Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Development of a Microservices-Based Platform for Timesheet Management at the University of Évora

Carlos Manuel Machado Palma

Orientador(es) | Pedro Salgueiro

Vitor Beires Nogueira

Évora 2025

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Development of a Microservices-Based Platform for Timesheet Management at the University of Évora

Carlos Manuel Machado Palma

Orientador(es) | Pedro Salgueiro
Vitor Beires Nogueira

Évora 2025

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	Teresa Gonçalves (Universidade de Évora)
Vogais	José Saias (Universidade de Évora) (Arguente) Pedro Salgueiro (Universidade de Évora) (Orientador)

Évora 2025

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family, especially my parents, Carlos and Isabel, my sister Luísa, my grandparents Tininha and Chico, my cousins José Francisco, Carolina, and Bárbara, and my uncle and aunt, José and Cristina, for their unwavering support throughout my academic journey.

I am also profoundly grateful to my girlfriend, Nádia, who has stood by my side during the majority of my academic path. Her constant encouragement and motivation have inspired me to persevere and strive for improvement every day.

Furthermore, I wish to thank all the professors from the University of Évora, as well as my teachers from Vidigueira Primary School and Diogo de Gouveia High School in Beja, who have influenced my academic development in various ways and contributed to shaping the person I am today.

I would like to extend a special acknowledgment to Professor Fernanda, who had a significant impact at the beginning of my journey by teaching me the foundational concepts that continue to guide me. I am also deeply thankful to Professors Vítor Nogueira and Pedro Salgueiro, who conceptualized the idea for this dissertation and provided continuous guidance and support throughout its development.

Abstract

This dissertation presents the design and development of a microservices-based platform for timesheet management at the University of Évora. The aim of this work is to modernize and streamline the existing timesheet processes, improving efficiency, scalability, and maintainability through the adoption of a microservices architecture. The platform leverages technologies such as Spring Boot, Angular, both Sql and No Sql databases to ensure real-time data processing, fault tolerance, and seamless integration with the existing university systems.

This work begins with a thorough analysis of the current timesheet management workflow, identifying key pain points and requirements. Based on this analysis, a modular architecture was designed, enabling independent development, deployment, and scaling of individual services. Each microservice is responsible for a specific domain function, facilitating better maintainability and testability.

A prototype of the platform was implemented and evaluated in a simulated university environment. The results demonstrate significant improvements in task automation, response time, and overall system reliability.

Keywords: timesheets, microservices, Spring Boot, CI/CD, Docker, GitLab, DevOps

Sumário

Desenvolvimento de uma plataforma baseada em Microserviços para a gestão de Folhas de Horas na Universidade de Évora

Esta dissertação apresenta o desenho e o desenvolvimento de uma plataforma baseada em microserviços para a gestão de folhas de horas na Universidade de Évora. O objetivo deste trabalho é modernizar e simplificar os processos existentes de gestão de horas, melhorando a eficiência, a escalabilidade e a capacidade de manutenção através da adoção de uma arquitetura de microserviços. A plataforma recorre a tecnologias como Spring Boot, Angular e bases de dados SQL e NoSQL, de forma a garantir o processamento de dados em tempo real, a tolerância a falhas e a integração transparente com os sistemas existentes na Universidade.

O trabalho inicia-se com uma análise abrangente do fluxo atual de gestão de folhas de horas, identificando os principais desafios e requisitos. Com base nessa análise, foi concebida uma arquitetura modular que permite o desenvolvimento, a implementação e a escalabilidade independentes de cada serviço. Cada microserviço é responsável por uma função específica do domínio, o que facilita a manutenção e testabilidade.

Foi implementado e avaliado um protótipo da plataforma num ambiente universitário simulado. Os resultados demonstram melhorias significativas na automatização de tarefas, no tempo de resposta e na fiabilidade geral do sistema.

Palavras-chave: folhas de horas, microserviços, Spring Boot, CI/CD, Docker, GitLab, DevOps

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

This chapter introduces the context and motivation for this dissertation, highlighting the limitations of manual timesheet management using spreadsheets and explains the need for a digital solution that improves data accuracy, reduces errors, and enables real-time monitoring. The chapter also outlines the main objectives of the study, focusing on the design and implementation of a scalable, user-friendly timesheet management platform using modern software engineering practices. Finally, the methodology adopted for system development and validation is briefly presented.

1.1 Problem Context

The manual management of employee timesheets through Excel spreadsheets, once a practical solution for smaller organizations, has proven increasingly insufficient as workforce size and operational complexity have expanded. This method is vulnerable to a range of errors—incorrect data entry, overlooked records, and general inconsistencies—primarily due to human oversight and the absence of automated validation. As a result, data integrity is compromised, undermining decision-making processes and leading to unreliable reporting.

Furthermore, the exclusive reliance on spreadsheets complicates the consolidation of information across teams or departments, creating delays in payroll, project tracking, and compliance checks. The lack of integration with other organizational systems restricts real-time insight into workforce distribution and project status, thereby impeding effective strategic planning.

These difficulties highlight the urgent need for a comprehensive digital platform that automates the collection and validation of timesheet data. Such a system would ensure more consistent and accurate information for management purposes, while enabling real-time analytics and reporting. Implementing this solution is critical not only for minimizing errors and reducing administrative workload, but also for enhancing operational efficiency and adapting to changing business requirements. This dissertation presents a modern software architecture that prioritizes maintainability, scalability, and user-friendly accessibility.

1.2 Motivation

The challenges inherent in manual timesheet management emphasize the importance of developing a more efficient, reliable, and scalable solution. Inaccurate or delayed timesheet data not only affects operational efficiency but also has broader implications for project management, employee satisfaction, and organizational compliance. As companies increasingly adopt flexible and remote working arrangements, the limitations of spreadsheet-based tracking become even more pronounced, highlighting the need for a modern, digital approach.

The motivation for this dissertation lies in addressing these deficiencies by designing and implementing a dedicated timesheet management platform that leverages contemporary software engineering practices. By employing a microservices architecture, the system can achieve modularity, scalability, and maintainability, enabling the organization to adapt more rapidly to evolving requirements. Integrating technologies such as Angular for the frontend, Spring Boot for backend services, and Docker for containerized deployment ensures that the platform is both user-friendly and operationally robust.

Additionally, implementing continuous integration and continuous deployment (CI/CD) pipelines enhances the system's reliability and accelerates the delivery of new features, while automated validations reduce human error and improve data quality. The development of this platform is therefore motivated by the desire to transform a historically error-prone, time-consuming process into an efficient, transparent, and dependable solution, ultimately contributing to better organizational performance and informed decision-making.

1.3 Objectives

The main goal of this dissertation is to design and create a digital timesheet management platform. This platform aims to overcome the issues of manual spreadsheet processes by offering a reliable, scalable, and user-friendly solution. The resulting system should improve data accuracy, lessen administrative work, and allow for real-time monitoring and reporting while following modern software engineering practices.

To reach this overall goal, several specific objectives are outlined. First, the system's requirements, both functional and non-functional, need to be analyzed and formalized to make sure the proposed solution meets the needs of the organization. Based on these requirements, a microservices-based architecture will be created to support modularity, scalability, and maintainability, this approach will help the system evolve and adjust to new situations over time. The services will be built with Spring Boot, Java and Maven for backend functions and Angular for the frontend. This setup will ensure smooth interaction between components and provide a consistent user experience. Special attention will be given to including automated validation mechanisms, which are crucial for reducing human error and enhancing the reliability of recorded data. The system will be designed using a configurable approach so that validations, urls and all the other possible variables are easy to change if needed. Lastly, continuous integration and continuous deployment practices will be added using GitLab pipelines to ensure automated builds, testing, and deployment. Docker will be used to support containerized deployment and help with the transition between development and production environments. Together, these objectives will help create a strong platform that can turn timesheet management into an efficient, clear, and sustainable process.

1.4 Methodology

The approach taken in this dissertation follows a clear path that blends theoretical research with practical application. The first step involved a thorough literature review focused on time management systems, software architecture models, and current DevOps practices. This review laid the groundwork for spotting the shortcomings of existing methods and for outlining the needs of a more efficient and scalable solution. Special focus was given to microservices, continuous integration and deployment, containerization technologies, frontend frameworks like Angular, and other tools that could assist in implementing the system.

After the analysis, the requirements for the system were gathered and clearly defined. Both functional and non-functional needs were taken into account, focusing on reliability, scalability, usability, and security. This stage closely matched the practical issues found in the manual timesheet management process, making sure the system design directly responded to the organizational context.

Once the requirements were set, the solution's architecture was crafted using a microservices model. The chosen technologies—Spring Boot, Java, and Maven for backend services, Angular for the frontend, and Docker for containerization—were selected for their modularity, maintainability, and ability to meet future demands. This architecture was enhanced by using GitLab pipelines and Docker Hub Repositories for continuous integration and deployment, which ensured the system could be built, tested, and delivered reliably and automatically.

The implementation stage transformed the architectural design into a functioning prototype. Each microservice was developed separately, tested individually, and then combined to create the full platform in a local setting. The frontend was developed alongside, concentrating on usability and consistency for a smooth interaction with the backend services. Care was taken to implement adjustable validation mechanisms that reduced human error and improved data quality. Lastly, the microservices were moved to a new environment provided by the university for testing in a real-world context.

System validation was done through a mix of automated testing and manual evaluation. Unit and integration tests were run to confirm the accuracy of individual components and their interactions, while scalability and usability were examined through targeted experiments and user feedback. This evaluation was crucial to see if the developed platform truly met the goals defined in the early phases of the dissertation.

1.5 Document Structure

This dissertation is organized into eight chapters that provide a detailed account of the research, design, and implementation of the proposed timesheet management platform. Chapter 1 introduces the problem context, the motivation for the work, the objectives, and the chosen methodology. It highlights the limitations of manual timesheet management and explains the need for a digital solution. Chapter 2 presents the current state of the field, analyzing and compares existing approaches to timesheet management while reviewing modern software engineering practices and design frameworks that inform the proposed system. Chapter 3 and Chapter 4 focus on the system requirements and architecture, detailing both functional and non-functional requirements, as well as the overall design decisions, technologies, and models that support the platform. Chapter 5 describes the implementation, explaining how the system was developed, how the various components interact, and how technologies like Spring Boot, Angular, and Docker were integrated to create a cohesive solution. Chapter 6 is dedicated to validating and evaluating the platform, covering testing methods, performance assessment, and user feedback to show how the solution addresses the initial problems. Finally, Chapter 7 and Chapter 8 conclude the dissertation, summarizing the main contributions, discusses the work's limitations, and suggests potential directions for future research and system development.

Chapter 2

Background and State of the Art

This chapter examines the current technologies and tools relevant to timesheet management systems, focusing on both frontend and backend solutions, data handling, and deployment practices. It reviews popular frontend frameworks, libraries for exporting and generating reports, API communication methods, and containerization platforms, highlighting their strengths and typical use cases. A critical analysis of existing systems identifies limitations and gaps, particularly in educational environments, providing the basis for designing a flexible, scalable, and domain-specific timesheet platform tailored to academic institutions.

2.1 Timesheet Management Systems

Several commercial time tracking systems are already in use within the educational sector. These systems are designed to streamline the process of logging, monitoring, and analyzing the hours worked by teachers and other staff members. Below is an analysis of some notable systems and the technologies they employ. Table 2.1 is a structured comparison of these systems.

Jibble

Jibble is a versatile time tracking system widely used in educational institutions due to its flexibility in supporting various check-in methods, including facial recognition, GPS-based tracking, and mobile app integrations. This system is built with modern web technologies such as React and Node.js, offering a scalable and responsive user interface. Jibble also integrates seamlessly with third-party tools like Slack and Microsoft Teams, making it a popular choice for hybrid learning environments [1].

Acadly

Acadly is primarily a classroom management tool that incorporates automatic attendance tracking and student engagement features, such as polls and quizzes. It integrates with Learning Management Systems (LMS) through APIs and uses cloud technologies for real-time data processing. The system's backend is built on Amazon Web Services (AWS), ensuring high availability and scalability. Its frontend is developed with Angular, allowing a dynamic user experience across devices [2].

Alma

Alma is a comprehensive Student Information System (SIS) that includes time tracking and attendance management among its many features. Designed for integration with Google Classroom and other educational tools, Alma uses RESTful APIs to facilitate data exchange. Its architecture relies on cloud-native technologies, providing secure and reliable data management. The platform's frontend is built with Vue.js, chosen for its lightweight and adaptable nature [3].

BrioHR

BrioHR, although primarily a Human Resources Management System (HRMS), offers robust time tracking capabilities tailored for educational institutions. It leverages machine learning algorithms for attendance pattern analysis and uses React for its frontend, combined with a Python-based backend for efficient data processing. The platform also supports integration with payroll systems, enhancing its utility in managing staff time and attendance [4].

Table 2.1: Comparison of Educational Timesheet Systems

System	Main Features	Technology Stack	Limitations in Academic Context
Jibble	Face/GPS check-in, Slack/Teams integration	React, Node.js, Cloud backend	Limited project-based reporting
Acadly	Attendance + engagement tools (polls, quizzes)	Angular frontend, AWS backend	Focused on students, not staff timesheets
Alma	Student Information System with time tracking	Vue.js frontend, REST APIs, Cloud-native	Primarily SIS, lacks dedicated staff workflow
BrioHR	HR management with ML attendance analysis	React frontend, Python backend	Geared to HR, not academic research/projects

2.2 Software Architectures

The option of choosing whether to develop an application in microservices or monolithic architecture models is one of the most important decisions that software architects have to make as it has ramifications in the long-term as well. This section examines these two styles in a comparative approach in terms of their basic structure, strengths and shortcomings.

2.2.1 Monolithic Architecture

A monolithic architecture is a software design where all parts of the system, like the user interface, business logic, and data access layers, are built together as one cohesive unit with a shared codebase. Despite its simplistic design, this structure presents several constraints as changing any part of the system requires developers to update and redeploy the entire application, which can complicate the maintenance and updating process.

A key benefit of monolithic architecture is its ease of use in the initial development phases, as it enables simpler management with all components in one project. This can result in quicker development times because there is no requirement to handle the intricacies of distributed systems, simplifying the process of monitoring changes, debugging, and testing the entire system. Deployment is also simpler because the complete application is deployed as a single executable bundle.

As the application expands, monolithic architectures can become harder to handle because the system is not modular, restricting scalability as individual components cannot be scaled separately. This also negatively impacts the system's reliability, as a failure in a single component can propagate and affect the entire application. Moreover, any modification to the underlying foundation or technologies used in the monolithic architecture requires a complete system redeployment, which may result in increased downtime as well as additional time and cost overheads. The structure also lacks adaptability in incorporating new technologies due to its strong connection to the original design decisions[5].

2.2.2 Microservices Architecture

A microservices architecture is a modern approach to software design in which an application is structured as a collection of small, loosely coupled services, each responsible for a specific function. Unlike monolithic architectures, in which the entire application is developed as a single cohesive unit, microservices architectures enable each service to be developed, deployed, and maintained independently. Each microservice encapsulates its own business logic and typically manages its own database, functioning as a self-contained unit. This autonomy in deployment and management allows individual services to scale independently, providing greater flexibility and adaptability as the system evolves.

One of the primary benefits of microservices is their scalability, since each service is modular, specific components can be scaled without impacting the entire system, which makes it easier to respond to increasing user demand in different parts of the application. This modular nature also facilitates more frequent updates, as changes to one microservice do not require redeploying the entire application, reducing downtime and improving system agility. Additionally, microservices architectures are well-suited to DevOps practices, particularly continuous integration and delivery (CI/CD), as they allow for faster iteration and more reliable rollbacks when problems occur.

However, the independence of each service introduces significant complexity. Managing multiple services, each with its own codebase, database, and deployment pipeline, can lead to high complexity where the sheer number of services in the system creates organizational and technical challenges, making it harder to maintain consistency across the architecture. Coordination between teams also becomes more complex, as each service might be developed by different groups, which requires effective communication and integration strategies to ensure smooth operation.

Moreover, while microservices offer flexibility in terms of technology choices, where different teams can use different programming languages or frameworks, this flexibility can lead to a lack of standardization, over time, this can make system maintenance harder, as the diversity of tools and approaches across services increases. The cost of infrastructure also rises with microservices, as each service might require its own dedicated resources for hosting, monitoring, and deployment, potentially leading to exponential increases in operational expenses.

In terms of reliability, while microservices are often more resilient than monolithic systems—since the failure of one service does not necessarily bring down the entire application—debugging can be more challenging. Distributed services mean distributed logs and metrics, which complicates tracing the source of issues, especially when multiple services are involved in a single business process[5].

2.2.3 Architecture Advantages and Disadvantages

Both architectures are valid approaches and usually they are used depending on the requirements of the system.Table 2.2 sums the main differences between the 2 architectures.

Monolithic Architecture – Advantages and Disadvantages

Monolithic Architecture has the distinct benefits and drawbacks. Monolithic is particularly beneficial because it is simpler to develop; in this case, the whole code is stored in one place allowing developers to work on one codebase, hence making the development process easier, on top of that, deploying is simple enough since the application is wrapped in one binary file.

Unfortunately, monolithic architectures have also disadvantages that need to be considered. Maintenance is a problem that compounded as the application size increased, with the larger codebase adding the difficulty of management and updates. Reliability is another issue, since a malfunction in an application component will most probably cause a complete system breakdown. Furthermore, availability issues are often linked to the need to redeploy the entire application when updates or changes are applied, which is a common practice in monolithic architectures and inevitably introduces the risk of system downtime. Additionally, monolithic systems lack flexibility, making scalability a significant challenge, as individual components cannot be scaled independently.

