Politecnico di Torino

Master's Degree in Computer Engineering

A.y. 2023/2024

Graduation session December 2024

Design and implementation of an integrated DevOps framework for Digital Twins as a Service software platform

Supervisors:

Luigi De Russis

Peter Gorm Larsen

Co-supervisor:

Prasad Talasila

Candidate:

Vanessa Scherma

Abstract

Digital twins are virtual representations of physical entities. They are an emerging technology gaining increasing importance in several domains thanks to their ability to simulate and understand their physical counterpart, consequently improving decision-making processes. Digital Twin as a Service is a software platform developed by the INTO-CPS Association at Aarhus University, which provides the ability to build, use, and share digital twins with other users. The work presented in this thesis was carried out in collaboration with Aarhus University. Its main objective was to design and implement a DevOps framework for managing digital twins lifecycles, integrating GitLab APIs and CI/CD pipelines. The development of the framework was followed by the implementation of user interfaces in order to integrate the work with the Digital Twin as a Service web application. The proposed solution has been evaluated and tested, demonstrating positive results in terms of ease of use, simplifying digital twins management operations, and improving user interaction.

Acknowledgements

My graditude goes to Prof. Luigi De Russis, for his helpfulness throughout this journey.

This work would not have been possible without the guidance, support and patience of Prof. Peter Gorm Larsen and the insightful advices and practical assistance of Dr. Prasad Talasila. Thank you for giving me this enriching opportunity.

Thanks to my parents and my sister, who have always been there for me, supporting me during these years of study.

Thanks to my partner Simone, for the unwavering love he has always shown me and for standing by my side even during the most difficult times.

I am deeply grateful to my family for their boundless trust and to everyone who has shown me their closeness. A special thanks to my grandparents, for their constant care.

Thanks to Turin, the city that allowed me to grow and mature, for the first time away from home.

Finally, thanks to Aarhus, my new home, for welcoming me in this new chapter of my life.

Table of Contents

List of Tables

List of Figures

Acronyms

AAA

Arrange, Act, Assert

API

Application Programming Interface

ASDP

Agile Software Development Process

AAS

Asset Administration Shell

AWS

Amazon Web Services

CD

Continuous Deployment

CI

Continuous Integration

CPS

Cyber-Physical System

CRUD

Create, Read, Update, Delete

DT

Digital Twins

DTaaS

Digital Twins as a Service

E2E

End-to-End

GUI

Graphical User Interface

HCI

Human Computer Interaction

REST

Representational State Transfer

SUT

System Under Test

UAT

User Acceptance Testing

Chapter 1

Introduction

1.1 Overview

The Fourth Industrial Revolution is accelerating the digitalization process. The virtual and physical worlds are becoming increasingly connected and integrated, which is contributing to the growing popularity of new technologies, such as the Digital Twins (DTs) [1]. In particular, the technological advancements achieved have enabled its implementation and diffusion in numerous sectors of industry [2].

DTs are virtual copies of physical entities, such as machines or systems, reproducing their form, status, properties, behaviour and rules. Their objective is to simulate the behaviour of the respective entities in order to monitor their state, recognise complexities and anomalies, assess system performance and predict future trends [1]. The physical asset and its digital representation can communicate mutually through bi-directional and real-time interactions [2], resulting in a complex system [1]. They are therefore useful for monitoring and verifying the functioning of the actual system, as well as for proposing changes, optimisations and improving decision-making processes [2].

The inherent complexity of DTs and their increasing adoption across industries leads to the need for an infrastructure and a methodology that facilitates their implementation and management. To this end, DevOps technology can be integrated.

DevOps is a software development methodology that combines *development* (Dev) and *operations* (Ops) with the view of providing efficiency, speed, and security during software development and delivery. It is a culture of collaboration between development and operations teams that have shared responsibility for fast and reliable delivery of software. By integrating principles such as automation, continuous improvement, and fast feedback, DevOps gives a team the ability to

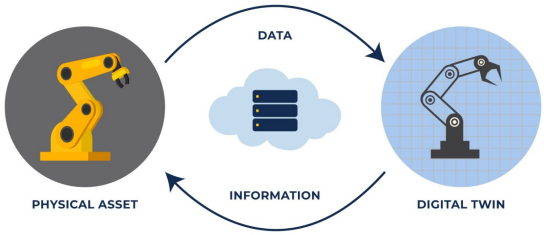

Figure 1.1: Physical asset and its digital twin [3]

deliver high-quality software quickly. It puts an emphasis on teamwork, reducing waste, and maintaining a strong focus on user requirements, and provides an advantage to businesses with this methodology [4].

