

In Figure 5.5, we can observe two consecutive and identical requests to execute the script named `scriptflasktwo`. This script, in addition to generating a PDF, contains a sleep function that makes the program wait for 30 seconds. The image on the left corresponds to the first call, when the hash did not exist, whereas the image on the right shows the execution after a previous generation request had already been made.

Using a tool like Postman, we can easily verify the time each request took. At the bottom of both requests, we can see that the first took considerably longer (30.18 s) than the second request (105 ms), since the latter did not execute the script again but retrieved the data directly from MinIO. This confirms that the functionality was correctly implemented.

Asynchronous scripts

From the analysis of the results so far, it has already been possible to verify the implementation of one of the three types of scripts: the always-available ones. However, the results for the Scheduled and On-Demand scripts still need to be examined. As discussed in the previous chapter, to ensure their implementation we used the Kubernetes objects CronJob and Job, respectively.

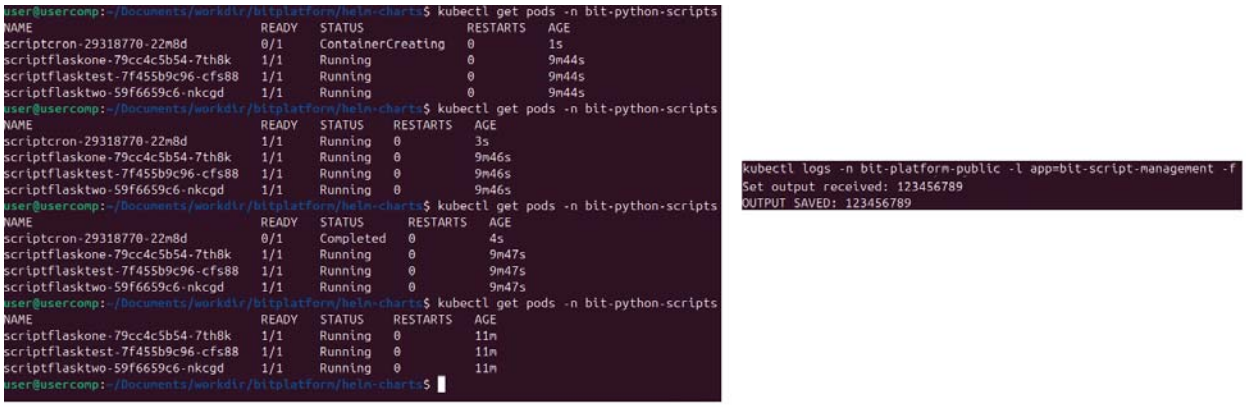

Figure 5.6: CronJob Execution

To validate the results of the CronJob, we created an example script called scriptcron that is scheduled to run every five minutes, with its configurations available in the file cronjob-scriptcron.yaml.

In the Figure 5.6 on the left, we can see that the corresponding pod, called scriptcron, was successfully launched by Kubernetes. The pod passed through all states successfully and was eventually removed. On the right, we can observe a log from the bit-script-management pod, which received an example input asynchronously from the processing of scriptcron and confirming that the scripts can call this API.

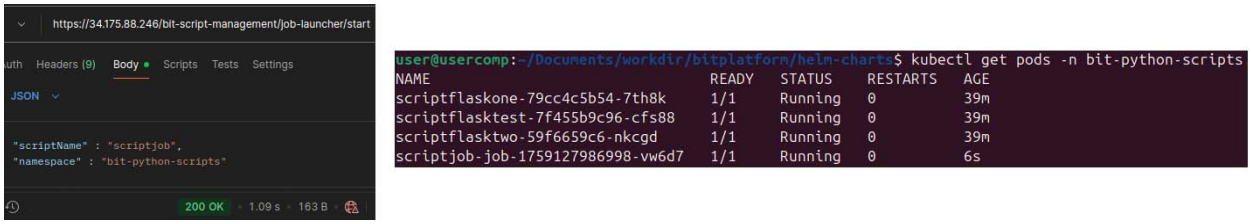

Figure 5.7: Job Execution

Regarding Kubernetes Jobs, we need to make a call to bit-script-management. This API contains the service that communicates directly with the Kubernetes API through a ServiceAccount created for this purpose, and which is able to launch our On-Demand scripts.

In Figure 5.7, we can observe a successful execution of this call.

Contained environment

Based on the validations conducted so far, it was confirmed that the bit-script-management API is able to communicate with the namespace in which the scripts are deployed, and that the scripts themselves are also capable of invoking this API. The objective of maintaining the scripts in as isolated an environment as possible remains, making it necessary to determine whether these scripts are able to perform calls outside the cluster.