Microservices Architecture – Advantages and Disadvantages

Microservices architecture presents distinct advantages and disadvantages, making it a compelling choice within contemporary software development practices. A strong advantage of it is easy maintenance whereby each microservice serves a specific business capability, so the developers can get to the functionality and thus manage them more effectively. The architecture is also reliable, because, if one microservice failed, the other services are not affected, so the risk of the whole system failure is decreased. This architecture also promotes high availability, as a new version of a microservice is usually deployed with minimal downtime. The other big thing is scalability, microservices can be scaled independently, which means you can use the resources in a smart way.

However this architecture presents some drawbacks as well, the complexity of deployment is rising due to the distributed nature of microservices, a careful management and orchestrating of many services is needed to be done. Data integration can be a problem, since it is more challenging to maintain effective communication and consistency between independent services, which often requires complex solutions to be implemented.

Last but not least, the cost problem: using a microservices-based system usually involves more expenses, as it requires multiple environments, hosts, and sometimes even the involvement of separate cloud and infrastructure teams that handle the complexity.

Table 2.2: Advantages and Disadvantages of Monolithic and Microservices Architectures

Aspect	**Monolithic**	**Microservices**
Deployment	Single deployable unit; simpler initial setup	Independent service deployment; supports CI/CD
Scalability	Whole system scales as one piece	Individual services scale independently
Maintenance	Tight coupling increases technical debt	Easier to isolate and update components
Reliability	Failure in one module can crash the entire system	Failure in one service rarely affects others
Cost/Complexity	Lower infrastructure cost, simpler debugging	Higher infra cost, complex monitoring

2.3 Frameworks and Tools for Microservices Development

In software development, a framework provides a pre-defined structure that serves as a skeleton for building applications. It abstracts complex programming tasks by offering reusable components, tools, and libraries, allowing developers to focus on implementing application logic rather than underlying infrastructure. Frameworks improve consistency, efficiency, and scalability, making them essential in modern software development practices.

Frameworks are crucial for several key reasons. First, they help standardize the development by implementing consistent coding and design patterns, which helps teams easily collaborate and perform codebase maintenance. Second, they greatly enhance efficiency by making available ready-to-use components for common functionalities like database interactions, security, and user authentication, reducing development time and effort. Third, frameworks provide abstraction to mask low-level details of programming and enable developers to focus on the core feature set of an application. Further, scalability is considered in the design of frameworks, which makes them very suitable for architectures like microservices that must meet increasing demands. Lastly, established frameworks boast of extensive documentation, active community support, and regular updates that make them reliable to last for a long period.

In the context of microservices, frameworks play a pivotal role in simplifying the development, deployment, and management of distributed services. They provide essential mechanisms such as service discovery, communication, and resilience, which are critical for ensuring that independent services operate reliably and efficiently. Among the many frameworks available, Spring Boot and Spring Cloud stand out as particularly effective for building scalable and maintainable microservice architectures. The following sections present an overview of these frameworks and their key features.

2.3.1 Spring Boot

Spring Boot[6] is a robust framework that was designed to make it easier to build web applications. Based on the original Spring project, it greatly reduces the amount of configuration necessary, allowing developers to focus on creating strong microservices. Auto-configuration, standalone deployment, and numerous starter dependencies are only a few of its key features, which collectively make it possible to quickly set up and create. In addition, Spring Boot has excellent support for creating RESTful services, which makes it particularly suitable for microservice architecture.

2.3.2 Spring Cloud

Spring Cloud [7] complements Spring Boot by providing a set of tools and design patterns for microservices. It provides service discovery, circuit breakers, and centralized configuration management, each of which is an essential characteristic in addressing the complexities of microservices. The seamless integration between Spring Cloud and Spring Boot is very powerful, allowing developers to build robust and scalable microservices.

2.3.3 Relational and Non-Relational Databases

Databases are a critical component of modern software systems, serving as repositories for structured and unstructured data, they are broadly categorized into relational databases (SQL) and non-relational databases (NoSQL), each with distinct characteristics and use cases.

Relational databases are based on a structured schema, typically organized into tables with rows and columns, each table represents an entity, and relationships between tables are established using keys. This approach provides a high level of consistency and is ideal for systems requiring complex queries and transactions, such as financial applications. Popular relational database technologies include MySQL, PostgreSQL, and Microsoft SQL Server. These databases are ACID-compliant, ensuring reliable transactions and data integrity, which makes them suitable for use cases where consistency is critical [8].

On the other hand, non-relational databases offer greater flexibility and scalability by departing from the rigid tabular schema of relational systems, instead, they store data in various formats, including key-value pairs, documents, column families, or graphs. This diversity enables non-relational databases to handle diverse and evolving data models efficiently, making them an excellent choice for modern, distributed applications and big data use cases [9]. Table 2.3 summarizes the main differences between the two types of databases.

Relational Databases

Relational databases have firmly established themselves as the go-to choice for applications that need reliable transactions, well-organized data, and robust referential integrity.

The relational model, introduced by Codd, leverages normalization to minimize redundancy and utilizes primary and foreign keys to maintain integrity between different entities and SQL, being a declarative language, allows for complex queries, joins, and aggregations—key components for advanced analytics and enterprise-level reporting[8].

The ACID principles (Atomicity, Consistency, Isolation, Durability) guarantee dependable transactions, even during system failures or high-load scenarios,this reliability makes relational databases particularly suitable for critical applications like financial systems, inventory management, reservation platforms, and healthcare solutions.

Modern relational database management systems (RDBMS) have advanced to facilitate scalability and high availability through features like synchronous and asynchronous replication, sharding, horizontal partitioning, and fully managed cloud offerings (such as Amazon RDS, Azure SQL Database, and Google Cloud SQL). Many database engines also include features like stored procedures, triggers, and advanced indexing methods (like B-Tree, GiST, and GIN) to enhance performance for large queries.

Popular technologies in this space include PostgreSQL, known for its extensibility (including geospatial functionality through PostGIS), MySQL/MariaDB, favored for large-scale web applications, and Microsoft SQL Server, which excels in enterprise settings with strong business intelligence capabilities. These examples highlight the maturity and rich ecosystem that make relational databases a vital part of contemporary software development.

Non-Relational Databases

The non-relational databases are an important part of modern, distributed system which is designed to scale horizontally, able to handle unstructured data with high-performance query in distributed environment. They provide flexibility in how data is modeled, which makes them particularly useful for heterogeneous and rapidly changing datasets than their relational counterparts. Non-relational databases are further divided into several types, which are designed for different use cases.

Document databases store data in JSON-like documents, allowing developers to structure their data intuitively and flexibly. This approach is ideal for scenarios where the schema may evolve over time. Technologies such as MongoDB and Couchbase are commonly used in content management systems and real-time analytics due to their robust support for hierarchical data structures and dynamic queries.

In terms of cost, both MongoDB and Couchbase offer open-source editions suitable for cost-sensitive projects. MongoDB Community Edition is freely available and can be deployed locally or on private servers. However, managed solutions such as MongoDB Atlas incur charges based on resource usage and scalability requirements. Similarly, Couchbase Community Edition is open-source and free to use with the Apache 2.0 License as well. The Community Edition itself offers a great set of features but lacks some advanced functionalities and support available on the Enterprise edition. If you need enterprise-grade capabilities, both MongoDB and Couchbase offer commercial paid for managed services with enhanced functionality alongside professional support.[10, 11]

Key-value stores represent another category, pairing unique keys with corresponding values, these databases are highly efficient for simple operations, such as lookups and session storage. Examples include Redis, which is widely used for caching, and Amazon DynamoDB, which provides high-speed data access in cloud environments. Key-value stores are often employed in applications where low latency is critical [12].

For analytical workloads, column-family stores excel by storing data in columns rather than rows, facilitating fast aggregation and retrieval of large datasets. Technologies like Apache Cassandra and HBase are prominent in big data ecosystems, particularly for time-series data and distributed logging systems.

Lastly, graph databases are specialized for applications that require modeling relationships between entities, they are highly effective for use cases such as social networks, fraud detection, and recommendation engines. Examples include Neo4j, a popular open-source option, and Amazon Neptune, a managed graph database service. These systems leverage graph structures to enable complex queries over connected data efficiently [13].

All non-relational database types boast strengths of their own. MongoDB is more suited for rapid prototyping and flexible schema design, Apache Cassandra outshines at handling huge amounts of data with low latency in distributed environments. Databases are just a generic term for a place to store your data, different requirements of the application development will ensure whether you need an SQL database or NoSQL database such as queries which are more complex and larger in amount are done with full speed through Massive Processing Framework which is possible by storing the records on distributed clusters.

Table 2.3: Relational vs. Non-Relational Databases

Feature	**Relational (SQL)**	**Non-Relational (NoSQL)**
Schema	Fixed schema, normalized tables	Flexible or schema-less documents/keys/graphs
Consistency	ACID transactions, strong consistency	Eventual consistency (varies by type)
Scalability	Vertical scaling, limited horizontal sharding	Designed for horizontal scaling and big data
Query Language	SQL (complex joins, aggregations)	APIs or custom query languages (e.g., MongoDB queries)
Best For	Financial, transactional systems	Content management, real-time analytics, IoT

2.3.4 Frontend Tools for Interactive User Interfaces

Developing an intuitive and efficient user interface is crucial for a system that allows employees to log their working hours. Popular frontend frameworks offer different approaches to achieve this goal.

React, for instance, is a lightweight and flexible JavaScript library developed by Facebook, known for its component-based architecture that promotes re-usability and a virtual DOM that enhances rendering performance. It excels in projects requiring fast rendering, such as dashboards with frequent state changes. Additionally, its extensive ecosystem of third-party libraries provides significant flexibility for custom and scalable solutions, making it an excellent choice for development teams that value adaptability and a relatively shallow learning curve for new developers [14].

Angular, developed by Google, offers a comprehensive framework with built-in features such as two-way data binding, dependency injection, and a powerful command-line interface (CLI). This framework is particularly suited for large-scale, enterprise-grade applications that require a well-structured approach. Its strict architecture and integration with TypeScript promote maintainability and help reduce errors in complex systems. Teams already familiar with TypeScript or those managing complex frontend logic will benefit from Angular's robust and opinionated approach [15].

Vue.js is often considered a middle ground between React and Angular, offering a combination of simplicity and flexibility. It features a gentler learning curve compared to Angular while maintaining a robust reactivity system similar to React. Vue is particularly advantageous for small to medium-sized projects that require rapid development and gradual scalability. Its straightforward syntax and flexibility make it appealing to teams seeking a balance between ease of use and powerful features, especially in environments with limited resources or tight deadlines [16].

Choosing between these frameworks depends largely on the project's complexity and team expertise. React is ideal for projects prioritizing flexibility and performance with a broad ecosystem. Angular is best suited for enterprise-level applications requiring a structured and scalable approach, while Vue is an excellent choice for teams aiming for fast development with a simpler framework. Table 2.4 shows in a visual manner the main differences between the various frontend frameworks.

Table 2.4: Comparison of Major Frontend Frameworks

Framework	**Language**	**Strengths**	**Typical Use Case**
React	JavaScript/JSX	Flexible ecosystem, fast virtual DOM rendering	Dashboards and apps with frequent UI updates
Angular	TypeScript	Full-featured, opinionated, strong tooling (CLI, DI)	Large enterprise apps with strict architecture
Vue.js	JavaScript	Gentle learning curve, progressive adoption	Small to medium projects needing quick ramp-up

2.3.5 Data Transmission Approaches

Effective data transmission is essential in a timesheet management system, as it ensures that information about employee hours, project assignments, and approvals is accurately and efficiently exchanged between the frontend, backend, and external services. Choosing the right communication approach impacts the system's performance, scalability, and reliability, directly affecting real-time updates, report generation, and integration with other organizational platforms. Therefore, careful consideration of data transmission methods is crucial to maintain consistency, reduce errors, and provide a seamless user experience. In this context, the following sections focus on REST and SOAP, two widely adopted approaches that offer complementary advantages in terms of simplicity, reliability, and support for enterprise-level operations, which are particularly relevant for the needs of our platform. Additionally, modern query languages such as GraphQL are also considered, offering more precise and flexible data retrieval capabilities that address some of the limitations of traditional API approaches, particularly in systems with complex and dynamic data requirements.

REST (Representational State Transfer)

REST is an architectural style that relies on stateless communication and resource-oriented interactions. Built on top of the HTTP protocol, it employs standard HTTP methods—such as GET, POST, PUT, and DELETE—to manipulate resources identified by URLs. One of REST's primary advantages is its simplicity and ease of use, which has contributed to its widespread adoption for web services.

RESTful services are designed to be stateless, meaning that each request from a client contains all the information necessary for the server to fulfill that request. This statelessness enhances scalability and makes it easier to manage sessions. Additionally, REST can deliver data in multiple formats, such as JSON, XML, or HTML, making it flexible for various client applications [17].

SOAP (Simple Object Access Protocol)

SOAP, on the other hand, is a protocol designed for exchanging structured information in the implementation of web services, relies on XML as its message format and usually operates over HTTP, but can also work with other protocols such as SMTP or TCP. It is characterized by its formal structure and relies on a set of standards that ensure message integrity, security, and transaction compliance.

One of the primary advantages of SOAP is its ability to support complex operations through its extensibility and built-in error handling. SOAP services typically adhere to a strict contract defined by the Web Services Description Language (WSDL), which describes the available methods and their input/output requirements. This makes SOAP a suitable choice for enterprise-level applications that demand a high degree of reliability and security [18].

GraphQL (Query language for APIs)

GraphQL is a contemporary query language and runtime for APIs, created to offer a more efficient and flexible option compared to standard RESTful APIs. Originally created by Facebook in 2012 and made available to the public in 2015, GraphQL solves data retrieval problems like over-fetching and under-fetching by enabling clients to accurately define the data they require.

In contrast to REST, where every endpoint is predetermined with set data structures, GraphQL utilizes a solitary endpoint and provides a schema outlining the types available and their connections. Customers have the ability to construct queries to obtain the exact information they need in one request, eliminating the necessity for multiple trips to the server.

Although GraphQL is increasingly gaining popularity and offers several advantages, it continues to face challenges. Despite growing adoption within the technology community, there remains a lack of comprehensive empirical studies evaluating its use in real-world industrial and governmental contexts. Research on aspects such as performance under high load, security considerations, and best practices for schema design and maintenance is still ongoing.

In general, GraphQL represents a significant shift in API design, providing a more flexible and effective method for data communication, particularly in situations with complicated data demands and specific client requirements [19].

Choosing Between REST, SOAP, and GraphQL

Choosing between REST, SOAP, and GraphQL involves evaluating factors such as simplicity, performance, security, interoperability, and the specific requirements of the application. Each approach offers distinct advantages and trade-offs depending on the system's complexity and data needs. Table 2.5 is a structured comparison of these approaches.

In terms of simplicity, REST stands out as lightweight and easy to implement, making it suitable for rapid development and web-based applications. GraphQL, while slightly more complex due to schema definition, allows clients to request exactly the data they need, reducing over-fetching and under-fetching compared to REST. SOAP is more complex, adhering to strict standards that suit enterprise-level systems with extensive feature requirements.

Regarding performance, REST's stateless nature and scalability make it appropriate for high-load scenarios. GraphQL can improve efficiency by minimizing the number of requests required to fetch precise data, which is particularly beneficial for applications with complex or nested data structures. SOAP, while capable of handling complex operations, introduces additional overhead that can impact performance in certain contexts.

Security is another critical consideration. SOAP provides robust built-in security through WS-Security, ensuring message integrity and reliability. REST relies on HTTPS and external protocols such as OAuth, offering less integrated security compared to SOAP. GraphQL's security depends largely on implementation practices, including query depth limiting and authentication mechanisms, requiring careful configuration to match enterprise security needs.

Regarding interoperability, SOAP's platform-agnostic nature, supported by WSDL, ensures consistent integration across diverse systems. REST is highly flexible but may require additional documentation for consistent integration. GraphQL provides a strongly-typed schema and a single endpoint, simplifying client-server communication, though it may require more upfront design effort to ensure consistent integration across multiple clients.

In conclusion, REST is ideal for applications prioritizing simplicity, scalability, and rapid development, particularly in web and mobile environments. GraphQL is well-suited for systems with complex or variable data requirements that benefit from precise and efficient queries. SOAP remains the preferred choice for enterprise scenarios demanding strict security, reliability, and support for complex operations [20].

Table 2.5: Comparison of REST, SOAP, and GraphQL

Aspect	**REST**	**SOAP**	**GraphQL**
Simplicity	Simple, lightweight	Complex, strict	Moderate, schema required
Performance	Fast, stateless	Overhead in complex ops	Efficient, precise queries
Security	HTTPS/OAuth	WS-Security, strong	Depends on config
Interoperability	Flexible, manual docs	WSDL ensures consistency	Single endpoint, typed schema
Use Case	Web/mobile apps	Enterprise systems	Complex/variable data

Inter-service Communication

In microservices architectures, communication between services is a critical aspect of ensuring cohesion and reliability across the platform. While REST, SOAP, and GraphQL define how data is structured and transmitted, frameworks like Feign simplify the actual implementation of such communication in Java-based systems.