1.2 Digital Twins as a Service software platform

Digital Twin as a Service is a software platform developed by the INTO-CPS Association at Aarhus University. It supports the entire lifecycle of DTs and is based on the *Build-Use-Share* model [5]:

Build: enables the creation of DTs by reusing existing assets.

Use: allows users to run DTs directly on the platform.

Share: allows users to share their DTs (or the services offered by them) with other users.

The fundamental goal is to allow collaboration between different users, both in the realisation of DTs through the sharing of assets, but also of ready-to-use DTs or the services they offer. Users are in fact provided with private workspaces, but can also collaborate through the Library. This not only facilitates the realisation of DTs through the Library, but also facilitates the use of DTs by non-technical users [6].

1.3 Motivation

The increasing popularity of DTs in different domains results in the need for tools and methodologies that simplify their creation and use, especially when considering the complexity of digital models and the necessary interaction between the DT and

its physical asset. Especially from the perspective of non-technical users, there is a need to provide simple and intuitive interfaces.

These issues could be solved through the use of DevOps practices, making it possible to use an automated and user-friendly environment. For this reason, the thesis aims to provide a framework that combines DevOps and DTs.

1.4 Objectives

This thesis was dedicated to the design and implementation of an integrated framework using DevOps practices for the efficient management of DTs. The aim was to simplify the entire process by offering users of the DTaaS software platform the possibility of interacting with DTs via user-friendly web interfaces.

The principal objectives were:

Design of an automated infrastructure: design an infrastructure that allows users to easily execute their DTs.

DevOps integration: application of DevOps principles, such as the use of pipelines, to implement the necessary infrastructure.

Lifecycle management of DTs: offer the possibility for users to create, delete, modify their DTs.

Intuitive user interfaces: create user-friendly web interfaces to facilitate interaction with DTs, making the process accessible even to non-technical users.

Evaluation with end users: testing the proposed framework through surveys with end users, analysing feedback and identifying strengths and areas for improvement.

By integrating infrastructure and user interfaces, the ultimate goal was to streamline the complex procedures associated with DT management.

1.5 Structure

The following section provides an overview of the remaining chapters of the thesis and a suggestion for the order of reading.

Chapter 2: aims to identify standards, common challenges, and best practices in the design and development of user interfaces, with an emphasis on optimizing the user experience through the abstraction and automation of DevOps tools. It also analyses the latest DT platforms to provide an overview of the state of the art

in user interfaces for DTs.

Chapter 3: gives an overview of the technologies used in the project.

Chapter 4: describes the implementation choices of the DevOps framework.

Chapter 5: describes the key aspects and the main components that have been implemented in the front-end.

Chapter 6: presents the results of user testing, intending to identify potential problems and strengths.

Chapter 7: discusses the results of this thesis and suggests possibilities for future work.

Chapter 2

Related work

This chapter introduces a new perspective that shifts the focus from purely technical advances to optimising the user experience by abstracting and automating DevOps tools. The chapter then analyses common standards and challenges in the field of Human-Computer Interaction (HCI). It also examines the state of the art in user interfaces for DT applications, with an in-depth look at some case studies implementing a traditional graphical user interface (GUI).

2.1 DTs and DevOps

In recent years, the integration of DTs and DevOps has gained increasing attention as a strategy to support the development, integration and deployment of DTs.

An important example of this approach is presented in the article authored by Aissat et al. [7], which analyses how a DevOps infrastructure can improve the scalability and evolution of DTs. One of the main pillars proposed by the authors is the DevOps infrastructure to ensure continuous and automated workflows across all development phases.