To verify this, a script named scriptflasktest was created, and permissions were explicitly configured to allow its execution through bit-script-management. This script attempts to invoke the Auth0 API directly in order to obtain a user token. As the scope of this work did not include detailed configurations of external components, this API remains unprotected and can therefore be accessed by any web request.

In order to test the permissions of our cluster, we executed this script both locally and within the cluster. As expected, in Figure 5.8 we can observe that the local execution successfully obtained a token, while the execution performed inside the cluster resulted in an error.

It is also important to note that even if an unforeseen pod were to appear within the cluster, it would not be able to communicate with the others. Due to Kubernetes NetworkPolicies, all communications are denied by default.

Figure 5.8: Blocking outside call

5.3 Discussion

We took a closer look at the results of the implementation discussed in the previous chapter.

Our process begins immediately by providing tools capable of creating the infrastructure from scratch. Terraform allowed us to allocate the cloud resources necessary for our applications to function properly. Later, our custom helm charts were applied, and the necessary applications were deployed. To complement the discussion on these results, an analysis of the implementation is now proposed in light of the use cases and requirements made in chapter 3.

With the creation of the exposed services bit-user-management, bit-resources and bit-script-management, and the endpoints discussed in this chapter, it was possible to provide means for BITalino system users to authenticate and authorize themselves (UC1), access the data collected in the sessions (UC2), and execute their scripts (UC3) in a controlled environment.

Since it was not possible to define solid requirements for future scripts to be developed, we used an object storage solution that facilitates handling unstructured data to

persist the results produced by the scripts and make them accessible to users (UC4).

Mechanisms were implemented to anonymize user data within MinIO for cases where users wish to remain registered in the system but without their data being used for processing (UC5). Additionally, the possibility to completely delete these data was also provided (UC6).

Below is a list specifying what was implemented for each Functional requirement and Non Functional requirement:

- **FR1 - Different types of Session Files:** Session data from the BITalino device can have variable columns depending on the sensor or context. The system supports this via a consistent session ingestion service (bit-gateway, bit-user-management, bit-resources) and object storage, which can store session inputs while keeping the sessionID and `canAnonymize` metadata for traceability.

- **FR2 - Different types of Output Data:** Output data from scripts is managed using MinIO ensures that outputs in any format are persistently stored and accessible to users without format constraints.

- **FR3 - Avoid repeated processing:** Hashing of session input is used to detect repeated requests.

- **FR4 - Different sizes of output data:** The system accommodates variable output sizes by using scalable object storage (MinIO) and containerized scripts that can handle both small and large files.

- **FR5 - Script execution can be synchronous or asynchronous:** Scripts can be executed either synchronously or asynchronously. Both executions are managed via the `bit-script-management` service.

- **FR6 - Script deployment on demand:** On-demand scripts are deployed as Kubernetes Jobs and created by calling directly the kubernetes API.

- **FR7 - Keep statistical data:** The system anonymizes user data stored in MinIO based on the `canAnonymize` flag. Users can delete their session data (UC6) while preserving anonymized statistics for aggregate analysis. This ensures compliance with GDPR principles, while statistical data remains available for processing without identifying individual users.

- **NFR1 - Availability:** Kubernetes Deployments with multiple replicas, bit-gateway API and Ingress for external access, CronJobs and Jobs for scheduled/on-demand scripts.

- **NFR2 - Scalability:** Horizontal scaling via `replicaCount` in Helm values; independent scaling of microservices; Kubernetes node pool autoscaling; scripts containerized and independently deployable.
- **NFR3 - Portability:** Containerized microservices with Helm charts and Terraform provisioning; works on GKE or any Kubernetes cluster.
- **NFR4 - Performance:** Hashing mechanism to prevent repeated processing; efficient intra-cluster communication using ClusterIP; minimal sidecars; lightweight `securityContext`.
- **NFR5 - Traceability:** SessionId, userId, scriptName stored with outputs in MinIO; ConfigMaps for service URLs; logging and auditability supported.
- **NFR6 - Maintainability and Operability:** Microservices designed for independent deployment; infrastructure as code with Helm and Terraform.
- **NFR7 - Fault Tolerance:** Kubernetes self-healing Deployments; CronJob restart policies; default deny all NetworkPolicies; MinIO for persistent data; ServiceAccount.
- **NFR8 - GDPR Compliance:** Data anonymization (`canAnonymize`); selective deletion of user data; Secrets for sensitive data; NetworkPolicies and namespaces isolate users; future possibility of mTLS via Istio.
- **NFR9 - OWASP Top 10:** NetworkPolicies restrict pod communication; multiple configuration in kubernetes added; Ingress HTTPS enforced; JWT tokens for API authentication; isolated namespaces for scripts.