Feign Clients [21] are declarative REST clients integrated with Spring Cloud that allow developers to define service-to-service calls through annotated interfaces rather than manually managing HTTP connections. This abstraction reduces boilerplate code, improves readability, and allows seamless integration with service discovery mechanisms such as Netflix Eureka or Spring Cloud LoadBalancer. Feign also supports fault tolerance through integration with tools like Hystrix or Resilience4j, providing retries, fallbacks, and circuit breaking in case of service failures.

Besides Feign, other communication mechanisms are commonly used in microservices ecosystems. For synchronous communication, developers may rely on *REST Templates* or *WebClient* (from Spring WebFlux)[22], which provide greater control over HTTP calls but require more manual configuration compared to Feign. For asynchronous communication, event-driven approaches based on message brokers such as Apache Kafka [23] or RabbitMQ [24] are often employed. These enable decoupled, scalable, and fault-tolerant communication patterns where services interact through published and consumed events rather than direct HTTP requests.

By combining declarative clients such as Feign for synchronous, request-response scenarios with event-driven communication for asynchronous use cases, microservices architectures achieve a balance of simplicity, scalability, and resilience, which is essential in complex platforms such as timesheet management systems.

2.3.6 Data Export Libraries

In any timesheet management system, the capability to generate and export documents is essential for reporting, auditing, and record-keeping purposes. Users require the ability to produce downloadable files in formats such as Excel and PDF to efficiently analyze, share, and archive timesheet data. Therefore, it is important to evaluate and select appropriate technologies that facilitate this functionality.

For exporting timesheet data into Excel files, FileSaver.js and XLSX.js provide efficient solutions. FileSaver.js is a lightweight library that enables browsers to save files locally, making it particularly useful for generating downloadable Excel files without relying on server-side processing. Its ease of integration and simplicity are key advantages, especially in scenarios where quick and straightforward downloads are required [25].

On the other hand, XLSX.js offers comprehensive tools for reading, manipulating, and writing Excel files in various formats, supporting more complex data export needs. Its flexibility in handling different file formats and seamless integration with frontend frameworks make it a preferred choice for projects requiring advanced Excel file manipulation [26]. Depending on the project's requirements, FileSaver.js is ideal for straightforward file downloads, while XLSX.js is more suitable for complex data handling

For server-side PDF generation in Java, the most widely used libraries include iText, Apache PDFBox, and OpenPDF.

iText provides a comprehensive API for creating dynamic PDF documents, supporting structured layouts, tables, and embedded graphics. It also offers advanced capabilities such as digital signatures and encryption, making it suitable for scenarios that require secure and well-formatted reports. [27]

Apache PDFBox, on the other hand, is a pure Java, open-source library that enables both the creation and manipulation of PDF files. It supports operations such as text extraction, merging, and splitting, which are particularly valuable for applications that need to process existing PDF content programmatically. [28]

Lastly, OpenPDF is an open-source Java library for PDF generation and manipulation, derived from an earlier version of iText before it adopted a commercial license. It provides functionality for creating structured documents with tables, images, and styled text, while also supporting encryption and digital signatures. Unlike iText, OpenPDF is fully distributed under the LGPL and MPL licenses, which makes it especially suitable for academic and research projects requiring free and permissive licensing. [29]

All three libraries are widely adopted in enterprise Java applications for automated document generation. While iText excels in advanced features and enterprise support, PDFBox is ideal for processing existing documents, and OpenPDF offers a licensing model that is more accessible for open-source, academic, and research contexts.

2.4 CI/CD, Docker, DevOps, GitLab Pipelines

This section explores DevOps practices and tools that facilitate automation, scalability, and reliability in software delivery. Topics include containerization, orchestration, and deployment pipelines.

2.4.1 Containerization Platforms

A containerization platform is a technology that allows for bundling an application and its operating system dependencies, libraries, and configuration files into one self-contained entity called a container. It abstracts the application away from the actual operating system to run uniformly on different environments, whether development, testing, staging, or production.

Containerization platforms are widely used in modern software development, especially in microservices architecture, due to the many advantages they offer. Among the key benefits is *portability.* Containers can be easily deployed across diverse environments, including a developer's laptop, private data centers, and public cloud platforms, provided the container runtime is supported. This flexibility allows for consistent and reliable application deployment, irrespective of the underlying infrastructure.

Another strong benefit of containerization is *isolation.* Each container has its isolated environment in which applications are running, so they won't affect each other's processes, dependencies, or configurations. The isolation contributes to system stability and simplifies the work with complex multi-service environments because it reduces conflicts between dependent components.

In terms of resource efficiency, containers are significantly lighter than traditional virtual machines, as they share the host system's kernel. This architecture reduces the overhead associated with running multiple operating system instances and enables more efficient utilization of computational resources, while also improving application startup times.

Containerization finally ensures the consistent deployment of the whole runtime environment of the application, its dependencies, libraries, and configuration files. It simply ensures that all applications will perform the same in all stages of development and deployment, which in turn reduces the chances of any environment-specific problems and enhances overall reliability related to software delivery pipelines.Building on this concept, the following sections present prominent containerization platforms that exemplify these benefits in practice.

Docker

Docker [30] is the major containerization platform that plays a vital role in microservices deployment by allowing developers to package an application and its dependencies into a container; Docker ensures consistency in the execution of the application across different environments. This model simplifies the deployment, scaling, and management of microservices by enabling them to run in isolation.

Inherent portability in Docker makes it easy to deploy microservices across multiple cloud and on-premises infrastructures.

Kubernetes

Kubernetes[31] provides orchestration for containerized applications, including microservices. It enables automated deployment, scaling, and management of containerized applications, simplifying the task of managing a complex microservices architecture. Starting from automated load balancing or service discovery over self-healing to other features, Kubernetes provides efficient ways for teams to maintain microservices at scale, ensuring high availability and optimal performance. Additionally, Kubernetes offers significant advantages in terms of compatibility, as it integrates seamlessly with Docker and other containerization platforms.

2.4.2 CI/CD and Deployment Pipelines

Continuous Integration (CI) and Continuous Deployment (CD) are fundamental practices in modern software engineering that aim to automate the process of building, testing, and deploying applications. CI/CD pipelines reduce manual errors, accelerate delivery, and ensure that software can be reliably released at any time, which is particularly important in microservices architectures where multiple services evolve independently.[32]

CI focuses on the frequent integration of code changes into a shared repository, accompanied by automated builds and tests. This practice ensures that code changes do not introduce regressions, maintaining system stability while enabling rapid feature development. CD extends this process by automating the deployment of validated changes to staging or production environments, ensuring consistent and reliable delivery across environments.[32]

2.4.3 GitLab Pipelines

GitLab Pipelines [33] provide a practical implementation of CI/CD, enabling teams to define the stages of software delivery—from build and test to deployment—using a single YAML configuration file. GitLab's integration with containerization platforms, such as Docker, allows pipelines to build, test, and deploy containerized applications consistently across different environments.

Key features of GitLab Pipelines include parallel job execution, automatic triggering based on repository events, and seamless integration with version control and container registries. These capabilities enhance automation, improve feedback loops, and facilitate reliable software delivery in complex systems. By leveraging pipelines, development teams can ensure that each microservice in a system is tested and deployed independently, improving scalability, maintainability, and operational efficiency.

In combination with container orchestration platforms like Kubernetes, GitLab Pipelines provide a robust framework for automating microservices deployment. Containers built with Docker can be automatically deployed and scaled, while Kubernetes manages their lifecycle and orchestration, resulting in a fully automated, reliable, and scalable software delivery process.

2.5 Critical Analysis and Research Gap

This section provides a critical synthesis of the state-of-the-art in time tracking systems for educational institutions. Drawing from the previously reviewed systems, architectures, frameworks, databases, frontend tools, communication approaches, APIs, and containerization technologies, it identifies existing strengths, limitations, and research gaps that motivate the proposed solution.

2.5.1 Critical Analysis

Existing commercial solutions such as Jibble, Acadly, Alma, and BrioHR provide robust time tracking and attendance management features, integrating with third-party services and offering cloud-based scalability, while using modern frontend frameworks like React, Angular, and Vue, RESTful APIs, and cloud-native backend architectures, yet they often fail to address the specific needs of educational institutions, such as detailed project-based logging for teaching, research, and administrative duties.

Monolithic architectures offer simplicity in development and deployment but face scalability, maintainability, and flexibility challenges as system complexity grows, whereas microservices architectures provide modularity, independent scalability, and improved reliability when combined with DevOps practices, containerization platforms such as Docker and Kubernetes, and orchestration tools, though they introduce additional complexity in service management, data integration, and infrastructure cost.

Frameworks like Spring Boot and Spring Cloud enable the development of scalable microservices with tools for RESTful API creation, service discovery, and centralized configuration management, relational databases such as PostgreSQL and MySQL ensure strong consistency and ACID compliance, while non-relational databases like MongoDB, Cassandra, and Redis offer flexibility and scalability for heterogeneous or high-volume data. Frontend frameworks and libraries including React, Angular, Vue, FileSaver.js, XLSX.js, iText, and Apache PDFBox support responsive interfaces and report generation, while backend communication relies on REST, SOAP, or GraphQL depending on data complexity, and containerization enhances deployment consistency, portability, and resource efficiency.

2.5.2 Research Gap

Despite these advances, significant gaps persist in educational time tracking systems. These include the lack of domain-specific customization for academic workflows, limited modularity and maintainability in architectures tailored for higher education, underdeveloped reporting and analytics capabilities, challenges in integrating with learning management systems, human resources systems, and financial management platforms, as well as insufficient strategies for handling heterogeneous data while ensuring privacy and compliance. Addressing these gaps motivates the development of a secure, flexible, and research-oriented timesheet system that leverages microservices, Spring Boot, modern frontend frameworks, and robust database strategies to enhance scalability, usability, integration, and analytics within educational institutions.

Chapter 3

System Requirements and Specification

This chapter introduces the functional and non-functional requirements that define the foundation of the proposed timesheet management system. It specifies the system's essential features, including user authentication, role-based access control, timesheet submission, approval workflows, and report generation, while also addressing critical quality attributes such as scalability, performance, security, usability, and maintainability. Furthermore, this chapter formalizes the system's expected behaviour through use cases and user stories, capturing both structured interaction flows and agile development needs from the end-user perspective.

3.1 Functional Requirements

The functional requirements of the proposed system establish the concrete features that must be implemented in order to replace the existing manual process of creating the timesheet with a reliable and automated solution. At its core, the platform must provide mechanisms for managing users registered on university systems, ensuring that authentication and access are properly aligned with institutional standards. Each user must be able to access the system according to their role, whether as an employee responsible for submitting timesheets or as an administrator responsible for validation and approval.

A central functionality of the system is the management of timesheets. Users must be able to create, edit, and submit their working hours in a structured way, while the system automatically performs validations to avoid common errors such as missing information, duplicated entries, or negative values. To reduce the time required for filling in data, the platform must also support the pre-filling of timesheets by consuming REST endpoint services provided by the university, ensuring consistency with existing records.

Administrators must be able to review the submitted timesheets, perform operations such as approving, rejecting, or requesting corrections. Beyond the approval workflow, the system should also allow administrators to filter and search timesheets according to parameters such as user, project, or time period, thereby facilitating the monitoring of activities across the institution. Another essential requirement is the generation of reports. The system must support the production of official documents in PDF format, enabling employees and managers to obtain structured reports that can be archived or shared with other entities. Finally, the entire platform must incorporate validation mechanisms, both at the data-entry stage and at the approval stage, to ensure the integrity, accuracy, and reliability of the information stored and processed.

3.2 Non-functional Requirements

Apart from functional attributes, the proposed system should fulfill a fusion of non-functional requirements which will help maintain the system's quality as well as the sustainability of the system for a prolonged period. Among these requirements, scalability is arguably the most important as the system should be designed to handle a growing number of users and timesheet submissions and manage virtually unlimited concurrent requests without a drop in quality of service. Choosing a microservices architecture satisfies this requirement as it allows the components to scale autonomously based upon the demand.

Performance is a second important factor. The system should be designed to perform common actions, like timesheet submission, timesheet validation and report generation, in a timely manner even during high traffic loads.

Considering the sensitive nature of the data processed by the system, robust security and data protection policies are essential. Users must undergo identity verification and role-based access control to ensure that functionalities are accessible only to authorized persons. Communication between components must be encrypted to prevent leakage or tampering of sensitive information. Additionally, the system must comply with the General Data Protection Regulation (Regulation (EU) 2016/679)

[34], and relevant institutional policies at the University of Évora, including its Data Protection Officer's guidelines. The University of Évora's privacy policies mandate [35] that personal data be collected only for specified, legitimate purposes; that data subjects have rights of access, rectification, erasure, and portability; and that appropriate organisational and technical measures are in place to ensure confidentiality, integrity, and lawful processing of personal data.

Usability is also a key requirement, as the system must be intuitive and easy to use by different categories of users. Employees should be able to record and submit their hours with minimal effort, while administrators should have access to efficient tools for filtering, validating, and approving submissions. The frontend must therefore provide a clear and consistent interface that reduces the likelihood of errors and facilitates adoption across the institution.

Finally, maintainability and reliability are indispensable qualities for ensuring the sustainability of the platform. The system must be designed in a modular and configurable manner, allowing validation rules, endpoints, and other parameters to be adapted without requiring extensive modifications to the codebase. In addition, automated testing and continuous deployment pipelines must support the delivery of new features while minimizing downtime and ensuring consistent system reliability. These non-functional requirements collectively guarantee that the platform will not only meet immediate organizational needs but also remain adaptable to future challenges.

3.3 Use Cases and User Stories

Use cases and user stories are fundamental parts in requirements engineering, providing a structured way to capture functional requirements from the end-user perspective. A use case describes a sequence of interactions between an actor and the system to achieve a specific goal, offering a detailed, scenario-based view of how the system should behave [36, 37]. In contrast, a user story is a concise statement of functionality expressed from the user's viewpoint, typically following the format "As a <role>, I want <goal> so that <reason>" [38, 39]. User stories favour brevity and flexibility, enabling agile teams to prioritise and iterate quickly, whereas use cases provide richer contextual information and support more formal analysis of interactions and edge cases [36, 37].

In this dissertation, these two artefacts complement each other: use cases capture complete interaction flows and system boundaries, while user stories describe granular functional needs that guide iterative development.

This section defines the primary interactions between users and the timesheet management platform. It describes how different actors interact with the system and captures functional needs from the end-user perspective.

3.3.1 Actors

- **Employee** – Academic staff member who records and submits worked hours.
- **Administrator** – Manager or department head responsible for validating, approving, or rejecting timesheets.

3.3.2 Use Cases

The main use cases of the timesheet management platform are described below.

UC1 – User Authentication

Actors: Employee, Administrator

Description: The user logs in using institutional credentials and receives a role-based session token.

Pre-conditions: The user must have a valid institutional account.

Post-conditions: The user is successfully authenticated and authorised to access the system.

UC2 – Submit Timesheet

Actor: Employee

Description: The employee creates or edits a timesheet for a specific project and time period. The system performs automatic validation of the entered data.

Pre-conditions: The user is authenticated.

Post-conditions: The timesheet is stored with the status "Submitted."

UC3 – Approve or Reject Timesheet

Actor: Administrator

Description: The administrator reviews submitted timesheets and can approve them or reject them.

Pre-conditions: At least one timesheet is pending approval and the administrator is authenticated.

Post-conditions: The timesheet status is updated to "Approved" or "Rejected".

UC4 – Generate Reports

Actors: Administrator, Employee

Description: Users export timesheet data to PDF for a chosen timesheet.

Pre-conditions: The user is authenticated.

Post-conditions: A downloadable report file is generated and stored.

UC5 – Search Timesheets

Actors: Administrator, Employee

Description: Users can filter by status and date and administrator can also filter by users; they should also be able to access them.

Pre-conditions: The user and administrator are authenticated.

Post-conditions: The timesheets are shown for the filters specified.

3.3.3 User Stories

The following user stories guide the agile development of the platform:

- **US1** – As an Employee, I want to log in with my university credentials so that I can securely access my timesheets.
- **US2** – As an Employee, I want to create, edit, and submit my monthly timesheet so that my working hours are accurately recorded.
- **US3** – As an Employee, I want automatic validation of entries so that I am alerted to missing or inconsistent data before submission.
- **US4** – As an Administrator, I want to review and approve or reject submitted timesheets so that only valid records are stored for payroll and project tracking.
- **US5** – As an Administrator, I want to search and filter timesheets by status, user, or date so that I can quickly locate specific information.
- **US6** – As an Administrator, I want to generate official PDF reports so that I can share accurate data with human resources or finance departments.