Building on these integrations, this thesis proposes a shift towards new approaches that focus on improving the user experience of DT management. Instead of using DevOps tools to streamline the technical part of code development, the aim is to abstract these tools and automate their functionalities in a way that simplifies operations from an end-user perspective. The latter will help to decouple highly technical layers of DT management from the user interface, making complex workflows invisible to the user.

In particular, the thesis points to the development of mechanisms that allow the smooth integration of DevOps processes into DT environments, without revealing

their complexity. This means that automation and abstraction will allow end users to perform such advanced operations as starting DT executions, view execution logs or configuring new assets without having any experience with DevOps tools. By hiding the underlying infrastructure, this methodology ensures that it works on its own, reducing not only accessibility thresholds but also potential user errors. This shift allows users to focus on domain-specific activities, while benefiting from the robustness, scalability and automation typically expected from a DevOps infrastructure.

2.2 User Interface and HCI

2.2.1 Definition

GUIs are crucial aspects of information technology because they bridge the gap between human users and computer systems. Research in the field of HCI design is essential as it facilitates the best application of its principles to user interfaces [8].

The book authored by Alan Dix et al. [9] illustrates that HCI is a term that has been popular since 1980, although it has older roots and was already applied in other domains, as discussed in the same source. It is a multidisciplinary field dealing with the design, evaluation, and implementation of interactive systems to be used in the context of user tasks and work. The entities involved are the *user*, the person trying to accomplish a task through an application, and the *computer*, a generic term that includes any technology from general desktop computers to embedded systems. The focus is on the interaction between the user and the computer, occurring to perform a task to satisfy a user's need. For a product to be successful from an HCI perspective, it must fulfil three requirements. It must be:

1. *useful* to satisfy the user's purpose
2. *usable* to perform it easily and without risk of error
3. *used* because it must be attractive enough for users to want to use it

Implementing well-designed GUIs is an expensive and time-consuming process [10]. It is crucial to devote attention and resources to such processes, as graphical interfaces are not only the meeting point between the user and the system, but they are fundamental to the success of a product in the marketplace [11]. A well-designed GUI can significantly improve usability, which is now a factor that can make the difference between users choosing one product over another [10]. We can imagine that the differentiating factor between competing products will be less and less the technical features per se, but rather the user interfaces and the degree of usability they offer [12].

2.2.2 Guidelines

There is no universal and generic theory concerning HCI [9], but it is possible to identify general guidelines and practices.

An artichle authored by Chao et al. [8] describes the three main phases on which the design of a GUI according to HCI principles is based. The first phase is *structural/conceptual design*, which focuses on analysing users' needs and the goals they need to achieve through tasks. It involves extensive research, which is carried out on users to understand their abilities, their experiences, and their reactions to different types of design. The second phase is *interactive design*, which aims to facilitate the interaction between the user and the computer by fine-tuning the various mechanisms for interacting with the interface. The last phase is *visual design*, which deals with the aesthetic aspect of the interface, taking into account the psychology and perceptions of users.

2.2.3 Common challenges

When dealing with user interfaces, some inherent problems make the task complex and require specific HCI skills.

An article authored by Myers et al. [10] illustrates some of the most common difficulties. The first difficulty concerns one of the fundamental characteristics of HCI: detailed knowledge of the users and the tasks they must perform to satisfy their needs. It is crucial to know the user's skills and experience so that the design of the functionalities offered by the application can be adapted accordingly. It can be a difficult task if, for example, the requirements for the application are still vague, incomplete, or even incorrect. In a more general way, it can be complex to formulate from the outset what all the possible users might be. It is not easy to predict how users will interact with the system, so it is essential to involve users directly, as developers may not be capable of adopting their perspective. In addition, the tasks and the target domain may be complex. Further complicating the design process is the need to meet the needs of different types of users, each with their possible workflows. Standards and guidelines are not sufficient, as it remains a creative process.

2.3 User Interfaces in DTs platforms

Well-designed GUIs are essential in complex applications [11]. Even in software platforms based on the use of DTs, it is necessary to have a human-machine interface to interact with the DT [13] and to guarantee that the user is aware of the state of the virtualised physical system [14]. Regardless of the type of DT and its application domain, the increasing importance of user interaction is driving the evolution of

the features provided, which should include the ability to change the state of the physical entity, display complex visualisations, and integrate technologies such as augmented or virtual reality [14]. Well-designed user interfaces could facilitate the management and interaction with physical counterparts. However, designers may face various challenges in identifying possible opportunities and risks during the design process [15].