Chapter 6

Conclusions

The BITalino device was developed to address a real need: simplifying and making the measurement of biosignals more economically accessible, something with numerous applications, as we have seen. However, this also raised several critical issues regarding how users' data were being handled. The lack of a dedicated software infrastructure for executing scripts and storing their results meant that it was not possible to have visibility over the data flow or the type of processing being applied. This work was designed to contribute to solving these problems.

We examined in detail why it is relevant to follow the practices set forth in the GDPR and how these measures influenced the design and implementation of the proposed solution. Security and privacy were also essential factors. Cybersecurity is a broad topic that affects many areas. While the primary goal of this work was not to focus exclusively on security, addressing it was unavoidable in order to establish a minimum level of trust in the solution. For this reason, we followed the OWASP Top 10 guidelines as a baseline for implementing protection mechanisms.

Since this project was built from the foundation, it was necessary to make decisions both at the application level and at the infrastructure level. The state-of-the-art analysis guided the design and implementation of the applications required to support the identified use cases and requirements. By using Terraform, we leveraged Infrastructure as Code, which significantly simplified deployment in any cloud provider or even on-premise environments.

Although the system is inherently unpredictable, a direct consequence of having to support different types of scripts, we implemented mechanisms to avoid repeated processing. By relying on a hashing mechanism, identical script executions with the same input did not trigger unnecessary computation. This approach not only improved system performance but also reduced resource consumption across the Kubernetes cluster.

The system supported multiple output data formats generated by containerized scripts, including JSON, PDFs, images, and other unstructured data formats. It was also able to process heterogeneous session data from BITalino devices, while preserving critical

metadata such as session identifiers and anonymization flags.

The object storage solution implemented with MinIO proved effective for storing these outputs, enabling persistent storage and facilitating user retrieval without constraints on data type or size.

6.1 Future work

6.1.1 Deployment (CI/CD) pipelines

In this work, it was assumed as a starting point that the scripts would already be available in an image registry such as Docker Hub. The lack of validation of potential security vulnerabilities in these scripts could compromise the infrastructure developed here. Mechanisms integrated into CI/CD pipelines, such as automated vulnerability scanning and image validation, would strengthen security.

6.1.2 Istio

As we discussed in 4.1.2, additional configurations, several Kubernetes objects were implemented. However, encrypted communication within the cluster was not guaranteed. Ensuring internal encryption would contribute to increasing overall security by protecting data exchanged between pods against interception or tampering, even if an attacker were to gain access to the internal network.

A possible solution for future work is the adoption of Istio, a service mesh framework that manages communication between microservices. Istio works by injecting sidecar proxies into each pod, enforcing mutual TLS (mTLS) for all service-to-service communication. This approach would guarantee encrypted traffic within the cluster and provide additional features such as fine-grained traffic management or fault injection. This implementation must not affect the usage o Kubernetes CronJobs.

6.1.3 Event-Driven

In future work, if the requirements so indicate, it may be justified to adopt an event-driven approach in certain parts of the architecture. In this context, KEDA (Kubernetes Event-Driven Autoscaling) could play an important role by enabling workloads to scale automatically based on external event sources. This would allow scripts to be triggered by events rather than Jobs, making execution more efficient.

Bibliography

[1] Regulation EU 2016/679 General Data Protection Regulation. https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng, 2016. Official Journal of the European Union, L119, 1–88.

[2] Inês Pereira Afonso. Plataforma para testes de segurança em aplicações web. Trabalho de projeto de mestrado em segurança informática, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, 2020. Orientado por Prof. Doutor Mário João Barata Calha.

[3] Tiago Almeida. Processamento de biosinais de desporto numa arquitetura em nuvem. Trabalho de projeto de mestrado em informática, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, 2023. Orientado por Prof. Doutor Mário João Barata Calha.

[4] Ana Priscila Alves, Hugo Silva, André Lourenço, and Ana Fred. Bittalino: A biosignal acquisition system based on the arduino. In *Proceedings of the International Conference on Biomedical Electronics and Devices (BIODEVICES 2013)*, Lisbon, Portugal, 2013.

[5] Beck, Kent et al. Manifesto for agile software development. https://agilemanifesto.org/, 2001.

[6] Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, July 2015. Accessed: 2025-09-30.

[7] Google LLC. Google cloud platform documentation. https://cloud.google.com/docs, 2025.

[8] HashiCorp. Terraform documentation. https://www.terraform.io/docs/, 2025.

[9] Helm. Helm documentation and charts. https://helm.sh/docs/, 2025.