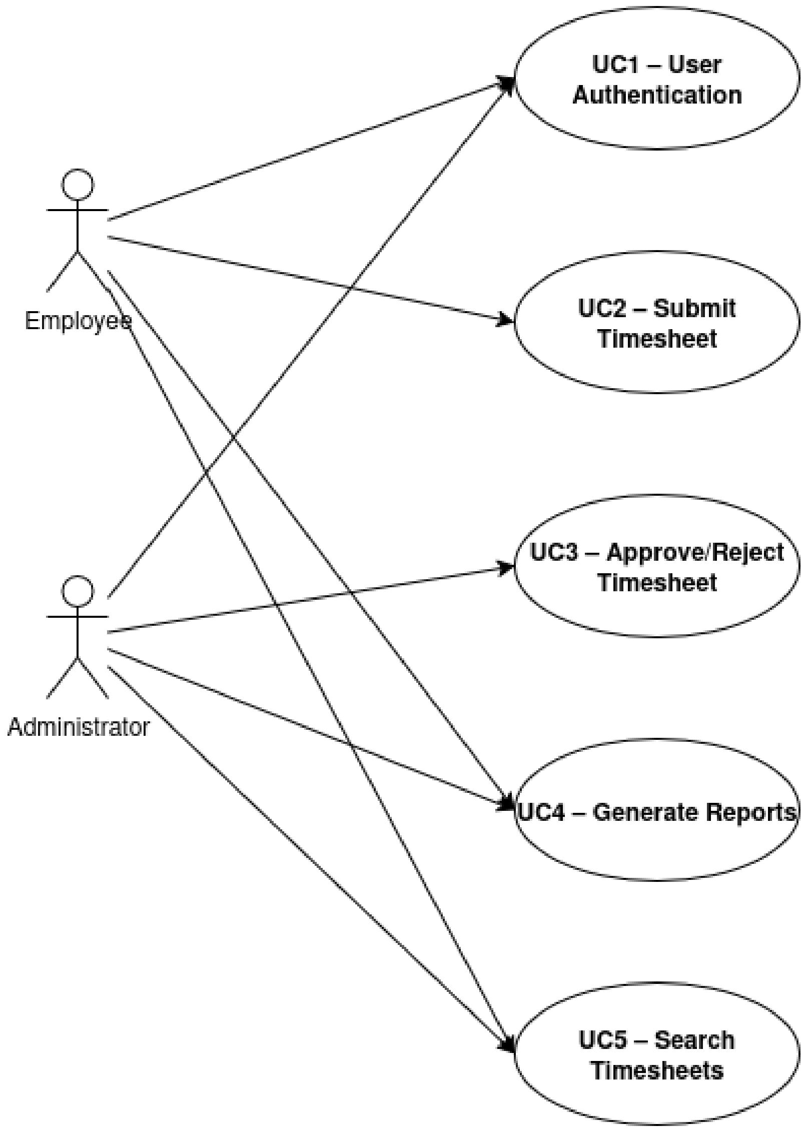


Figure 3.1: Use Case Diagram of the Timesheet Management Platform.

3.4 Data Model

The data model for the proposed timesheet management system has been designed to ensure flexibility, scalability, and maintainability while meeting both functional and non-functional requirements. The design of the proposed data model is strongly motivated by the structure of the manual Excel-based timesheet previously used at the University of Évora. In the manual version, each sheet corresponded to a specific month, containing the employee's identification (name), the calendar of days, the list of projects or activities, and the respective hours filled in per day. Totals and proposed hours were also included, along with summary calculations such as number of workdays and standard daily hours. Although functional, this format presented significant limitations in terms of automation, validation, and integration with other systems. Based on this structure, the data model was refined into a set of core entities to support automation, scalability, and reliability in an academic context.

3.4.1 Core Entities

The **User Entity** corresponds to the employee data previously recorded at the top of the Excel sheet (name, institutional credentials), representing individuals interacting with the system, encompassing academic staff and administrators. Each user is uniquely identified and associated with institutional credentials, while the assigned role determines access permissions and privileges across the platform.

The **Timesheet Entity** corresponds to a timesheet submitted by a user for a specific period, recording information such as the timesheet status, number of workdays, standard daily hours, submission and approval dates, and any associated comments. In addition, metadata fields capture calculated totals and other derived information that support reporting and validation.

Individual activities and project allocations are represented by the **TimesheetEntry Entity**, which is linked to its parent timesheet. Each entry records the date, the activity name, the number of hours worked, the type of activity, and its origin, whether system-generated (pre-filled) or user-edited. Where applicable, entries may also reference external identifiers, ensuring traceability and alignment with data obtained from university systems.

The **DocumentGenerationLog Entity** tracks the creation of PDF reports associated with timesheets. Each log entry includes the user and timesheet it pertains to, the document status at generation time, the total number of entries and hours included, the PDF filename, the version of the generated document, and success or failure information, along with any error messages and the binary content of the generated file. Timestamps record the exact moment of document creation, enabling accountability and auditing. This translation from the manual Excel format into normalized entities guarantees consistency, improves data validation, and enables advanced features such as automated reporting, secure storage, and seamless integration with external university systems.

3.4.2 Entity Relationships

The data model defines clear relationships among the core entities. A single user may own multiple timesheets, each of which may comprise multiple entries. Document generation logs are associated with the relevant timesheet and its corresponding user, supporting comprehensive reporting and traceability. Figure 3.2 is a Entity–Relationship diagram of the data model.

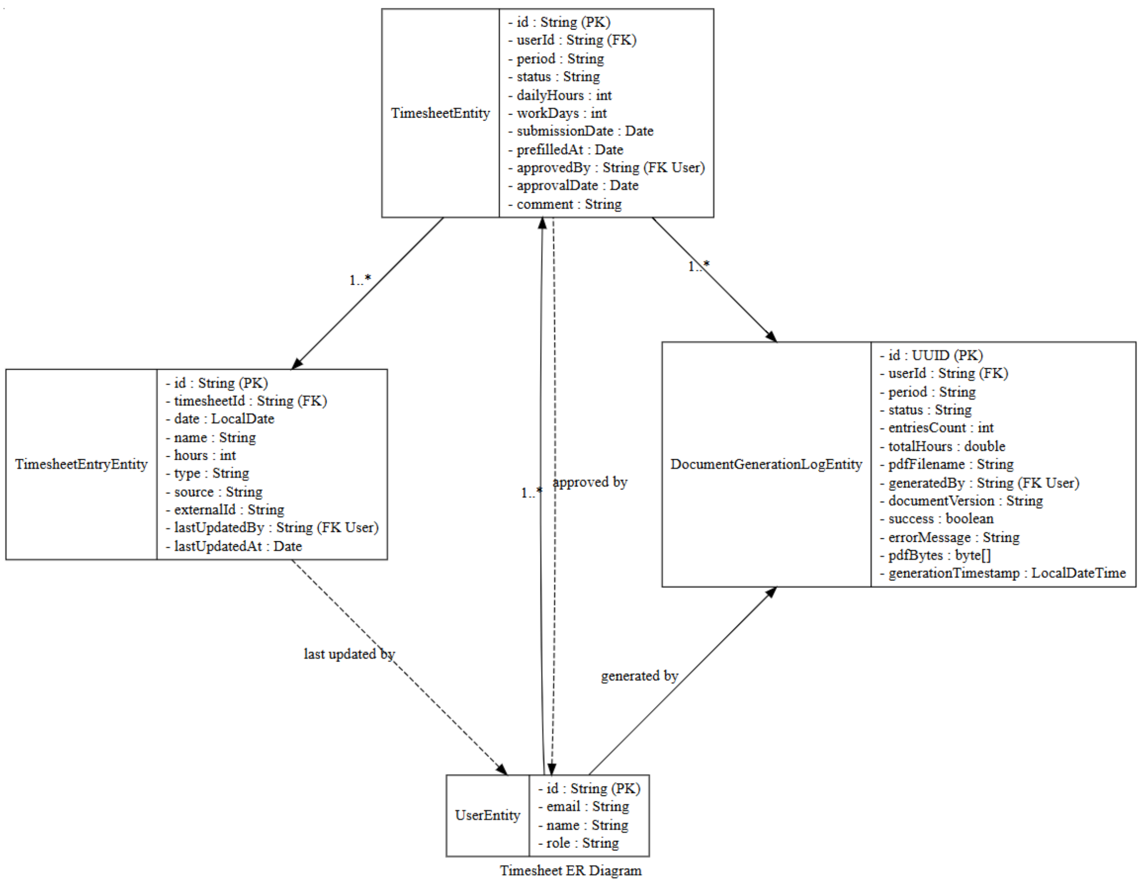


Figure 3.2: Entity–Relationship diagram.

Chapter 4

Architecture and Design

This section presents the rationale behind the technological decisions adopted in the development of the platform. Choices regarding architecture, backend and frontend frameworks, databases, security mechanisms, and deployment strategies were guided by the system's functional and non-functional requirements, as well as by the analysis of existing solutions. Each decision aimed to ensure scalability, maintainability, security, and alignment with the University of Évora's infrastructure, while leveraging prior experience to accelerate development and reduce implementation risks.

4.1 Overall Architecture

The design of the system follows a microservices architecture to provide scalability, maintainability, and modularity. At the frontend, an Angular application provides a responsive and intuitive user interface, allowing employees and administrators to create, edit, submit, and review timesheets as well as generate PDF reports. The frontend interacts with the User Management Service for user login and authentication, with the Timesheet Management Service for managing timesheets, and with the Document Generator Service for generating PDF documents, using RESTful APIs for all calls. The backend is implemented using Spring Boot with Java and consists of multiple independent services, each responsible for a specific domain. The Timesheet Management Service handles the creation, editing, validation, and retrieval of timesheets, storing timesheet data in MongoDB to allow efficient document-based operations. It periodically calls the User Management Service via a scheduled task to retrieve the list of users for whom timesheets need to be generated and queries the Information Management Service to obtain pre-filled data from external university systems. The Document Generator Service generates

PDF reports using the OpenPDF library and retrieves the corresponding timesheet data from the Timesheet Management Service via Feign clients, persisting document generation logs in PostgreSQL. The User Management Service manages user authentication, authorization, and profile information, leveraging PostgreSQL to maintain relational consistency and secure storage. The Information Management Service acts as an integration layer with external university systems, using Feign clients to retrieve data related to project hours, teaching assignments, and other activities, transforming this information into a format compatible with the Timesheet Management Service data model and returning it to the requesting service. The Config Server centralizes configuration management, ensuring consistent and easily updatable settings across all services. Figure 4.1 is a diagram of the system with the different interactions between services and databases.

Security is implemented at multiple levels, starting with user authentication through Google OAuth, which enables employees and administrators to securely log in using their institutional accounts. Upon successful authentication, the User Management Service generates a JSON Web Token (JWT) that is sent to the user's browser as a cookie. This token is included in all subsequent requests to backend services, allowing each service to verify the user's identity and enforce role-based access control. As a result, users are permitted to perform only the operations corresponding to their assigned roles, ensuring secure and controlled access to all functionalities. Services communicate via RESTful APIs when accessed by the frontend and use Feign clients for type-safe inter-service calls and communication with external endpoints, maintaining modularity, separation of concerns, and seamless integration. The combination of MongoDB and PostgreSQL ensures both flexible document storage for timesheets and relational storage for users and document logs, providing data integrity, scalability, and maintainability. Section 4.3 describes all the services and the interaction between them.

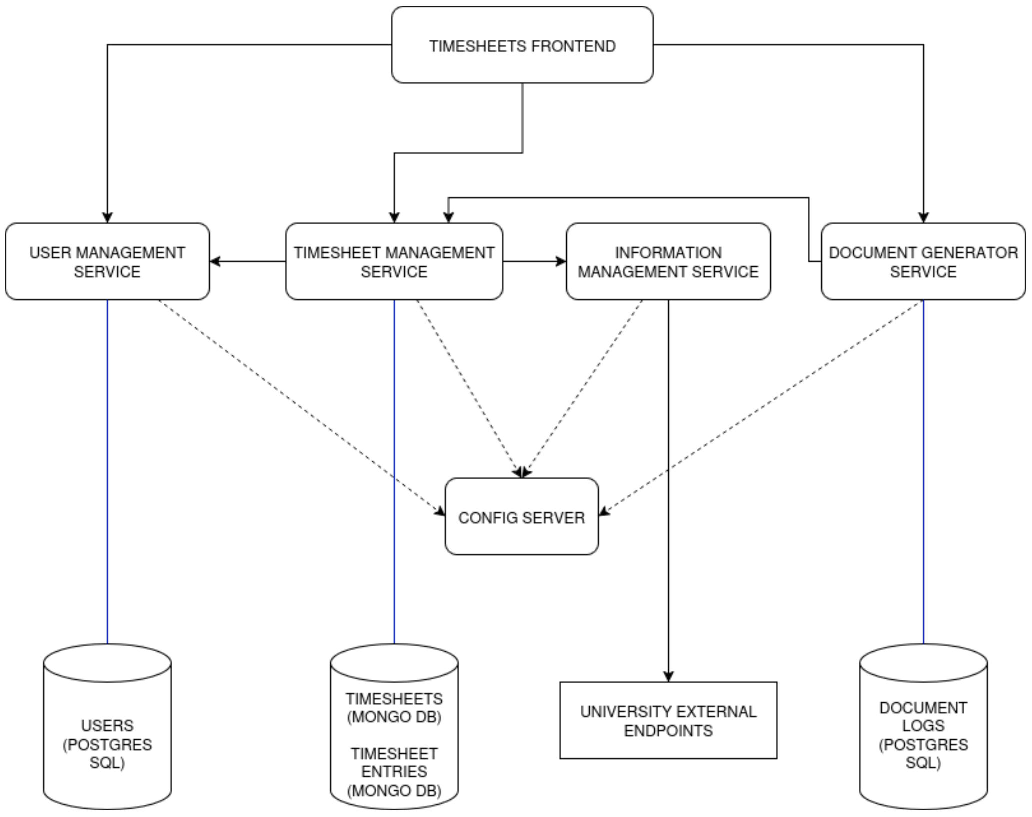

TIMESHEETS FRONTEND
USER MANAGEMENT SERVICE
TIMESHEET MANAGEMENT SERVICE
INFORMATION MANAGEMENT SERVICE
DOCUMENT GENERATOR SERVICE
CONFIG SERVER
USERS (POSTGRES SQL)
TIMESHEETS (MONGO DB)
TIMESHEET ENTRIES (MONGO DB)
UNIVERSITY EXTERNAL ENDPOINTS
DOCUMENT LOGS (POSTGRES SQL)

Figure 4.1: Overall system architecture

4.2 Technological Choices Justification

The technological decisions taken in the development of this platform were directly motivated by the functional and non-functional requirements identified in Chapter 2, as well as by the analysis of existing approaches in Chapter 3. The choice of a microservices architecture, implemented with Spring Boot and supported by containerization technologies such as Docker, ensures modularity, scalability, and maintainability, which are essential to address the growing demands of timesheet management in an academic context. According to Atlassian [5], microservices architectures are particularly advantageous for systems that require independent scaling, fault tolerance, and continuous delivery, all of which are critical for the context of this system since the number of users can escalate quickly and certain components, such as the timesheet management service, may require additional processing power. In contrast, a monolithic architecture, while simpler to implement initially, would pose significant challenges in terms of scalability and maintainability as the system evolves [40].

Spring Boot was selected as the backend framework due to its maturity, robust ecosystem, and seamless integration with Spring Cloud. It simplifies microservices development through features like auto-configuration, dependency injection, and RESTful API support, facilitating the creation of scalable and resilient architectures [41, 42]. The extensive community support and comprehensive documentation further ensure long-term sustainability [42].

Alternative frameworks such as Node.js and .NET Core were considered but found less suitable. Node.js, while flexible, poses security challenges in enterprise contexts due to issues like code injection, default cookie names, and cross-site scripting vulnerabilities [43]. Additionally, its ecosystem may require additional components for enterprise-level features such as security and data persistence [43].

.NET Core, though efficient, presents deployment complexities in Linux-based environments. While it supports Linux, deploying .NET Core applications on Linux can involve intricate configurations and dependencies, potentially increasing operational overhead [44].

Furthermore, prior experience with Spring Boot that helped significantly reducing the learning curve, accelerating the development process and enhancing overall productivity.

For the frontend, Angular was chosen because of its robustness and ability to support large-scale, enterprise-grade applications. Angular's strong typing with TypeScript, combined with its opinionated structure and two-way data binding, reduces development errors and ensures maintainability over time. Compared to React, which offers greater flexibility but requires the integration of third-party libraries for state management and routing, Angular provides a more complete framework out of the box, which accelerates development. Vue.js, while known for its simplicity, lacks the same level of adoption and enterprise support. For these reasons, Angular was considered the most suitable option for a system that must be both scalable and sustainable [15]. Similarly to the backend case, familiarity with Angular from previous projects contributed to a faster and more efficient development cycle, as common pitfalls and implementation patterns were already known.

With regard to data management, a polyglot persistence strategy was adopted, combining relational and non-relational databases. PostgreSQL was selected to handle user management and document generation logs, benefiting from strong ACID compliance and relational integrity [8]. While MySQL could also provide relational support, PostgreSQL offers superior extensibility and better support for advanced data types and indexing mechanisms, which are advantageous for complex queries[45]. MongoDB was employed for storing timesheet entries due to its flexibility in handling hierarchical, document-oriented data structures and its ability to scale horizontally [10]. Alternatives such as Cassandra or CouchDB were considered, but they lack the same balance between ease of development, community support, and integration with Java-based backends[46]. Prior experience with both PostgreSQL and MongoDB further supported this choice, enabling effective schema design and data modeling from the outset.

Security requirements demanded the use of OAuth 2.0 for authentication through institutional Google accounts, with JSON Web Tokens (JWT) used for secure communication across services. This choice ensures compliance with university standards and provides a reliable mechanism for enforcing role-based access control. Other approaches such as session-based authentication were considered, but they are less suitable in distributed microservices environments where stateless communication is preferred [47]. Here as well, practical familiarity with JWT-based authentication simplified its implementation, reducing the overall development time.