It is necessary to facilitate real-time interaction between users and DTs, providing clear feedback on the behaviour of digital representations of physical entities. Interactivity is particularly important in the case of critical DT applications [13] and when users can make decisions based on the behaviour of the DT [16], such as in DTaaS for what-if analysis [6].

The data generated may be huge and complex, making it difficult to visualise and interact with the DT and not allowing effective user interaction [13, 17]. Well-designed interfaces can help to present complex data to the user, facilitating analysis and understanding.

User-friendly interfaces may also support collaboration between different users, for example by allowing communication and coordination through a dashboard [14]. They may also allow the application to be used even by non-specialised users. In fact, the ability to easily configure DT is required to support domain experts who do not have high knowledge in computer programming, 3D modelling or AI algorithms [15].

2.4 Case studies

The growing complexity and importance of DTs has led to the development of various DT frameworks, both commercial and open-source [18]. In order to provide a general overview, this section presents an analysis of the UIs of four platforms: Amazon Web Services (AWS) IoT TwinMaker, Azure Digital Twins, Eclipse Ditto and AASX Package Explorer. It examines how their interfaces contribute to usability and the overall user experience.

2.4.1 AWS IoT TwinMaker

The AWS DT platform [19] offers a sophisticated and user-friendly interface designed to streamline the creation and management of DTs (see figure 2.1). The user-centric design of the AWS platform significantly enhances usability, making it accessible for users with varying levels of technical expertise. It provides a comprehensive dashboard that consolidates critical information and controls in a

single view. This layout enhances user experience by allowing users to monitor system status, performance metrics, and alerts without navigating multiple screens. AWS offers rich visualization tools that enable users to interact with their DTs through graphical representations. These tools support real-time data updates and dynamic interactions, making it easier to understand complex data sets and system behaviors. The platform allows users to customize their dashboards with widgets that display relevant information. This flexibility helps users tailor their workspace to their specific needs, improving efficiency and satisfaction. AWS's UI emphasizes intuitive navigation with clearly labeled menus and straightforward workflows. Users can easily access different sections of the platform, such as device management, data analysis, and automation settings.

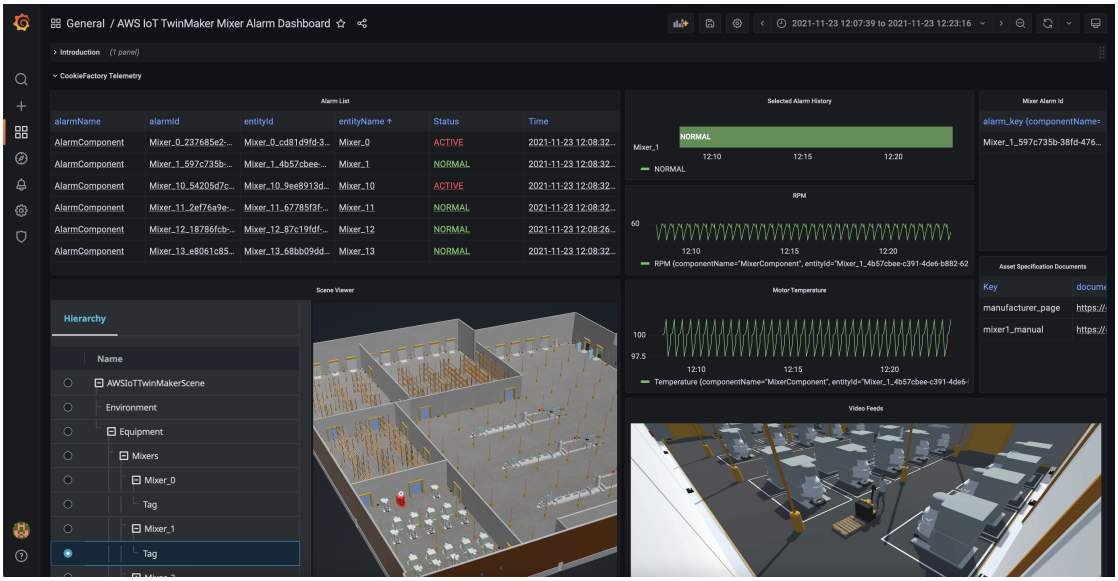

Figure 2.1: AWS IoT TwinMaker [20]