[10] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2024.

[11] Istio Authors. Istio: An open platform to connect, manage, and secure microservices. https://istio.io/.

[12] Surbhi Kanthed. Docker vs. podman: Architecture differences and their impact on modern workflows. *International Journal of Multidisciplinary Research*, 2023.

[13] Kubernetes. Kubernetes documentation. https://kubernetes.io/docs/, 2025.

[14] MinIO, Inc. Minio documentation. https://docs.min.io/, 2025.

[15] OWASP Foundation. Owasp top 10 - 2021. https://owasp.org/Top10/, 2021.

[16] César Páris, Jorge Barbosa, Emanuel Ferreira, and Anabela Gomes. Bitalino use and applications for health, education, home automation and industry. In *Proceedings of the Engineering Institute of Polytechnic of Coimbra Conference*, Coimbra, Portugal, 2017. Engineering Institute of Polytechnic of Coimbra; Centre for Informatics and Systems, University of Coimbra.

[17] André G. Pinto, Gil Dias, Virginie Felizardo, Nuno Pombo, Hugo Silva, and Paulo Fazendeiro. Electrocardiography, electromyography, and accelerometry signals collected with BITalino while swimming: Device assembly and preliminary results. In *2016 IEEE 12th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2016.

[18] Red Hat, Inc. Podman documentation. https://podman.io/documentation.html, 2025.

[19] Sebastião Silva. A framework for supporting privacy in the computation of biosignals. Trabalho de projeto de mestrado em informática, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, 2021. Orientado por Prof. Doutor Mário João Barata Calha.

[20] Statista. Statistics and market data portal. https://www.statista.com/, 2025. Accessed: September 30, 2025.

[21] Neal Thoms. A definition of devops for the masses. https://devops.com/definition-devops-masses/, January 2018. Accessed: 2025-09-30.

[22] B. Wolford. What is GDPR, the EU's new data protection law? [https://gdpr.eu/what-is-gdpr/.

Appendix A

Terraform files

Here is a list of the configuration files created for deploying the infrastructure and its dependencies.

main.tf

```
data "google_client_config" "default" {}

resource "google_project_service" "gcp_services" {
  for_each = toset([
    "container.googleapis.com",
    "compute.googleapis.com",
  ])
  project = var.project_id
  service = each.key
  disable_on_destroy = false
}

resource "google_container_cluster" "primary" {
  name = "bit-cluster"
  location = var.region
  project = var.project_id

  remove_default_node_pool = true
  initial_node_count = 1

  network_policy {
    enabled = true
    provider = "CALICO"
  }

  depends_on = [google_project_service.gcp_services]
}

resource "google_container_node_pool" "primary_nodes" {
  name = "bit-node-pool"
  location = var.region
```

```
  cluster = google_container_cluster.primary.name
  project = var.project_id
  node_count = 1

  node_config {
    machine_type = "e2-standard-4"
    image_type = "COS_CONTAINERD"

    oauth_scopes = [
      "https://www.googleapis.com/auth/logging.write",
      "https://www.googleapis.com/auth/monitoring",
      "https://www.googleapis.com/auth/devstorage.read_only",
    ]
  }
}

resource "kubernetes_namespace" "ingress_nginx" {
  metadata { name = "ingress-nginx" }
}

resource "kubernetes_namespace" "cert_manager" {
  metadata { name = "cert-manager" }
}

resource "helm_release" "ingress_nginx" {
  name = "ingress-nginx"
  namespace = kubernetes_namespace.ingress_nginx.metadata[0].name
  repository = "https://kubernetes.github.io/ingress-nginx"
  chart = "ingress-nginx"
  version = "4.10.0"

  set {
    name = "controller.service.type"
    value = "LoadBalancer"
  }
}

resource "helm_release" "cert_manager" {
  name = "cert-manager"
  namespace = kubernetes_namespace.cert_manager.metadata[0].name
  repository = "https://charts.jetstack.io"
  chart = "cert-manager"
  version = "v1.13.1"
  set {
    name = "installCRDs"
    value = "true"
  }
}
```

```
resource "helm_release" "calico" {
  name = "calico"
  namespace = "kube-system"
  repository = "https://projectcalico.docs.tigera.io/charts"
  chart = "tigera-operator"
  version = "v3.27.3"
  depends_on = [google_container_cluster.primary]
}
```

outputs.tf

```
output "cluster_name" {
  value = google_container_cluster.primary.name
}

output "cluster_region" {
  value = var.region
}
```