For continuous integration and deployment, GitLab pipelines were integrated with Docker registries, enabling automated builds, testing, and deployments. Docker was chosen as the containerization platform given its portability, lightweight design, and ecosystem support, which simplify the deployment of microservices across different environments [30]. While alternatives such as Podman or LXC exist, Docker remains the industry standard with the most comprehensive community and tool integration. Kubernetes was considered for orchestration [31], but given the limited infrastructure of the university environment, full orchestration was not strictly necessary for the scope of this dissertation. Previous work with Docker and GitLab CI/CD pipelines made their adoption particularly efficient, since both configuration and debugging processes benefited from existing expertise.

Finally, libraries such as OpenPDF were integrated to generate PDF reports directly from timesheet data, providing a reliable and open-source solution for document creation. Alternatives like iText and Apache PDFBox were considered, but OpenPDF was selected due to its open licensing and compatibility with the Java ecosystem used in the backend. In contrast, iText, while powerful, has licensing costs that would hinder its use in an academic context, and PDFBox, although free, is less actively maintained compared to OpenPDF [27, 28].

Altogether, these technological choices were carefully aligned with the system's goals: to deliver a scalable, secure, and maintainable platform that integrates seamlessly with the University of Évora's infrastructure while remaining adaptable to future requirements. At the same time, they reflect a pragmatic dimension of the project, since the prior knowledge of the chosen technologies not only accelerated the development process but also reduced risks associated with adopting unfamiliar tools. A comparative overview of the alternative technologies is presented in Table 4.1.

Table 4.1: Comparison of alternative technologies for the system

Component	**Chosen Technology**	**Alternatives**	**Justification**
Backend Framework	Spring Boot	Node.js, .NET Core	Mature ecosystem, strong support for microservices, seamless with Spring Cloud, prior experience accelerated development.
Frontend Framework	Angular	React, Vue.js	Full-featured framework, strong typing (TypeScript), better suited for large-scale apps; React requires more third-party libs, Vue has less enterprise adoption.
Databases	PostgreSQL + MongoDB	MySQL, Cassandra, CouchDB	PostgreSQL ensures relational integrity and ACID compliance; MongoDB flexible for timesheets; Cassandra/CouchDB less aligned with project needs.

Continued on next page

Table 4.1 (continued)

Component	**Chosen Technology**	**Alternatives**	**Justification**
Authentication	OAuth2 + JWT	Session-based auth	OAuth2/JWT is stateless and scalable for microservices; sessions unsuitable for distributed architecture.
Containerization	Docker	Podman, LXC	Industry standard, wide community support, integrates smoothly with GitLab CI/CD.
PDF Generation	OpenPDF	iText, PDFBox	Open-source and license-friendly; iText requires commercial license, PDFBox less actively maintained.

4.3 Microservices Structure

The system was designed following a microservices architectural style, in which independent services are responsible for specific business capabilities. This approach enables each service to evolve, scale, and be deployed independently, thereby increasing the platform's flexibility, maintainability, and resilience. Figure 4.2 illustrates the overall structure of the microservices and their interactions.

4.3.1 User Management Service

At the core of the platform is the *User Management Service*, which handles user authentication and authorization using OAuth2 and JWT tokens. This service integrates with the university's identity provider to enforce role-based access control for employees, managers, and administrators. It communicates with the frontend to manage JWT tokens and with the *Timesheet Service* once per month to generate pre-filled timesheets for all registered users. All authenticated users are persisted to allow automatic timesheet creation and maintain historical records for auditing purposes.

4.3.2 Timesheet Management Service

The *Timesheet Management Service* implements the main functionality of the platform, including the creation, editing, validation, and storage of timesheet records. Configurable validation rules ensure data integrity by preventing errors such as duplicate entries or negative working hours. This service interacts with the frontend for timesheet management, with the *User Management Service* to retrieve user information monthly, and with the *Information Management Service* to pre-fill timesheets automatically or upon user request. Additionally, it provides data to the *Document Generator Service* for report generation.

4.3.3 Document Generator Service

For reporting and documentation, the *Document Generator Service* was developed to produce official PDF documents. It retrieves data from the *Timesheet Service* and generates structured reports that can be archived or shared with other departments, supporting compliance and administrative needs, this service maintains a log of all generated documents and interacts with the frontend only when a document is requested, as well as with the *Timesheet Service* to obtain the necessary data for document creation.

4.3.4 Information Management Service

The *Information Management Service* acts as a bridge between the platform and external university endpoints. It transforms external data into a format compatible with the system to pre-fill timesheets, this service communicates exclusively with the *Timesheet Service*, enabling seamless integration with new external data sources whenever required, thereby isolating external dependencies and minimizing impact on core services.

4.3.5 Config Server

To manage configurations across all services, a centralized *Config Server* was employed. It stores configuration properties in a version-controlled repository and provides them dynamically to each microservice at startup or during runtime refresh. This approach ensures configuration consistency, simplifies maintenance, and enables the platform to adapt rapidly to environmental changes or new requirements without redeploying services.

Rather than developing a custom solution, the *Config Server* is a Spring Boot module that can be readily integrated into the system. Once configured to point to a version-controlled repository (e.g., Git), it automatically retrieves configuration properties and distributes them to each microservice at startup or during runtime refresh. This eliminates the need to implement a configuration management service from scratch [48].

This microservices architecture promotes modularity, fault tolerance, and extensibility. It facilitates future enhancements, such as integration with enterprise resource planning systems, addition of analytics services for workforce planning, or incorporation of event-driven mechanisms to further decouple services. Moreover, the architecture supports best practices including centralized logging, monitoring, and error handling, enhancing the overall observability and maintainability of the platform. The structure is visually represented in Figure 4.2.

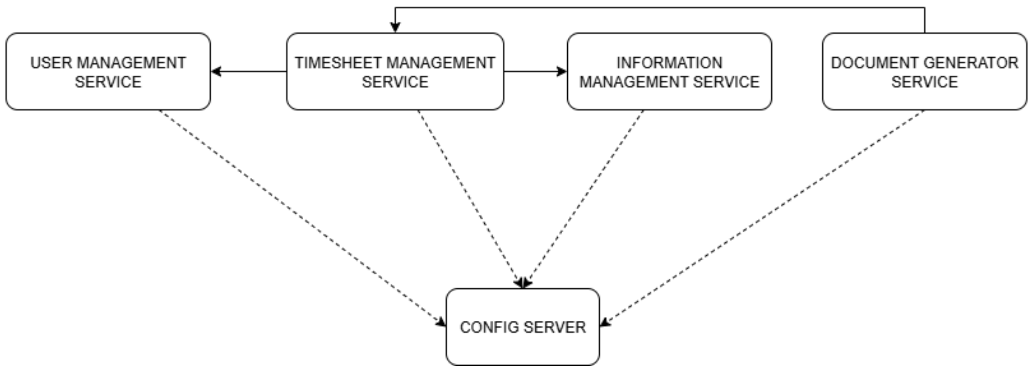

USER MANAGEMENT SERVICE
TIMESHEET MANAGEMENT SERVICE
INFORMATION MANAGEMENT SERVICE
DOCUMENT GENERATOR SERVICE
CONFIG SERVER

Figure 4.2: Overall architecture of the microservices platform, showing the main services and their interactions.

4.4 Frontend Integration

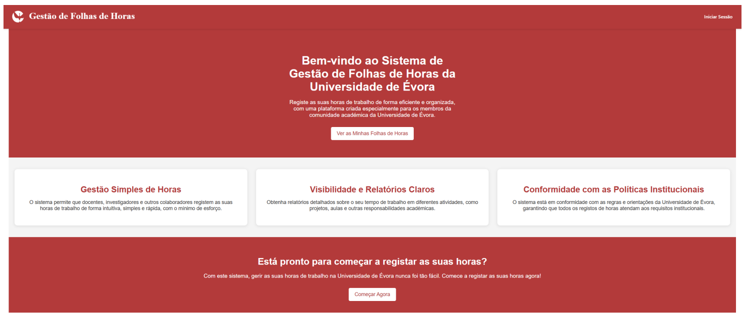
The frontend of the platform is implemented as an Angular application, providing a responsive and intuitive user interfaces. The application leverages TypeScript for strong typing, modular components, and two-way data binding, ensuring maintainability and reducing development errors.

The frontend interacts with multiple backend microservices to perform its core operations, Figure 4.4 illustrates the interaction between the frontend and backend services for the main flows of the system. The *Timesheet Management Service* is called to manage timesheet records, including creation, validation, submission, approval, and rejection of entries. The *Document Generator Service* is invoked to generate PDF reports for individual or aggregated timesheets. User authentication is handled through the *User Management Service*, which redirects users to Google OAuth for institutional login. Upon successful authentication, a JWT token is returned and stored in browser cookies. Every time the user attempts to access a protected section of the platform, the frontend calls the User Management Service to verify the session's validity, ensuring secure access control.

Many operations, such as client-side validations, page reloads, and visual error presentations, are performed entirely on the frontend to improve responsiveness and user experience. Angular services encapsulate HTTP requests, handle error responses, and provide a consistent API for frontend components. Interceptors automatically attach JWT tokens to requests and manage session expiration, further reinforcing secure communication. Validation is enforced on both the frontend and backend to prevent incorrect or inconsistent timesheet entries, providing immediate feedback to users through messages or highlighted fields.

The application is organized following a modular architecture, with separate modules and components for user management, timesheet operations, and report generation. This modularity promotes scalability, code reuse, and maintainability. Furthermore, the frontend ensures a seamless user experience through responsive layouts, interactive dashboards, and intuitive forms, allowing employees and administrators to efficiently manage timesheets, generate reports, and navigate the system securely.

Figure 4.3 presents the main user interfaces of the Timesheet Management Platform. It includes the home page, the timesheet form, user and administrator views. These screenshots illustrate the system's usability and visual coherence across different functionalities, highlighting the responsive design and intuitive navigation provided by the Angular frontend.


Gestão de Folhas de Horas

Iniciar Sessão

Bem-vindo ao Sistema de Gestão de Folhas de Horas da Universidade de Évora

Registe as suas horas de trabalho de forma eficiente e organizada, com uma plataforma criada especialmente para os membros da comunidade académica da Universidade de Évora.

Ver as Minhas Folhas de Horas

Gestão Simples de Horas

O sistema permite que docentes, investigadores e outros colaboradores registem as suas horas de trabalho de forma intuitiva, simples e rápida, com o mínimo de esforço.

Visibilidade e Relatórios Claros

Obtenha relatórios detalhados sobre o seu tempo de trabalho em diferentes atividades, como projetos, aulas e outras responsabilidades académicas.

Conformidade com as Políticas Institucionais

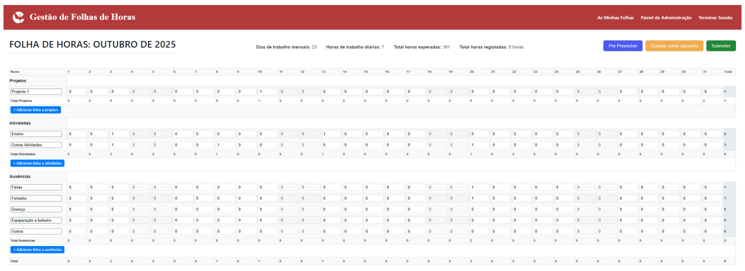
O sistema está em conformidade com as regras e orientações da Universidade de Évora, garantindo que todos os registos de horas atendam aos requisitos institucionais.

Está pronto para começar a registar as suas horas?

Com este sistema, gerir as suas horas de trabalho na Universidade de Évora nunca foi tão fácil. Comece a registar as suas horas agora!

Começar Agora

(a) Home screen


Gestão de Folhas de Horas

As Minhas Folhas Painel de Administração Terminar Sessão

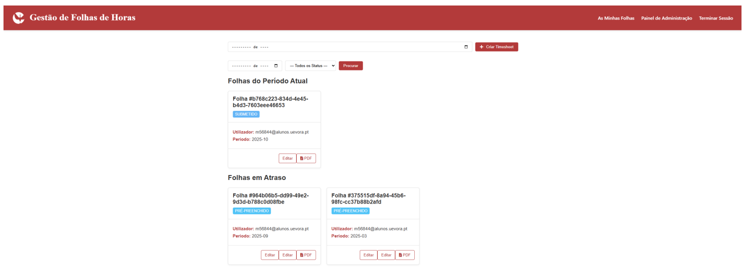
FOLHA DE HORAS: OUTUBRO DE 2025

Dias de trabalho mensais: 23 **Horas de trabalho diárias:** 7 **Total horas esperadas:** 161 **Total horas registadas:** 8 horas

Pré-Preencher Guardar como rascunho Submeter

Nome	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Total
Projetos																																
Projecto 1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Total Projetos	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
+ Adicionar linha a projetos																																
Atividades																																
Ensino	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
Outras Atividades	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	3
Total Atividades	0	0	2	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	5
+ Adicionar linha a atividades																																
Ausências																																
Férias	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
Feriados	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
Doença	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Equiparação a bolseiro	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Outros	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total Ausências	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	2
+ Adicionar linha a ausências																																
Total	0	0	2	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	8

(b) Timesheet form


Gestão de Folhas de Horas

As Minhas Folhas Painel de Administração Terminar Sessão

--------- de ---- + Criar Timesheet

--------- de ---- — Todos os Status — Procurar

Folhas do Período Atual

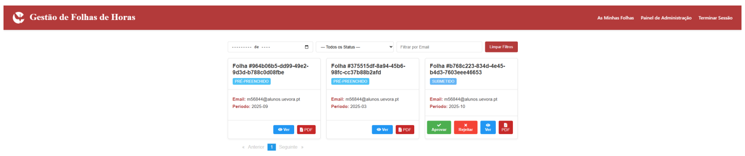
Folha #b768c223-834d-4e45-b4d3-7603eee46653
SUBMETIDO
Utilizador: m56844@alunos.uevora.pt
Período: 2025-10
Editar PDF

Folhas em Atraso

Folha #964b06b5-dd99-49e2-9d3d-b788c0d08fbe
PRÉ-PREENCHIDO
Utilizador: m56844@alunos.uevora.pt
Período: 2025-09
Editar Editar PDF

Folha #375515df-8a94-45b6-98fc-cc37b88b2afd
PRÉ-PREENCHIDO
Utilizador: m56844@alunos.uevora.pt
Período: 2025-03
Editar Editar PDF

(c) User dashboard


Gestão de Folhas de Horas

As Minhas Folhas Painel de Administração Terminar Sessão

--------- de ---- — Todos os Status — Filtrar por Email Limpar Filtros

Folha #964b06b5-dd99-49e2-9d3d-b788c0d08fbe
PRÉ-PREENCHIDO
Email: m56844@alunos.uevora.pt
Período: 2025-09
Ver PDF

Folha #375515df-8a94-45b6-98fc-cc37b88b2afd
PRÉ-PREENCHIDO
Email: m56844@alunos.uevora.pt
Período: 2025-03
Ver PDF

Folha #b768c223-834d-4e45-b4d3-7603eee46653
SUBMETIDO
Email: m56844@alunos.uevora.pt
Período: 2025-10
Aprovar Rejeitar Ver PDF

« Anterior 1 Seguinte »

(d) Admin management view

Figure 4.3: Frontend interface of the Timesheet Management Platform, illustrating the main user and administrative views.

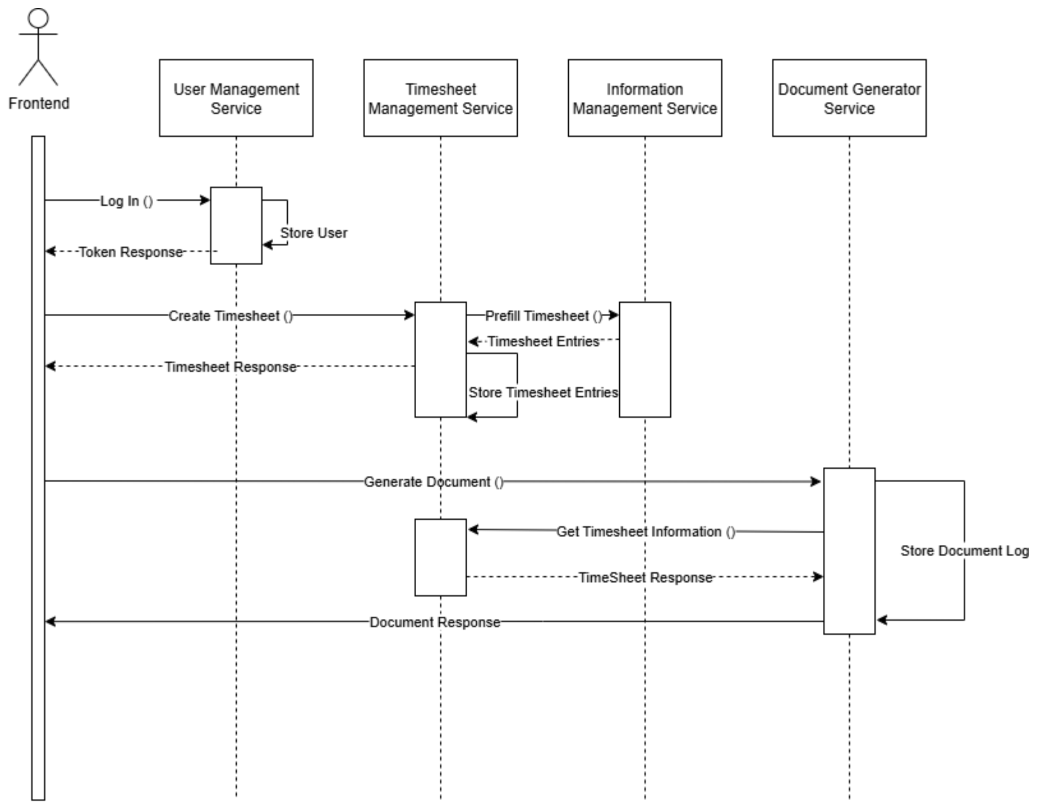


Figure 4.4: Interaction diagram between the Angular frontend and backend microservices, illustrating authentication, timesheet management, and PDF generation flows.