2.4.2 Azure Digital Twins

Microsoft Azure's DT platform [21] is known for its robust and user-friendly interface that facilitates efficient DT management (see figure 2.2). Azure provides an integrated development environment that combines coding, configuration, and management tools in one interface. It offers a graphical modeling tool that allows users to create and visualize their DTs using drag-and-drop components. This feature is particularly useful for users who prefer visual programming over text-based coding. The platform includes real-time monitoring capabilities that display live data feeds and system statuses. Users can set up alerts and notifications to stay informed about critical events and performance issues. Its UI supports detailed user

role and permission management, ensuring that access to various features and data is appropriately controlled. This capability enhances security and collaboration within teams.

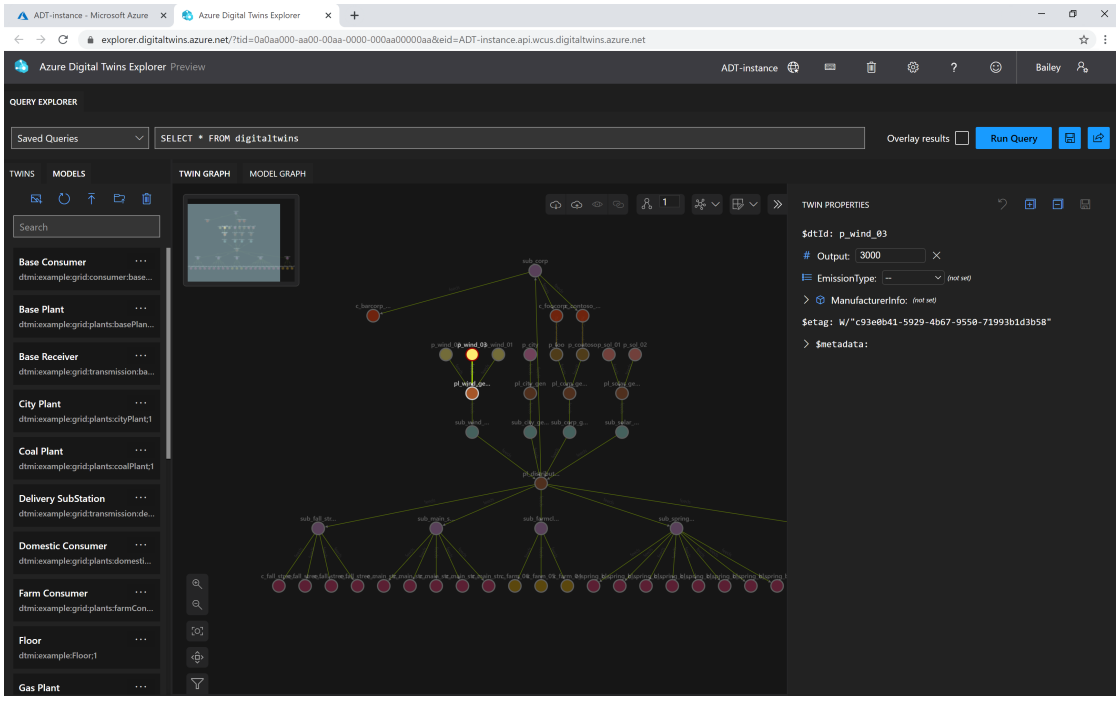

Figure 2.2: Azure Digital Twins [22]

2.4.3 Eclipse Ditto

Eclipse Ditto [23], which is the Eclipse DT platform representing the open-source community, offers a flexible and customizable interface designed to meet different user needs (see figure 2.3). Its UI is built on a modular design, allowing users to add or remove components based on their requirements. This flexibility is particularly beneficial for developers who need to tailor the interface to specific projects. The platform supports integration with various open-source tools and libraries, providing users with a wide range of options for extending functionality and customizing their workflows. As an open-source project, Eclipse benefits from continuous improvements and contributions from the community. Users can access and contribute to a growing repository of plugins and extensions, enhancing the platform's capabilities. Like AWS and Azure, Eclipse offers customizable dashboards that enable users to display relevant information and controls. This feature improves user experience by allowing personalized configurations. The platform provides extensive documentation and user guides, which help users

navigate the interface and utilize its features effectively. This support is crucial for ensuring that users can leverage the platform's full potential.