providers.tf

```
terraform {
  required_providers {
    google = { source="hashicorp/google" version="~> 4.50.0" }
    kubernetes = { source="hashicorp/kubernetes" version="~>
       2.20.0" }
    helm = { source="hashicorp/helm" version="~> 2.13.2" }
  }
}

provider "google" {
  project = var.project_id
  region = var.region
}

provider "kubernetes" {
  host = google_container_cluster.primary.endpoint
  token = data.google_client_config.default.access_token
  cluster_ca_certificate = base64decode(google_container_cluster.
     primary.master_auth[0].cluster_ca_certificate)
}

provider "helm" {
  kubernetes {
    host = google_container_cluster.primary.endpoint
    token = data.google_client_config.default.access_token
    cluster_ca_certificate = base64decode(
       google_container_cluster.primary.master_auth[0].
       cluster_ca_certificate)
  }
}
```

terraform.tfvars

```
project_id = "kinetic-catfish-443521-p4"
region = "europe-southwest1-a"
```

variables.tf

```
variable "project_id" {
  description = "GCP Project ID"
  type = string
}

variable "region" {
  description = "GCP region"
  type = string
  default = "europe-southwest1-a"
}
```

Appendix B

Helm files

Here is a list of the Helm Charts used for deploying the applications and their associated Kubernetes resources.

/workdir/bitplatform/helm-charts/common/values.yaml

```
global:
  replicaCount: 2
  image:
    repository: docker.io/DOCKER_USER
    tag: latest
    pullPolicy: Always
  namespaces:
    public: bit-platform-public
    private: bit-platform-private
    scripts: bit-python-scripts

networkPolicies:
  enabled: true
  rules:
    - name: allow-script-mgmt-to-storage
      source: bit-script-management
      target: bit-storage
      port: 8080

config:
  environment: "production"
```

/workdir/bitplatform/helm-charts/common/Chart.yaml

```
apiVersion: v2
name: bit-platform
description: Commun chart
type: application
version: 0.1.0

dependencies:
```

```
  - name: bit-user-management
    version: 0.1.0
    repository: "file://../bit-user-management"
  - name: bit-resources
    version: 0.1.0
    repository: "file://../bit-resources"
  - name: bit-script-management
    version: 0.1.0
    repository: "file://../bit-script-management"
  - name: bit-storage
    version: 0.1.0
    repository: "file://../bit-storage"
  - name: bit-python-scripts
    version: 0.1.0
    repository: "file://../bit-python-scripts"
  - name: ingress
    version: 0.1.0
    repository: "file://../ingress"
  - name: bit-gateway
    version: 0.1.0
    repository: "file://../bit-gateway"
```

/workdir/bitplatform/helm-charts/common/templates/namespaces.yaml

```
{{- if .Values.global.namespaces }}
---
apiVersion: v1
kind: Namespace
metadata:
  name: {{ .Values.global.namespaces.public }}
  labels:
    name: {{ .Values.global.namespaces.public }}
    environment: production
---
apiVersion: v1
kind: Namespace
metadata:
  name: {{ .Values.global.namespaces.private }}
  labels:
    name: {{ .Values.global.namespaces.private }}
    environment: production
    network-policy: isolated
{{- end }}
```

/workdir/bitplatform/helm-charts/common/templates/network-policies.yaml

```
{{- if .Values.networkPolicies.enabled }}
# Allow only bit-script-management to access bit-storage
{{- range .Values.networkPolicies.rules }}
---
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: {{ .name }}
  namespace: {{ $.Values.global.namespaces.private }}
spec:
  podSelector:
    matchLabels:
      app: {{ .target }}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: {{ $.Values.global.
              namespaces.public }}
      podSelector:
        matchLabels:
          app: {{ .source }}
    ports:
    - protocol: TCP
      port: {{ .port }}

{{- end }}

# Default deny all incoming traffic to private namespace
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-to-private
  namespace: {{ .Values.global.namespaces.private }}
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress: []
{{- end }}
```

/workdir/bitplatform/helm-charts/common/templates/configmap.yaml

```
{{- if .Values.config }}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-app-config
  namespace: {{ .Values.global.namespaces.public }}
  labels:
```

```
    app: bit-platform
    chart: {{ .Chart.Name }}-{{ .Chart.Version }}
    release: {{ .Release.Name }}
data:
  SPRING_PROFILES_ACTIVE: {{ .Values.config.environment | quote
     }}
  BIT_STORAGE_URL: "http://{{ .Release.Name }}-bit-storage.{{ .
     Values.global.namespaces.private }}.svc.cluster.local:8080"
  SCRIPT_FLASK_ONE: "http://scriptflaskone.{{ .Values.global.
     namespaces.scripts }}.svc.cluster.local:5000/process-session
     "
  SCRIPT_FLASK_TWO: "http://scriptflasktwo.{{ .Values.global.
     namespaces.scripts }}.svc.cluster.local:5000/generate_pdf"
  SCRIPT_FLASK_TEST: "http://scriptflasktest.{{ .Values.global.
     namespaces.scripts }}.svc.cluster.local:5000/trigger-auth0"
{{- end }}
```