Chapter 5

Implementation

This chapter presents the overall architecture and implementation details of the Timesheet Management Platform. It describes the structure and responsibilities of each backend microservice, the organization of the frontend application, and the repository layout that supports modular development and continuous integration. The chapter also explains the platform's Dockerization and deployment strategy, emphasizing how containerization ensures consistency and reproducibility across environments. Finally, it discusses the main technical challenges encountered during development and the solutions adopted to achieve scalability, maintainability, and operational efficiency.

5.1 Main Modules

The platform is organized into several main modules, each responsible for specific business functionalities and designed to promote modularity, maintainability, and scalability. These modules include both backend microservices and frontend Angular components, as well as configuration management for the system.

5.1.1 Backend Structure

The backend consists of multiple independent microservices, each structured in a layered architecture to enforce separation of concerns and maintainability. Each service is internally divided into controllers, service layers, persistence layers, models, properties, and configuration components. Controllers handle incoming requests and route them to the appropriate service layer, where business logic is implemented. The persistence layer is responsible for interacting with the underlying database, either MongoDB or PostgreSQL depending on the service, ensuring data integrity and efficient operations. Models define the domain entities and data transfer objects, facilitating consistent data representation across layers and services. Properties and configuration files are used to externalize environment-specific settings, while the Config Server centralizes configuration management for all microservices, retrieving YAML files from a version-controlled repository and providing dynamic updates at startup or runtime refresh. This layered organization ensures that each service remains modular, testable, and easily maintainable.

User Management Service

The User Management Service is responsible for handling user authentication, authorization, and profile management, with a particular focus on integration with Google OAuth for institutional login. Internally, it uses a layered architecture to separate concerns between controllers, services, repositories, and security configuration, ensuring maintainability, testability, and clear responsibility boundaries.

The `GoogleOAuthController` orchestrates the OAuth flow, managing redirections to Google login, handling callback requests for authorization code exchanges, and managing refresh and logout operations. Access and refresh tokens, as well as the ID token, are stored in HTTP-only cookies, providing stateless session management across the platform. The controller relies on `AuthService` to interact with Google OAuth endpoints via `RestTemplate`, exchanging authorization codes for tokens, retrieving user information, and persisting or updating user profiles in the database using `UserRepository`. Internal JWTs are generated for authenticated sessions, including user ID, role, and email claims, with expiration tracking handled by `JwtUtil`.

Administrative user management is implemented in the `UsersAdminController`, which provides endpoints for creating, updating, deleting, and retrieving users. This controller enforces role-based access control, ensuring that only users with `ROLE_ADMIN` can perform management operations. Input validation and strict role checks are performed, and exceptions are raised using `ResponseStatusException` when requests are invalid or unauthorized. Business logic for user entities is handled by `UserService`, which maps between `UserEntity` and the domain model `User`, providing consistent CRUD operations while keeping persistence concerns within `UserRepository`.

Security is configured in `SecurityConfig`, which disables CSRF protection to support stateless sessions, sets up CORS filtering for allowed origins and HTTP methods, and integrates JWT-based authentication through `JwtAuthenticationFilter` and `JwtValidator`. Passwords are encoded using `BCryptPasswordEncoder`, and session state is entirely managed through JWTs, providing a secure and scalable solution for authentication.

The main endpoints exposed by the User Management Service include:

- **GET /auth/login** – Redirects the user to Google OAuth login.
- **GET /auth/callback** – Handles Google OAuth callback and token exchange, setting authentication cookies.
- **POST /auth/refresh** – Issues a new JWT using the provided refresh token.
- **POST /auth/logout** – Clears authentication cookies to log out the user.
- **GET /users/me** – Retrieves details of the currently authenticated user from JWT claims.
- **GET /users/{id}** – Retrieves a specific user by ID.
- **PUT /users/{id}** – Updates user information and roles.
- **DELETE /users/{id}** – Deletes a user entity.
- **GET /users** – Lists all users.
- **POST /users** – Creates a new user and persists it in the database.

Timesheet Management Service

The Timesheet Service implements the core functionality of the platform, providing a complete workflow for timesheet creation, editing, validation, submission, approval, and rejection. It is designed with a modular architecture where the `TimesheetController` handles incoming HTTP requests, delegating business logic to the `DefaultTimesheetService`, which in turn interacts with the persistence layer through `TimesheetRepository` and `TimesheetEntryRepository`. The service integrates with the User Management Service to retrieve authenticated user information and with the Information Management Service to pre-fill timesheet entries, ensuring that timesheets are accurate and consistent with historical and organizational data.

Timesheet entries are stored in MongoDB, allowing for flexible, document-oriented operations and efficient retrieval for both individual users and administrative overviews. Validation rules for timesheets and entries are enforced through the `TimesheetValidations` and `TimesheetEntryValidations` classes, ensuring data integrity before persisting or updating entries. Additionally, the service provides a monthly scheduler (`TimesheetScheduler`) that automatically generates and pre-fills timesheets for all users based on organizational data, reducing manual effort and ensuring compliance with reporting periods.

Security is enforced by checking the authenticated user against the timesheet's owner, with role-based access control ensuring that only administrators can approve or reject entries. The service provides detailed logging at every step, including timesheet submission, updates, pre-filling, and status changes, facilitating auditing and traceability.

The main endpoints exposed by the Timesheet Service include:

- **POST /timesheets** – Submit a new timesheet, validating entries and persisting them in the database.
- **GET /timesheets** – Retrieve timesheets based on user ID, period, or status, optionally generating blank entries if requested.
- **GET /timesheets/{id}** – Retrieve detailed information for a specific timesheet, ensuring the requesting user is authorized.
- **PUT /timesheets/{id}** – Update an existing timesheet, replacing previous entries while enforcing validation rules.
- **PUT /timesheets/{id}/status** – Approve or reject a timesheet, restricted to users with administrative privileges.
- **GET /timesheets/{id}/prefill** – Pre-fill a specific timesheet automatically with entries from the Information Management Service.
- **GET /timesheets/{period}/create-and-prefill** – Create and pre-fill a new timesheet for a specified period, removing any existing entries for the same period.

Document Generator Service

The Document Generator Service is responsible for producing PDF reports on-demand, primarily for timesheet data. It integrates directly with the Timesheet

Service through a Feign client (`TimesheetClient`) to retrieve timesheet information and relies on the OpenPDF library for PDF generation. Generated documents are logged in PostgreSQL using the `PdfDocumentsRepository`, ensuring traceability of all generated reports and supporting audit requirements.

At the core of the service, the `PDFController` exposes the HTTP endpoint for generating PDFs. It retrieves the authenticated user from the Spring Security context and uses the Feign client to fetch the corresponding timesheet data. The business logic for PDF generation is encapsulated in the `DefaultDocumentGeneratorService`, which constructs the document using `PdfHeaderBuilder` and `PdfTableBuilder`, adds headers and tables corresponding to the timesheet entries, and logs the result using `PdfLogger`. The PDF is returned as a Spring `Resource` with appropriate headers for file download, including a dynamically generated filename based on the user ID and timesheet period.

The service also implements error handling at multiple layers: if the user is not authenticated, the controller throws an `UNAUTHORIZED` exception; if PDF generation fails, a `PdfGenerationException` is thrown, and the failure is logged for auditing. The PDF output includes a custom header and footer with page numbers and user identifiers, ensuring clarity and traceability for each document.

The main endpoint exposed by the Document Generator Service is:

- **GET /timesheets/{timesheetId}/pdf** – Generates a PDF containing the timesheet entries for the specified timesheet ID, returning it as a downloadable file.

Information Management Service

The Information Management Service functions as an integration layer with external university systems, such as the SIIUE platform, isolating the rest of the platform from external dependencies and transforming incoming data into a format compatible with the Timesheet Service. Its primary purpose is to retrieve and process user-related information, such as projects and activity summaries, and convert them into structured timesheet entries for pre-filling purposes.

At the core of the service, the `InformationManagementController` exposes the endpoint for fetching timesheet entries. It ensures that the requesting user is authenticated using Spring Security, and delegates the actual data retrieval and transformation to the `DefaultInformationManagementService`. This service communicates with external systems via the `SiiueClient` Feign client, sending HTTP requests with user identifiers, month, and year as parameters.

The business logic in `DefaultInformationManagementService` aggregates data from two primary sources: projects and summaries. Project entries are transformed into daily timesheet entries by calculating the total monthly hours based on standard workdays and the project's dedication percentage, distributing both the integer and fractional hours across the available days. Summary entries, representing activities such as teaching or administrative work, are converted from minutes to hours with simple rounding logic. The service ensures that all entries are annotated with their source and type, distinguishing between project-based and activity-based records.

The service also handles errors gracefully: failures in fetching data from external systems are caught and logged, and empty or missing responses are replaced with default objects to maintain service reliability.

The main endpoint exposed by the Information Management Service is:

- **GET /information/timesheet-entries** – Retrieves timesheet entries for a user for a given month and year, combining data from projects and summaries into a unified format compatible with the Timesheet Service.

5.1.2 Frontend Structure

HomeComponent

The `HomeComponent` serves as the landing page, providing users with navigation to key sections of the application and quick access to dashboards.

NavbarComponent

The `NavbarComponent` implements the main navigation bar, allowing users to toggle the menu, log in, and log out, with authentication handled by the `AuthService`.

TimesheetListComponent

The `TimesheetListComponent` lists all timesheets available to the user, supporting filtering by period and status, pagination, and the creation of new timesheets. It categorizes timesheets into current, overdue, and other periods to give a clear overview. Users can also download timesheets in PDF format directly from this component.

TimesheetSearchResultsComponent

The `TimesheetSearchResultsComponent` displays the results of timesheet searches. It supports filtering by period and status, pagination, and uses the `TimesheetService` to retrieve filtered data from the backend.

TimesheetFormComponent

The `TimesheetFormComponent` allows creation and editing of timesheets. It dynamically manages sections for projects, activities, and absences, including default rows. The component handles real-time input validation, prevents invalid entries, calculates totals per row, per section, and per day, and interacts with backend services to save, draft, submit, or pre-fill timesheets. Feedback on actions is provided via `MatSnackBar` notifications, and loading states are indicated using `MatProgressSpinnerModule`.

AdminTimesheetManagementComponent

The `AdminTimesheetManagementComponent` enables administrators to view, filter, and manage timesheets from all users. It supports approving or rejecting timesheets with optional comments and allows downloading timesheets as PDFs. Filtering is applied directly through service calls to the backend, and loading states are reflected in the UI to improve user experience.

AdminTimesheetViewComponent

The `AdminTimesheetViewComponent` provides a detailed view of individual timesheets for administrators. It organizes entries into sections, calculates totals per row, section, and day, and allows status updates with client-side validation to prevent invalid submissions. All calculations and validations are handled on the frontend for responsiveness.

The frontend leverages Angular services such as `TimesheetService`, `DateUtilsService`, `TimesheetValidatorService`, and `TimesheetCalculatorService` to centralize business logic, perform calculations, validate data, and interact with the backend. This approach ensures consistency across components, reduces code duplication, and allows the UI to remain reactive and responsive. By combining modular components, reusable services, and strong typing, the frontend achieves a maintainable, scalable, and user-friendly structure that enhances the overall application experience.

5.2 Repository Structure and GitLab CI/CD Pipelines

The platform's source code is organized under a main group named *Timesheet Management UE*, which contains multiple repositories corresponding to individual services and frontend components. These repositories include the Config Server, Configurations, Document Generator Service, Information Management Service, Timesheet Management Service, Timesheet Management Frontend, and User Management Service. Each repository is structured to facilitate modular development, independent deployment, and maintainability. Figure 5.1 represents the group and repositories structure.

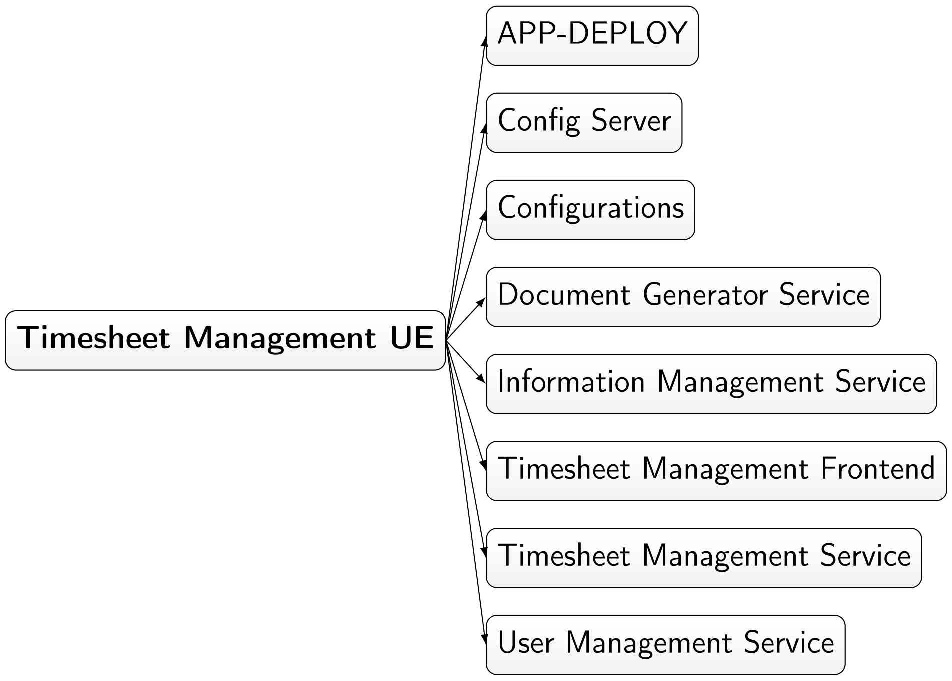

Timesheet Management UE
APP-DEPLOY
Config Server
Configurations
Document Generator Service
Information Management Service
Timesheet Management Frontend
Timesheet Management Service
User Management Service

Figure 5.1: GitLab group and repositories structure.

All backend services follow a consistent layered architecture and share a standardized GitLab CI/CD pipeline. The pipeline begins with the `build` stage, which compiles the Java project using Maven and ensures that dependencies are correctly resolved in a local cache. This is followed by the `test` stage, where automated unit and integration tests are executed to guarantee code quality. In the `package` stage, Maven generates the service JAR files, which are stored as artifacts for subsequent stages. The `docker-build-push` stage builds Docker images for the service, tags them with the commit SHA and the `latest` tag, and pushes them to a Docker registry. Finally, manual deployment stages `deploy-dev` and `deploy-prod` allow the deployment of development and production versions of each service. This uniform pipeline ensures consistency, reliability, and reproducibility across all backend services.

The frontend repository, corresponding to the Timesheet Management Angular application, also follows a dedicated CI/CD pipeline. The pipeline starts with the `install` stage, where Node.js dependencies are installed via `npm ci`, followed by a `lint` stage to enforce code style and quality. The `test` stage is included for future automated tests, while the `build` stage compiles the Angular application for production and stores the build artifacts. Similar to backend services, a `docker-build-push` stage builds and pushes Docker images, with manual deployment stages for development and production environments. This separation allows independent versioning and deployment of the frontend, while ensuring integration with backend services through environment-specific configurations.

The *Configurations* repository serves as a centralized location for all configuration files required by the platform's microservices. Each service retrieves its respective configuration properties from this repository via the Config Server, which ensures that environment-specific settings, such as database URLs, API endpoints, and feature flags, are consistently applied across development, testing, and production environments. By maintaining configurations in a version-controlled repository, the platform benefits from traceability, auditability, and the ability to roll back changes if necessary. This approach also simplifies the process of updating configurations, as modifications can be propagated dynamically to running services without the need for redeployment, improving maintainability and operational efficiency.

Additionally, there is a dedicated *App Deploy* repository, which contains the `docker-compose` configuration and orchestrates the deployment of all services. Its GitLab CI/CD pipeline consists of a single `deploy` stage that synchronizes the project files to the deployment environment using `rsync`, pulls the latest Docker images, and launches all services in detached mode. This approach centralizes deployment, simplifies environment management, and allows the team to deploy the entire platform with a single pipeline. Figure 5.2 visually represents the diferent steps of each pipeline.

Overall, this repository structure and standardized CI/CD configuration promote modularity, reproducibility, and scalability. Each service and the frontend can evolve independently, while the deployment process ensures consistent environments across development, testing, and production stages.