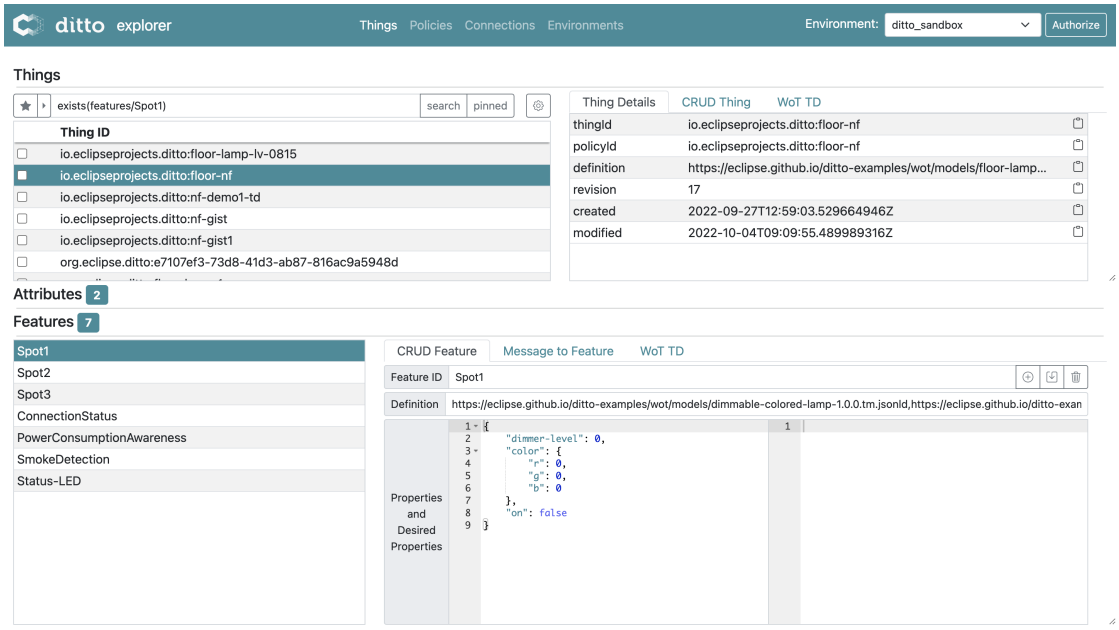

Figure 2.3: EC Ditto [24]

2.4.4 AASX Package Explorer

AASX [25], a part of the Asset Administration Shell (AAS) implementations, provides a reference standard and framework for DTs in the industrial context of Industry 4.0 [18] (see figure 2.4). The AASX framework offers a GUI client that allows users to manage AAS objects in various formats such as AASX, JSON, and XML. The AASX interface is noted for its simplicity and easy of deployment. It includes various screencasts to help learn and adapt to the system quickly, reducing the learning curve and making it accessible to a broader range of users. While the AASX interface excels in managing static assets, it is limited in its native support for dynamic assets.

The summary table 2.1 analyses the relative pros and cons of the interfaces of the examined platforms. This table marks both strengths that play a role in the improved usability and user efficiency and limitations that may affect the overall experience.