/workdir/bitplatform/helm-charts/certs/clusterissuer.yaml

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned-cluster-issuer
spec:
  selfSigned: {}
```

//bit-user-management

/workdir/bitplatform/helm-charts/bit-user-management/Chart.yaml

```
apiVersion: v2
name: bit-user-management
description: Helm chart for bit-user-management
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-user-management/templates/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-bit-user-management
  namespace: {{ .Values.global.namespaces.public }}
spec:
  type: ClusterIP
  selector:
    app: bit-user-management
  ports:
    - port: 8080
      targetPort: 8080
```

/workdir/bitplatform/helm-charts/bit-user-management/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-bit-user-management
  namespace: {{ .Values.global.namespaces.public }}
spec:
  replicas: {{ .Values.global.replicaCount }}
  selector:
    matchLabels:
      app: bit-user-management
  template:
    metadata:
      labels:
        app: bit-user-management
    spec:
      containers:
        - name: bit-user-management
          image: "{{ .Values.global.image.repository }}/bit-user-
             management:{{ .Values.global.image.tag }}"
          imagePullPolicy: {{ .Values.global.image.pullPolicy }}
          ports:
            - containerPort: 8080
```

//bit-resources

/workdir/bitplatform/helm-charts/bit-resources/Chart.yaml

```
apiVersion: v2
name: bit-resources
description: Helm chart for bit-resources
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-resources/templates/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-bit-resources
  namespace: {{ .Values.global.namespaces.public }}
spec:
  type: ClusterIP
  selector:
    app: bit-resources
  ports:
    - port: 8080
      targetPort: 8080
```

/workdir/bitplatform/helm-charts/bit-resources/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-bit-resources
  namespace: {{ .Values.global.namespaces.public }}
spec:
  replicas: {{ .Values.global.replicaCount }}
  selector:
    matchLabels:
      app: bit-resources
  template:
    metadata:
      labels:
        app: bit-resources
    spec:
      containers:
        - name: bit-resources
          image: "{{ .Values.global.image.repository }}/bit-
             resources:{{ .Values.global.image.tag }}"
          imagePullPolicy: {{ .Values.global.image.pullPolicy }}
          ports:
            - containerPort: 8080
```

//bit-script-management

/workdir/bitplatform/helm-charts/bit-script-management/Chart.yaml

```
apiVersion: v2
name: bit-script-management
description: Helm chart for bit-script-management
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-script-management/templates/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-bit-script-management
  namespace: {{ .Values.global.namespaces.public }}
spec:
  type: ClusterIP
  selector:
    app: bit-script-management
  ports:
    - port: 8080
      targetPort: 8080
```

/workdir/bitplatform/helm-charts/bit-script-management/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-bit-script-management
  namespace: {{ .Values.global.namespaces.public }}
spec:
  replicas: {{ .Values.global.replicaCount }}
  selector:
    matchLabels:
      app: bit-script-management
  template:
    metadata:
      labels:
        app: bit-script-management
    spec:
      serviceAccountName: job-launcher-sa
      containers:
        - name: bit-script-management
          image: "{{ .Values.global.image.repository }}/bit-script-
             management:{{ .Values.global.image.tag }}"
          imagePullPolicy: {{ .Values.global.image.pullPolicy }}
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: {{ .Release.Name }}-app-config
```

/workdir/bitplatform/helm-charts/bit-script-management/templates/serviceaccount-job-launcher.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: job-launcher-sa
  namespace: {{ .Values.global.namespaces.public }}
```

/workdir/bitplatform/helm-charts/bit-script-management/templates/role-job-launcher.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: job-launcher-role
  namespace: {{ $.Values.global.namespaces.scripts }}
rules:
  - apiGroups: ["batch"]
    resources: ["jobs"]
    verbs: ["create", "get", "list", "watch"]
```

/workdir/bitplatform/helm-charts/bit-script-management/templates/rolebinding-job-launcher.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: job-launcher-binding
  namespace: {{ $.Values.global.namespaces.scripts }}
subjects:
  - kind: ServiceAccount
    name: job-launcher-sa
    namespace: {{ $.Values.global.namespaces.public }}
roleRef:
  kind: Role
  name: job-launcher-role
  apiGroup: rbac.authorization.k8s.io
```

//bit-storage

/workdir/bitplatform/helm-charts/bit-storage/Chart.yaml