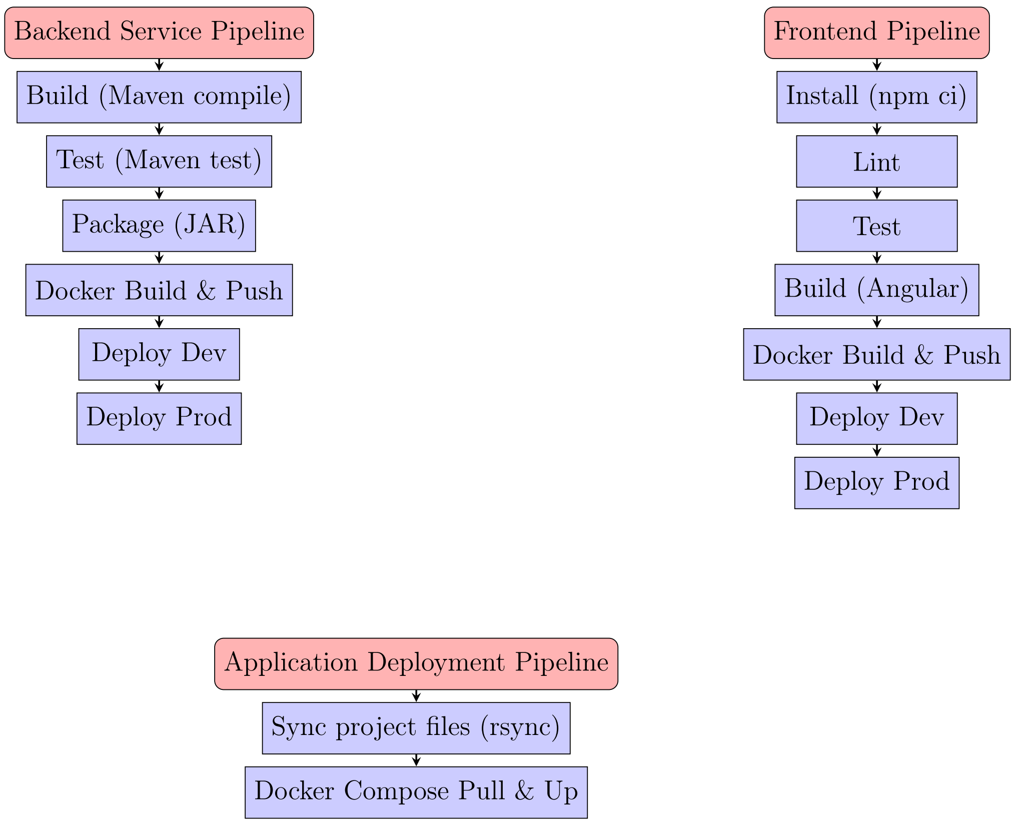


Figure 5.2: CI/CD pipelines overview for backend, frontend, and application deployment.

5.3 Dockerization and Deployment

The platform employs Docker to containerize both backend and frontend services, ensuring consistency, portability, and reproducibility across development, testing, and production environments. Each backend service is built using a multi-stage Dockerfile: the first stage uses a Maven image to compile and package the service into a JAR file, while the second stage employs a lightweight Java runtime image to execute the application. Health checks are configured for each backend container, allowing dependent services to start only after verifying that required services are operational and healthy.

The frontend Angular application follows a similar multi-stage Dockerization approach. Initially, a Node.js image is used to install dependencies and build the production-ready Angular application. The compiled output is then served using an Nginx image, providing optimized static content delivery. The Nginx configuration not only serves the Angular application efficiently but also acts as a reverse proxy for backend microservices. HTTP requests are automatically redirected to HTTPS, and SSL termination is handled using certificates obtained from Let's Encrypt. Requests directed to API endpoints, such as user management, timesheet, document generation, and information management services, are proxied to their respective containers, while appropriate headers and CORS policies ensure secure and seamless communication between frontend and backend components. Figure 5.3 visually represents the nginx proxy strategy.

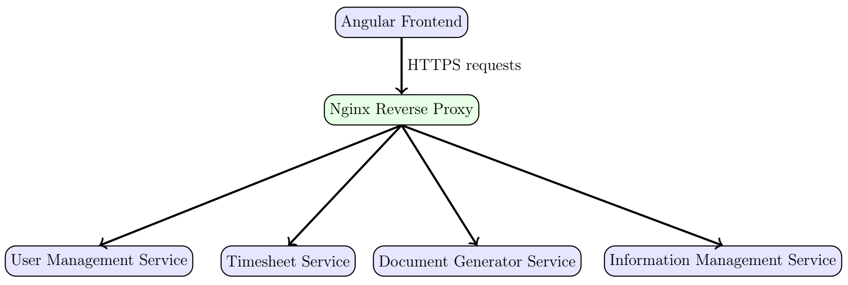


Figure 5.3: Nginx Reverse Proxy Architecture for the Timesheet Management Platform

All services, including backend, frontend, Config Server, MongoDB, and PostgreSQL, are orchestrated using Docker Compose. The Docker Compose configuration specifies service dependencies, environment variables, port mappings, persistent volumes, and health checks. The Config Server acts as a central configuration source, ensuring that all microservices dynamically load the required properties before startup. Database containers are configured with persistent volumes to maintain data durability and consistency across container restarts. Nginx depends on all application services and provides a single entry point, simplifying access, enabling secure connections, and centralizing request routing. This orchestration ensures that the entire platform can be consistently started, stopped, and managed in a controlled environment, reducing deployment errors and facilitating reproducible development and testing cycles.

5.4 Technical Challenges and Solutions

During the implementation of the platform, numerous technical challenges were encountered, spanning backend and frontend development, deployment processes, and system integration. Each of these challenges required careful consideration of alternatives and the adoption of solutions that ensured the scalability, maintainability, and reliability of the overall system.

A primary challenge involved the design of the editable timesheet, which featured a complex data structure and the need to dynamically manage user inputs. Initial attempts to implement flexible layouts resulted in inconsistencies and usability issues. This problem was ultimately addressed by developing a structured Angular form, which provided a more intuitive interface, enforced client-side validation, and facilitated seamless integration with backend services.

Deployment also presented significant difficulties. The initial approach involved transferring JAR files directly to the target environment via SSH; however, VPN restrictions rendered this method infeasible. The challenge was resolved by configuring GitLab CI/CD pipelines to build Docker images, push them to Docker Hub, and subsequently pull them into the deployment environment. This solution not only circumvented network limitations but also aligned with the platform's overarching containerization strategy.

Security constituted another critical concern. Google OAuth initially returned only basic user information, which proved insufficient for internal authorization requirements. To address this limitation, the platform was designed to transform OAuth responses into JSON Web Tokens (JWT), which were propagated across the system. This mechanism enabled a consistent and secure authentication and authorization framework for all services.

On the frontend, reusing logic across components posed challenges due to an initial lack of clear separation, resulting in code duplication and maintenance difficulties. This was mitigated by introducing an Angular service layer, responsible for encapsulating shared logic and managing API communication. Additionally, the need to handle environment-specific settings prompted the adoption of multiple configuration files, centrally managed via Spring Cloud Config Server, ensuring consistency and reducing configuration errors.

Validation rules required careful simplification. Early designs envisioned a large set of complex constraints, but this approach proved difficult to implement and maintain. Consequently, the validation layer was reduced to core business rules, such as enforcing a maximum number of hours per day and preventing duplicate project entries, striking a balance between robustness and maintainability.

Ensuring database consistency across services represented another challenge, particularly given the dual use of MongoDB and PostgreSQL. It was necessary to evaluate which database was most appropriate for each service, ensuring data integrity while avoiding unnecessary coupling. MongoDB was selected for timesheet records due to its document-oriented structure, whereas PostgreSQL was employed for structured logging and reporting, balancing flexibility with consistency.

Error handling and resilience were non-trivial concerns. Failures in one service occasionally propagated, causing cascading errors across the platform. This was mitigated by implementing fallback mechanisms and enhancing frontend error reporting, which provided users with clear feedback while preserving system stability.

Standardization of CI/CD pipelines was another priority. Initially, services followed heterogeneous build and deployment procedures, complicating maintenance. The adoption of template-based GitLab CI/CD pipelines resolved this issue, ensuring consistency across both backend and frontend services.

Defining an effective testing strategy also posed challenges. While unit, integration, and end-to-end (E2E) tests were considered, resource constraints necessitated a pragmatic approach. The final strategy combined unit tests with selected integration tests, achieving a balance between coverage, quality assurance, and development effort.

When enabling HTTPS, inter-service communication initially failed because containers did not recognize trusted certificates, leading to persistent connection errors. Various approaches were tested, including distributing certificates to each container, but these proved ineffective. The final solution involved generating Let's Encrypt certificates and configuring Nginx as a central reverse proxy to handle HTTPS termination and route requests to the appropriate services. This approach simplified certificate management, ensured secure communication, and adhered to security best practices.

Finally, logging and monitoring emerged as essential requirements for operational reliability. Establishing structured and centralized logs capable of supporting auditing and troubleshooting was challenging. A structured logging strategy was adopted, enabling traceability across microservices and providing valuable insights during both development and production phases.

Overall, addressing these challenges significantly enhanced the robustness of the platform. The adopted solutions not only ensured the system's functionality but also promoted long-term maintainability, scalability, and operational efficiency.

Chapter 6

Validation and Evaluation

This chapter presents the strategies and results of the validation and evaluation of the Timesheet Management Platform. It details the multi-level testing approach employed to ensure functional correctness, reliability, and usability across backend microservices and the Angular frontend. Additionally, the chapter discusses preliminary performance and scalability observations, as well as initial usability evaluations, highlighting both achievements and areas for future improvement. The goal is to provide a clear assessment of the platform's effectiveness and to establish a foundation for ongoing optimization and enhancement.

6.1 Testing Strategy

Ensuring the correctness and reliability of the platform required the adoption of a multi-level testing strategy, adapted to the specific requirements of backend microservices, the document generation service, and the Angular frontend. The approach combined automated unit and integration tests with functional and exploratory testing, while also integrating testing stages into the CI/CD pipelines to guarantee quality throughout the development lifecycle.

6.1.1 Unit Tests for Microservices

All backend microservices were tested using unit tests to validate business logic in isolation, `JUnit` framework was employed in combination with `Mockito` for mocking dependencies, which enabled precise validation of service-layer functionality without requiring database or external service interaction. Unit tests covered critical components such as authentication logic, validation rules for timesheets, and configuration handling, ensuring correctness of individual modules and facilitating refactoring.

This approach aligns with best practices for microservices unit testing [49].

6.1.2 Integration Tests for Microservices

Beyond unit testing, integration tests were implemented to verify the interaction between layers within each microservice, including the flow from controllers to repositories. Embedded databases, such as in-memory `H2` for relational services and embedded MongoDB instances for document-oriented services, were used to validate persistence operations and schema consistency. These integration tests ensured that application components behaved correctly in realistic scenarios and that database interactions conformed to the expected domain models [50].

6.1.3 Document Generator Service Testing

The Document Generator Service presented a unique challenge, since its output consisted of PDF documents rather than structured data. A pragmatic testing approach was adopted: documents were generated during the testing process and manually inspected to confirm correctness of structure, headers, page counts, and content placement. While automated validation was limited, minimal structural checks were performed (such as verifying the existence of a valid file and expected metadata). Future improvements could include baseline comparison methods, where generated documents are compared against predefined templates to allow automated verification.

6.1.4 Frontend Testing

Testing of the Angular frontend was primarily conducted through exploratory and functional testing during the development phase. By serving the application locally, it was possible to validate user interactions, navigation flows, and form submissions in real time. This approach ensured that client-side validation rules, such as preventing duplicate project entries and enforcing maximum daily work hours, were correctly applied and consistently enforced. Although comprehensive unit and end-to-end testing frameworks such as Jasmine, Karma, or Cypress were considered, they were not fully implemented due to time constraints. The strategy therefore focused on manual validation of user workflows, with the potential to extend coverage through automated UI testing in future iterations [51].

6.1.5 Continuous Integration and Delivery (CI/CD)

Automated testing was integrated into the GitLab CI/CD pipelines for backend services. Unit and integration tests were executed as part of the build process, ensuring that regressions or errors were identified prior to deployment. This practice enhanced confidence in the reliability of each release and provided rapid feedback during the development cycle [52]. Additionally, tools such as SonarQube were considered to enforce code quality metrics and maintain consistency throughout the development lifecycle [53].

CI/CD pipelines were also implemented for frontend services, incorporating linting and testing steps. Although unit tests have not yet been fully developed for the frontend, the pipelines establish a foundation for automated quality assurance and continuous validation as the application evolves.

6.2 Performance and Scalability Experiments

Although performance and scalability are critical aspects of modern web applications, formal experiments were not conducted during the current implementation of the platform. Instead, the platform's functionality was manually validated by generating multiple timesheets and performing repeated operations to observe system behavior under nominal conditions.

Performance and scalability testing are essential to ensure that a system can handle increasing workloads efficiently and maintain low latency under peak usage [54]. In microservices-based architectures, performance bottlenecks may arise at various points, including database access, inter-service communication, and frontend rendering. Similarly, the platform's scalability depends on the ability to replicate services, manage load balancing, and efficiently utilize resources [55].

Key performance metrics for microservices include response time, throughput, resource utilization, and service availability [56]. A comprehensive evaluation typically involves both load testing and stress testing, simulating realistic and peak workloads to identify potential bottlenecks and inform optimization strategies. Techniques such as horizontal scaling, caching mechanisms, and asynchronous communication can significantly improve system scalability and reliability.

For future iterations, it is planned to implement systematic performance and scalability experiments using automated testing frameworks. These experiments will allow quantitative measurement of system limits and optimization of microservice deployment strategies, including database replication and load balancing, additionally, integrating performance monitoring tools will provide continuous insight into system behavior, ensuring that the platform can meet real-world demands as usage grows.

In summary, while initial manual testing confirmed functional correctness, formal performance and scalability evaluation remains an essential task for future work to guarantee the platform's efficiency, robustness, and ability to handle increasing workloads.

6.3 Usability Evaluation

Usability evaluation is a fundamental aspect of software development, ensuring that a platform is intuitive, efficient, and satisfying for end users [57]. In this project, usability considerations were incorporated throughout both the design and implementation phases, with the objective of delivering a user-friendly interface tailored to the needs of administrative staff and students.

6.3.1 Client Sessions and Preliminary Evaluation

Preliminary evaluation was conducted through interactive sessions with a representative client, during which users saw the platform and the respective features. These sessions provided valuable qualitative insights into potential improvements, including enhancements to interface layout, navigation clarity, and immediate validation feedback.

6.3.2 Design Considerations for Usability

The frontend architecture, implemented using Angular, was designed with modular components to facilitate navigation and task completion. Client-side validation ensured that user input errors, such as duplicate entries or exceeding permitted working hours, were promptly highlighted, thereby increasing efficiency and reducing mistakes. Additionally, consistent navigation structures and uniform styling across components contributed to a more coherent and learnable user experience, supporting confidence and satisfaction during interaction with the system.

6.3.3 Limitations and Future Work

Despite the benefits gained from client sessions, the evaluation was limited in scope and did not employ formal usability testing techniques. Comprehensive studies, such as controlled user experiments, surveys, and task-based assessments, could provide quantitative measures of task completion time, error rates, and overall user satisfaction [58]. Furthermore, future work could explore alternative interface designs and ensure compliance with accessibility standards, thereby improving inclusivity and usability for a wider range of users.

In conclusion, while the preliminary evaluation with the client informed several design improvements, more systematic usability assessments are required to robustly evaluate the platform's effectiveness and guide further enhancements.

Chapter 7

Discussion

This chapter presents a critical analysis of the results obtained from the implementation of the microservices-based timesheet management platform. Its main goal is to evaluate how well the dissertation objectives were achieved, identify system limitations, and suggest potential improvements and directions for future work. The discussion covers multiple perspectives, including architecture and backend functionality, performance and service integration, frontend usability, security, and DevOps practices. It aims to highlight both the benefits of the system—such as modularity, scalability, and automation—and the trade-offs, operational challenges, and opportunities for further development.

7.1 Critical Analysis of Results

The evaluation of the proposed microservices-based timesheet management platform demonstrates that the primary objectives of the dissertation were successfully achieved. The system replaced the previous spreadsheet-driven process with a modular, scalable, and maintainable solution that supports secure user authentication, automated validations, and real-time reporting.

Nevertheless, the results also highlight important considerations and trade-offs. First, while independent service deployment improved scalability and fault tolerance, it introduced additional operational complexity. Service discovery, monitoring, and inter-service communication required careful configuration of Spring Cloud components and consistent use of Feign clients; without automated observability and detailed logging, debugging distributed failures can become time-consuming. Maintaining such a system requires a person with knowledge in software development, infrastructure, DevOps, and CI/CD practices. Additionally, potential costs associated with infrastructure (GitLab, Docker Hub), especially when using the free tier, may arise as the system scales.

Second, the dual-database approach (PostgreSQL for relational data and MongoDB for document storage) provided flexibility and performance, but increased maintenance overhead. Schema evolution, backup strategies, and data consistency between services demand strict governance to avoid long-term technical debt. Future iterations could explore unified data-access abstractions or improved schema-management tooling to reduce this burden.

From a frontend perspective, the Angular application achieved the intended usability goals, offering clear workflows for employees and administrators. Nonetheless, usability feedback indicated opportunities for further refinement, including more responsive interfaces, enhanced accessibility features, additional contextual information, and potential new features suggested by users. Moreover, the platform currently lacks integration with certain university endpoints, such as holidays, absences, or other administrative data. Given the microservices architecture, such integrations would be straightforward to implement in future releases.

Security mechanisms—OAuth 2.0 authentication and JWT-based inter-service communication—proved effective during testing. Continuous monitoring and periodic penetration testing will be essential to mitigate emerging threats and ensure compliance with university data-protection policies.