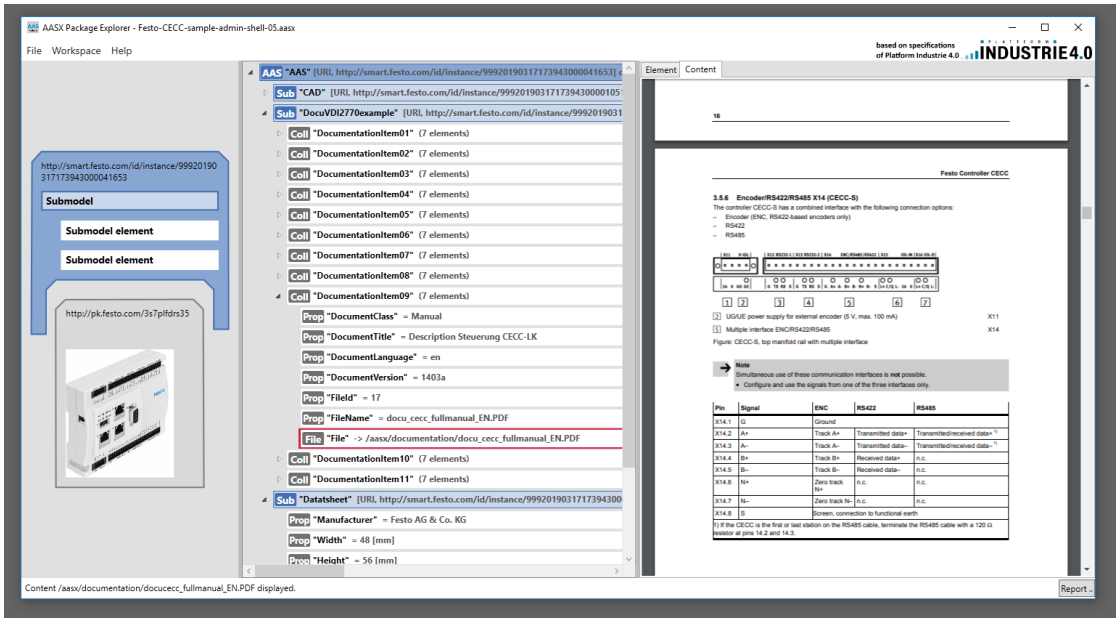

Figure 2.4: AASX Package Explorer [25]

Table 2.1: Comparison of pros and cons of different DT platform interfaces

Platform	Pros	Cons
AWS IoT TwinMaker	- User-friendly and intuitive interface is accessible for users with various technical skills. - Centralized dashboard consolidates critical information in a single view, improving efficiency. - Customizable widgets enable users to customize their dashboards, improving user satisfaction. - Advanced visualization tools support real-time data updates and dynamic interactions, aiding in the interpretation of complex data sets.	- Advanced features may require a more challenging learning process for users with less experience. - Customization options may overwhelm users with a preference for simplicity.

Platform	Pros	Cons
Azure Digital Twins	- Combines coding, configuration and management tools into a single UI. - Graphical modeling tool provides drag and drop support, helping those who prefer visual programming. - Real-time monitoring enhances system oversight and responsiveness.	- Complexity of features may be difficult for non-developers or anyone without experience in programming or modeling. - Feature-rich and real-time monitoring often add a performance overhead which can affect the efficiency of system operations.
Eclipse Ditto	- Modular design allows interface customization, making it highly adaptable to various user needs. - Supports integration with a wide range of open-source tools and libraries, offering flexibility. - Continuous community contributions ensure up-to-date functionality and new features. - Customizable dashboards improve user experience by allowing tailored configurations.	- Open-source nature might lead to varying levels of support and documentation quality, which can hinder users' ability to fully leverage its features. - Customization flexibility may require more effort from developers.
AASX Package Explorer	- Simple and easily deployable interface, reducing the learning curve and broadening accessibility. - Includes screencasts and tutorials that aid in user adaptation and expedite the process of user onboarding. - Effective management of static assets, particularly in the industrial context.	- Limited native support for dynamic assets restricts its applicability in more complex or real-time DT scenarios. - Less flexibility compared to other platforms.

Chapter 3

Background

This chapter will present the principal technologies employed in this thesis. A theoretical context is provided to give a more complete view of the work.

3.1 HCI and Agile development

The thesis work is organised according to the principles of Agile development. It thus becomes necessary to integrate HCI criteria with those of Agile development. Despite the lack of standardised guidelines for achieving this [26], an overview of the different possibilities is provided below.

As reported in De Silva et al. [27], agile development was introduced to significantly reduce the time required to release a software product, as well as to provide a solution for change management during the development process. Additionally, it is an approach that takes teamwork and project management into account. The term *agile* encompasses several different iterative methodologies which share the same underlying principles. The Agile Manifesto, published in 2001, elucidates the principles underlying agile development in 12 concise points [28].