```
apiVersion: v2
name: bit-storage
description: Helm chart for bit-storage
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-storage/values.yaml

```
minio:
  accessKey: "minioadmin"
  secretKey: "minioadmin"
```

/workdir/bitplatform/helm-charts/bit-storage/templates/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-bit-storage
  namespace: {{ .Values.global.namespaces.private }}
  labels:
    app: bit-storage
spec:
  type: ClusterIP
  selector:
    app: bit-storage
  ports:
    - port: 8080
      targetPort: 8080
```

/workdir/bitplatform/helm-charts/bit-storage/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-bit-storage
  namespace: {{ .Values.global.namespaces.private }}
spec:
  replicas: {{ .Values.global.replicaCount }}
  selector:
    matchLabels:
      app: bit-storage
  template:
    metadata:
      labels:
        app: bit-storage
    spec:
      containers:
        - name: bit-storage
          image: "{{ .Values.global.image.repository }}/bit-storage
             :{{ .Values.global.image.tag }}"
          imagePullPolicy: {{ .Values.global.image.pullPolicy }}
          ports:
            - containerPort: 8080
          env:
            - name: MINIO_ACCESS_KEY
              valueFrom:
                secretKeyRef:
                  name: minio-credentials
                  key: MINIO_ACCESS_KEY
            - name: MINIO_SECRET_KEY
              valueFrom:
                secretKeyRef:
                  name: minio-credentials
                  key: MINIO_SECRET_KEY
```

/workdir/bitplatform/helm-charts/bit-storage/templates/minio-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: minio-credentials
  namespace: {{ .Values.global.namespaces.private }}
type: Opaque
data:
  MINIO_ACCESS_KEY: {{ .Values.minio.accessKey | b64enc | quote
     }}
  MINIO_SECRET_KEY: {{ .Values.minio.secretKey | b64enc | quote
     }}
```

//SCRIPTS

/workdir/bitplatform/helm-charts/bit-python-scripts/Chart.yaml

```
apiVersion: v2
name: bit-python-scripts
description: Helm chart for scripts
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-python-scripts/values.yaml

```
global:
  namespaces:
    scripts: bit-python-scripts # private namespace just for
       python scripts
  image:
    repository: docker.io/pedropereirammp
    tag: latest
    pullPolicy: Always

replicaCount: 1

scripts:
  - name: scriptflaskone
    port: 5000
  - name: scriptflasktwo
    port: 5000
  - name: scriptflasktest
    port: 5000
  - name: scriptcron
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: {{ .Values.global.namespaces.scripts }}
  labels:
    environment: production
    network-policy: isolated
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/networkpolicy-flask.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: flask-scripts-policy
  namespace: {{ .Values.global.namespaces.scripts }}
```

```
spec:
  podSelector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - scriptflaskone
          - scriptflasktwo
          - scriptflasktest
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: {{ .Values.global.
                  namespaces.public }}
          podSelector:
            matchLabels:
              app: bit-script-management
      ports:
        - protocol: TCP
          port: 5000
  egress: []
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/networkpolicy-cron.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: cron-scripts-policy
  namespace: {{ .Values.global.namespaces.scripts }}
spec:
  podSelector:
    matchLabels:
      app: scriptcron
  policyTypes:
    - Egress
  egress:
    # Allow egress to bit-script-management
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: {{ .Values.global.
                  namespaces.public }}
          podSelector:
            matchLabels:
              app: bit-script-management
```

```
    ports:
      - protocol: TCP
        port: 8080

   # Allow DNS lookups (required for service discovery!)
   - to:
      - namespaceSelector: {}
     ports:
      - protocol: UDP
        port: 53
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/networkpolicy-job.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: job-scripts-policy
  namespace: {{ .Values.global.namespaces.scripts }}
spec:
  podSelector:
    matchLabels:
      app: scriptjob
  policyTypes:
    - Egress
  egress:
    - to:
       - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: {{ .Values.global.
               namespaces.public }}
         podSelector:
          matchLabels:
            app: bit-script-management
     ports:
       - protocol: TCP
         port: 8080
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/networkpolicy-default-deny.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress-egress
  namespace: {{ .Values.global.namespaces.scripts }}
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress: []
```

```
  egress: []
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/deployment.yaml