Overall, the experimental results confirm that a microservices architecture, supported by Spring Boot, Docker, and GitLab CI/CD, is appropriate for an institutional timesheet management system. The platform delivers measurable gains in automation, reliability, and scalability. The identified limitations, however, highlight areas for future improvement, including enhanced usability, expanded feature set, integration with additional university services, and streamlined maintenance processes, providing a clear roadmap for subsequent development.

7.2 System Limitations and Potential Improvements

Although the implementation of the Timesheet Management Platform successfully met the defined functional and non-functional objectives, several limitations were identified throughout the development and validation process. These constraints are mainly related to the scope of testing, infrastructure limitations, external dependencies, and the inherent complexity of a microservices-based architecture.

From an architectural perspective, the adoption of a microservices approach introduced additional operational and configuration challenges. While modularity and scalability were achieved, maintaining multiple independent services required continuous synchronization between repositories and configuration files. The reliance on a central Config Server and externalized YAML configurations, although beneficial for flexibility, occasionally caused deployment synchronization issues during pipeline executions. Furthermore, the absence of a centralized orchestration layer such as Kubernetes restricted the system's ability to perform automatic scaling and self-healing in the event of failures.

The implementation of continuous integration and continuous deployment (CI/CD) pipelines in GitLab significantly improved automation and delivery reliability; however, the current setup still presents some operational limitations. The use of GitLab's free tier introduces execution time restrictions, which may become a bottleneck as the project evolves and pipelines grow in complexity. A transition to a paid GitLab subscription or an internally hosted solution with dedicated runners would mitigate this limitation. Additionally, the absence of a private artifact repository, such as Sonatype Nexus or GitLab Container Registry with extended retention, constrains the long-term management of Docker images and build artifacts, potentially affecting traceability and version control.

In terms of backend functionality, although the core services operate reliably, some external integrations remain incomplete. For instance, the planned integration with endpoints related to vacation or absence data from the university's information systems has not yet been implemented. These endpoints are essential to ensure that timesheets automatically account for non-working periods, thereby improving data accuracy. Nonetheless, given the modular nature of the microservices architecture, the integration of these additional endpoints can be implemented without significant architectural refactoring, as each external connection can be encapsulated within its dedicated service.

Testing revealed that the pre-filling of timesheets introduces a noticeable delay of several seconds in some cases. This latency primarily originates from the response time of the external university endpoints accessed by the Information Management Service. Although the internal processing time of the platform is relatively low, the dependency on external data sources remains a critical factor affecting overall responsiveness. Implementing caching or asynchronous preloading strategies could mitigate this limitation in future iterations.

From a testing perspective, the validation strategy included unit and integration tests across all services; however, not all potential edge cases or concurrent access scenarios were covered. Inter-service communication, particularly through Feign clients, was verified under controlled conditions but not under sustained concurrent load or high-volume data exchanges. Similarly, error-handling mechanisms for network timeouts and partial service unavailability were tested only in simulated conditions, which may not fully reflect production behaviour.

On the frontend side, the Angular application achieved a high level of usability and responsiveness; nevertheless, formal usability testing was limited to preliminary sessions with a small number of participants. Quantitative usability metrics, such as task completion times or user satisfaction scores, were not systematically collected. Moreover, automated end-to-end testing with frameworks has not yet been integrated into the CI/CD pipeline, which limits regression detection in future deployments.

Finally, the dependency on external authentication through Google OAuth, while beneficial for integration and user convenience, introduces potential availability risks. Any temporary unavailability or latency in the OAuth service directly impacts user access to the platform. The absence of redundant authentication mechanisms further increases this dependency.

In summary, while the Timesheet Management Platform demonstrates robustness and reliability within its current deployment scope, these limitations highlight the need for further infrastructure improvements, extended testing coverage, and enhanced monitoring. The pending integrations with additional endpoints, improvements in pipeline infrastructure, and optimization of response times represent natural next steps toward ensuring the platform's long-term scalability, performance, and operational resilience.

Chapter 8

Conclusions and Future Work

This chapter presents the final reflections of this dissertation, highlighting the main contributions, evaluating the extent to which the initial objectives were achieved, and proposing directions for future development. The discussion synthesizes the technical, organizational, and academic outcomes of the research, emphasizing the impact of the implemented microservices-based timesheet management platform on both the operational efficiency of the University of Évora and the field of software engineering. By revisiting the objectives and summarizing the contributions, this chapter provides a clear assessment of the work accomplished, while the proposed future work outlines opportunities to further enhance performance, scalability, usability, and system integration.

8.1 Summary of Contributions

This dissertation advances both the academic field of software engineering and the operational efficiency of the University of Évora by providing a practical solution to the limitations of manual, spreadsheet-based timesheet management. The research demonstrates the successful implementation of a microservices-based platform that offers modularity, scalability, and maintainability, serving as a reference model for institutional applications.

The work presents a validated architectural approach that decouples core business domains and integrates a polyglot persistence strategy, illustrating how contemporary software engineering practices can be applied to institutional administrative systems. Moreover, the platform introduces a fully digitalized, auditable workflow, reducing errors and administrative overhead, while enabling real-time monitoring and reporting. The adoption of a modern technology stack and DevOps practices provides a documented foundation for future system extensions and integrations.

Beyond technical achievements, the dissertation emphasizes best practices in system design, including automated validation mechanisms, modularity, and user-centered interfaces. The contributions provide both a tangible tool for the University and an academic case study on the application of microservices in administrative contexts, highlighting opportunities for further enhancements in performance, usability, and system integration.

8.2 Objectives Revisited

The primary goal of this dissertation was to develop a digital timesheet management platform capable of addressing the inefficiencies and errors inherent in spreadsheet-based processes. This included improving data accuracy, reducing administrative workload, and providing mechanisms for real-time monitoring and reporting.

Upon reflection, the study demonstrates that these objectives have been largely achieved. The platform delivers a modular architecture that enhances resilience and maintainability, alongside mechanisms that ensure reliable data entry and validation. The usability goals were addressed through an accessible and intuitive frontend, while operational robustness was supported by CI/CD pipelines and containerized deployment.

8.3 Future Work Proposals

While the Timesheet Management Platform has demonstrated its effectiveness in replacing manual spreadsheet-based processes with a scalable, modular, and maintainable solution, several opportunities for future work remain. A primary area for development involves the systematic evaluation of performance and scalability under realistic workloads. Implementing automated stress and load testing frameworks would provide quantitative metrics on system responsiveness, resource utilization, and service availability, enabling informed optimization of the microservices deployment, database replication, and load balancing strategies.

Another avenue for future work concerns the integration of additional institutional endpoints, such as leave management, absence tracking, and other administrative services. These integrations would further enhance data accuracy, reduce manual intervention, and consolidate organizational information within a single, unified platform. Expanding the API Gateway and service architecture to accommodate new modules would be facilitated by the modular microservices design, allowing the system to evolve without major architectural refactoring.

Enhancing usability and accessibility represents a further area of improvement. Comprehensive user studies, including task-based assessments, surveys, and controlled experiments, would provide quantitative insights into interface effectiveness, task completion times, and user satisfaction. Additionally, exploring alternative interface designs and ensuring compliance with accessibility standards would broaden the platform's usability for diverse user groups.

From a technical perspective, the introduction of advanced monitoring and observability tools would support proactive system maintenance and early detection of failures. Coupled with extended automated testing, including end-to-end UI tests, these measures would increase reliability and reduce operational overhead. Finally, investigating container orchestration solutions, such as Kubernetes, could enable automatic scaling, self-healing, and more efficient resource management, further reinforcing the platform's resilience and operational robustness.

Collectively, these future work directions provide a roadmap for the continuous improvement of the platform, ensuring its adaptability to evolving organizational requirements while maintaining high standards of reliability, usability, and performance.

Bibliography

[1] Time tracking software. Available at: https://www.jibble.io [Accessed: 2024-06-30].

[2] Acadly features overview. Available at: https://www.acadly.com [Accessed: 2024-06-30].

[3] Student information systems. Available at: https://www.getalma.com [Accessed: 2024-06-30].

[4] Briohr solutions. Available at: https://www.briohr.com [Accessed: 2024-06-30].

[5] Atlassian. (2024) Microservices vs monolithic architecture. Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith [Accessed: 2024-04-30].

[6] Pivotal Software. (2021) Spring boot. Available at: https://spring.io/projects/spring-boot [Accessed: 2024-04-30].

[7] ——. (2021) Spring cloud. Available at: https://spring.io/projects/spring-cloud [Accessed: 2024-04-30].

[8] E. F. Codd. (1970) A relational model of data for large shared data banks. [Accessed: 2024-04-30].

[9] M. Stonebraker and J. M. Hellerstein, "What goes around comes around," in *Readings in Database Systems, 4th Edition*, 2005, available at: https://doi.org/10.1145/3685980.3685984 [Accessed: 2024-04-30].

[10] MongoDB Documentation. What is mongodb? Available at: https://www.mongodb.com/docs [Accessed: 2024-04-30].

[11] Couchbase Documentation. Introduction to couchbase. Available at: https://www.couchbase.com/resources [Accessed: 2024-04-30].

[12] Redis Documentation. Introduction to redis. Available at: https://redis.io/docs [Accessed: 2024-04-30].

[13] Amazon Web Services. Amazon neptune documentation. Available at: https://aws.amazon.com/pt/neptune/ [Accessed: 2024-04-30].

[14] React documentation. Available at: https://react.dev/ [Accessed: 2024-04-30].

[15] Angular documentation. Available at: https://angular.io/ [Accessed: 2024-04-30].

[16] Vue.js documentation. Available at: https://vuejs.org/ [Accessed: 2024-04-30].

[17] GeeksforGeek. (2025) Rest api introduction. Available at: https://www.geeksforgeeks.org/node-js/rest-api-introduction/ [Accessed: 2025-10-13].

[18] GeeksforGeeks. (2025) Basics of soap - simple object access protocol. Available at: https://www.geeksforgeeks.org/computer-networks/basics-of-soap-simple-object-access-protocol/ [Accessed: 2025-10-13].

[19] ——. (2025) Graphql tutorial. Available at: https://www.geeksforgeeks.org/graphql/graphql-tutorial/ [Accessed: 2025-10-13].

[20] Red Hat. (2024) Rest e soap: entenda as diferenças. Available at: https://doi.org/10.5539/mas.v12n3p175 [Accessed: 2024-04-30].

[21] Spring Team. (2025) Spring cloud openfeign documentation. Available at: https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/ [Accessed: 2025-10-02].

[22] ——. (2025) Spring webclient documentation. Available at: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-client [Accessed: 2025-10-02].

[23] Apache. (2025) Apache kafka documentation. Available at: https://kafka.apache.org/documentation/ [Accessed: 2025-10-02].

[24] Rabbit MQ. (2025) Rabbit mq documentation. Available at: https://www.rabbitmq.com/docs [Accessed: 2025-10-02].

[25] Filesaver.js documentation. Available at: https://github.com/eligrey/FileSaver.js [Accessed: 2024-04-30].

[26] Xlsx.js documentation. Available at: https://github.com/SheetJS/sheetjs [Accessed: 2024-04-30].

[27] iText Software. (2021) itext pdf library. Available at: https://itextpdf.com/ [Accessed: 2024-04-30].

[28] Apache Software Foundation. (2021) Apache pdfbox. Available at: https://pdfbox.apache.org/ [Accessed: 2024-04-30].

[29] LibrePDF Community. (2025) Openpdf: Open source java library for pdf generation and manipulation. Available at: https://github.com/LibrePDF/OpenPDF [Accessed: 2025-10-01].

[30] Docker, Inc. (2021) Docker. Available at: https://www.docker.com [Accessed: 2024-04-30].

[31] Cloud Native Computing Foundation. (2021) Kubernetes. Available at: https://kubernetes.io [Accessed: 2024-04-30].

[32] A. Mahida. (2021) A review on continuous integration and continuous deployment (ci/cd) for machine learning. Available at: https://www.researchgate.net/profile/Ankur-Mahida-2/publication/379875894_A_Review_on_Continuous_Integration_and_Continuous_Deployment_CICD_for_Machine_Learning/links/6621b5e143f8df018d198ee9/A-Review-on-Continuous-Integration-and-Continuous-Deployment-CI-CD-for-Machine-Learnin
pdf [Accessed: 2025-09-30].

[33] M. S. Arefeen and M. Schiller. (2019) Continuous integration using gitlab. Available at: https://doi.org/10.26685/urncst.152 [Accessed: 2025-09-30].

[34] European Union. (2016) General data protection regulation (gdpr) — regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016. Available at: https://eur-lex.europa.eu/eli/reg/2016/679/oj [Accessed: 2025-10-01].

[35] Universidade de Évora. (2025) Política de privacidade e proteção de dados da universidade de Évora. Available at: https://www.uevora.pt/termos-e-condicoes-e-politica-de-privacidade/politica-de-privacidade [Accessed: 2025-10-01].

[36] F. Dalpiaz and A. Sturm. (2020) Conceptualizing requirements using user stories and use cases: A controlled experiment. Available at: https://doi.org/10.1007/978-3-030-44429-7$_1$6[$Accessed : 2024 - 09 - 19$].

[37] V. P. Guides. (n.d.) User stories and use cases: A comprehensive guide to agile development. Available at: https://guides.visual-paradigm.com/user-stories-and-user-cases-a-comprehensive-guide-to-agile-development/ [Accessed: 2024-09-19].

[38] A. Alliance. (n.d.) What are user stories? Available at: https://www.agilealliance.org/glossary/user-stories/ [Accessed: 2024-09-19].

[39] Y. Francino. (2022) What is a user story? definition from whatis.com. Available at: https://www.techtarget.com/searchsoftwarequality/definition/user-story [Accessed: 2024-09-19].

[40] O. Al-Debagy and D. Martinek. (2020) From monolithic systems to microservices: A comparative study of performance and maintainability for different architectural styles. Available at: https://doi.org/10.3390/app10175797 [Accessed: 2025-09-10].

[41] Stackify. (2024) What is spring boot? Available at: https://stackify.com/what-is-spring-boot/ [Accessed: 2025-10-02].

[42] S. B. Documentation. (2025) Spring boot documentation overview. Available at: https://docs.spring.io/spring-boot/documentation.html [Accessed: 2025-10-02].

[43] A. Security. (2023) Node.js security: Top risks and 5 critical best practices. Available at: https://www.aquasec.com/cloud-native-academy/application-security/node-js-security/ [Accessed: 2025-10-02].

[44] Microsoft. (2025) Net application publishing overview. Available at: https://learn.microsoft.com/en-us/dotnet/core/deploying/ [Accessed: 2025-10-02].

[45] S. Tech. (2024) Mysql vs postgresql: A comprehensive comparison. Available at: https://qiita.com/sourabhtech/items/69d728172ad82c8490a7 [Accessed: 2025-10-02].

[46] G. Baruffa, M. Femminella, M. Pergolesi, and G. Reali. (2020) Comparison of mongodb and cassandra databases for spectrum monitoring as-a-service. Accessed: 2025-10-02.

[47] A. Bucko, K. Vishi, B. Krasniqi, and B. Rexha. (2023) Enhancing jwt authentication and authorization in web applications based on user behavior history. Available at: https://doi.org/10.3390/computers12040078 [Accessed: 2025-09-10].

[48] P. Software. (2025) Spring cloud config reference documentation. Available at: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/ [Accessed: 2025-10-02].

[49] M. Waseem, P. Liang, G. Márquez, and A. D. Salle. (2020) Testing microservices architecture-based applications: A systematic mapping study. Available at: https://doi.org/10.1109/APSEC51365.2020.00020 [Accessed: 2025-09-18].

[50] I. Ghani, W. M. Wan-Kadir, A. Mustafa, and M. Imran Babir. (2019) Microservice testing approaches: A systematic literature review. Available at: https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/3856 [Accessed: 2025-09-18].

[51] J. Palmer, C. Cohn, M. Giambalvo, and C. Nishina. (2018) Testing angular applications. USA. Available at: https://www.manning.com/books/testing-angular-applications [Accessed: 2025-09-18].

[52] GitLab. (2021) Ci/cd best practices for microservices. Available at: https://docs.gitlab.com/ee/ci/ [Accessed: 2025-09-18].

[53] SonarSource. (2025) Sonarqube: Continuous code quality inspection. Available at: https://www.sonarsource.com/products/sonarqube/ [Accessed: 2025-09-21].

[54] L. S. Iyer, B. Gupta, and N. Johri. (2005) Performance, scalability and reliability issues in web applications. Available at: https://doi.org/10.1108/02635570510599959 [Accessed: 2025-09-21].

[55] A. S. Shethiya. (2025) Scalability and performance optimization in web application development. Available at: https://ijstpublication.com/index.php/ijst/article/view/1 [Accessed: 2025-09-21].

[56] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner. (2018) Investigating performance metrics for scaling microservices in cloudiot-environments. Available at: https://doi.org/10.1145/3184407.3184428 [Accessed: 2025-09-21].

[57] B. E. John. (1996) Evaluating usability evaluation techniques. Available at: https://dl.acm.org/doi/10.1145/242224.242233 [Accessed: 2025-09-23].

[58] E. Insfran and A. Fernandez. (2008) A systematic review of usability evaluation in web development. Springer. Available at: https://link.springer.com/chapter/10.1007/978-3-540-85200-1_10 [Accessed: 2025-09-23].

UNIVERSIDADE DE ÉVORA
ESCOLA DE CIÊNCIAS E TECNOLOGIA