```
{{- range .Values.scripts }}
{{- if and .port (ne .name "scriptcron") }} # only make
   Deployments for scripts with ports
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .name }}
  namespace: {{ $.Values.global.namespaces.scripts }}
spec:
  replicas: {{ $.Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .name }}
  template:
    metadata:
      labels:
        app: {{ .name }}
    spec:
      containers:
        - name: {{ .name }}
          image: "{{ $.Values.global.image.repository }}/{{ .name
             }}:{{ $.Values.global.image.tag }}"
          imagePullPolicy: {{ $.Values.global.image.pullPolicy }}
          ports:
            - containerPort: {{ .port }}
          securityContext:
            runAsNonRoot: true
            runAsUser: 1000
            readOnlyRootFilesystem: true
            allowPrivilegeEscalation: false
            capabilities:
              drop: ["ALL"]
---
apiVersion: v1
kind: Service
metadata:
  name: {{ .name }}
  namespace: {{ $.Values.global.namespaces.scripts }}
  labels:
    app: {{ .name }}
spec:
  type: ClusterIP
  selector:
    app: {{ .name }}
```

```
  ports:
    - port: {{ .port }}
      targetPort: {{ .port }}
{{- end }}
{{- end }}
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/cronjob-scriptcron.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: scriptcron
  namespace: {{ .Values.global.namespaces.scripts }}
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      ttlSecondsAfterFinished: 60
      template:
        metadata:
          labels:
            app: scriptcron
        spec:
          restartPolicy: OnFailure
          containers:
            - name: scriptcron
              image: "{{ .Values.global.image.repository }}/
                 scriptcron:{{ .Values.global.image.tag }}"
              imagePullPolicy: {{ .Values.global.image.pullPolicy }}
              volumeMounts:
                - name: config-volume
                  mountPath: /config
                  readOnly: true
              securityContext:
                runAsNonRoot: true
                runAsUser: 1000
                readOnlyRootFilesystem: true
                allowPrivilegeEscalation: false
                capabilities:
                  drop: ["ALL"]
          volumes:
            - name: config-volume
              configMap:
                name: scriptcron-config
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/configmap-scriptcron.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: scriptcron-config
  namespace: {{ .Values.global.namespaces.scripts }}
data:
  bit-script-mgmt-url: "http://{{ .Release.Name }}-bit-script-
     management.{{ .Values.global.namespaces.public }}.svc.
     cluster.local:8080/script-async"
```

/workdir/bitplatform/helm-charts/bit-python-scripts/templates/configmap-job.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scriptjob-config
  namespace: {{ .Values.global.namespaces.scripts }}
data:
  bit-script-mgmt-url: "http://{{ .Release.Name }}-bit-script-
     management.{{ .Values.global.namespaces.public }}.svc.
     cluster.local:8080/script-async"
```

//INGRESS

/workdir/bitplatform/helm-charts/ingress/Chart.yaml

```
apiVersion: v2
name: ingress
description: Helm chart for ingress
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/ingress/templates/ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: bit-ingress
  namespace: {{ .Values.global.namespaces.public }}
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
spec:
  ingressClassName: nginx
  tls:
    - hosts:
       - localhost
      secretName: bit-platform-tls
  rules:
    - http:
       paths:
```

```
      - path: /bit-user-management(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: {{ .Release.Name }}-bit-user-management
            port:
              number: 8080
      - path: /bit-resources(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: {{ .Release.Name }}-bit-resources
            port:
              number: 8080
      - path: /bit-script-management(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: {{ .Release.Name }}-bit-script-management
            port:
              number: 8080
```

/workdir/bitplatform/helm-charts/bit-gateway/Chart.yaml

```
apiVersion: v2
name: bit-gateway
description: Helm chart for bit-gateway
type: application
version: 0.1.0
appVersion: "1.0.0"
```

/workdir/bitplatform/helm-charts/bit-gateway/templates/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-bit-gateway
  namespace: {{ .Values.global.namespaces.public }}
spec:
  type: ClusterIP
  selector:
    app: bit-gateway
  ports:
    - port: 8080
      targetPort: 8080
```

/workdir/bitplatform/helm-charts/bit-gateway/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: {{ .Release.Name }}-bit-gateway
  namespace: {{ .Values.global.namespaces.public }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: bit-gateway
  template:
    metadata:
      labels:
        app: bit-gateway
    spec:
      containers:
        - name: bit-gateway
          image: "{{ .Values.global.image.repository }}/bit-gateway
             :{{ .Values.global.image.tag }}"
          imagePullPolicy: {{ .Values.global.image.pullPolicy }}
          ports:
            - containerPort: 8080
```

Appendix C

Commands

Commands to deploy the infrastructure:

>gcloud auth login

>gcloud config set project PROJECT_ID

>terraform init

>terraform apply

To associate local Kubectl with the new cluster:

>gcloud container clusters get-credentials bit-cluster

>helm install bit-platform ./commun –set ingress.enabled=true