Agile Software Development Processes (ASDPs), which aim to enhance the velocity of software production, do not typically prioritize usability. Furthermore, they do not offer guidance on how to enhance end-user satisfaction through usability engineering, which is a fundamental tenet of HCI [26]. Since they focus on different aspects of software development, using them in combination may not be immediate. The initial phase of human-computer interaction comprises the analysis of requirements and the creation of a series of low-fidelity prototypes, executed iteratively and progressively improving the level of detail. This approach may not be optimal when applying the principles of agile development, which prioritises the continuous delivery of working software with each iteration [29].

For this thesis work, a simple integration model was used, based on the one proposed by Tariq [29]. The steps included and slightly revised are:

Meetings: the identification of the different types of users and their requirements is made possible through a series of meetings, which are therefore a fundamental part of the process. Agile development is characterised by a reduction in the amount of documentation required in comparison to other traditional methodologies. This allows the project to be divided into smaller, more manageable components, which are then addressed iteratively. Any issues or feedback that arise after an iteration are handled at the next sprint.

Mid-fidelity mock-ups: used to gather feedback before implementing interfaces, they are useful as they allow to obtain advice from a usability point of view.

Specify the Context of Use: the context in which the product will be used is defined, and meetings are conducted to obtain more and more details to exploit.

Prioritisation of Requirements: necessary to define the order in which the various requirements will be designed and implemented, based on user interest.

Implementation: interfaces are coded using established languages and technologies.

Unit Testing: individual components of the system are tested independently in order to verify their correct functioning.

Each sprint involves the execution of a specific subset of the interfaces to be realised, contributing to the progress of the project in an incremental and iterative manner, improving its final quality.

3.2 Prototyping

Prototyping is a technique where a preliminary model or simulation of a product is made to test ideas and features before full production. The activity ranges from a low-fidelity sketch to a high-fidelity interactive design. It helps teams to validate ideas before the detailed development process [30].

An article by The Good [30] underlines the importance of mid-fidelity prototypes, which are also known as *mock-ups*. They achieve a balance between the simplicity of low fidelity and the precision associated with high fidelity. Mid-fidelity designs are well suited for expedited testing; they give out enough detail to elicit beneficial feedback but avoid overwhelming users or the need for too many resources. This approach allows for the effective optimisation of user experiences.

3.3 DevOps

3.3.1 GitLab

GitLab is employed as an OAuth service provider, enabling users to securely access the DTaaS website.

Each user is allocated a personal folder on the GitLab platform, which is organised in subfolders to facilitate the management of their DTs assets. Figure 3.1 illustrates the subfolders present in each user's GitLab account.

Name	Last commit	Last update
common	Adds small examples for...	4 weeks ago
data	filename changes	4 weeks ago
digital_twins	Vanessa scherma main p...	1 week ago
functions	filename changes	4 weeks ago
models	filename changes	4 weeks ago
tools	filename changes	4 weeks ago
.gitlab-ci.yml	Vanessa scherma main p...	1 week ago
BUILD.md	Vanessa scherma main p...	1 week ago
README.md	Initial commit	1 year ago

Figure 3.1: User file system on GitLab

The digital_twins folder contains DTs that have been pre-built by one or more users. The intention is that they should be sufficiently flexible to be reconfigured as required for specific use cases [5].

3.3.2 CI/CD pipelines

Continuous Integration (CI) and Continuous Deployment (CD) represent two key components of the DevOps methodology. CI involves frequent integration of code changes into a common repository. Each integration triggers automated builds and tests that permit the detection of issues at an early stage. This practice ensures that the changes made to the code are checked fast enough, reducing the possibilities of integration problems and hence ensuring high-quality software. CD automates the process of release, ensuring that code changes are automatically

tested and prepared for deployment. Teams using CD can deploy updates rapidly and reliably, improving the responsiveness and quality of software. Performed together, CI/CD automates the whole delivery pipeline for software, increasing efficiency and reducing errors. They entirely eliminate, or significantly reduce, the manual human input required for a code change to be moved from a commit to a production environment. The entire process of compilation, testing (including unit, integration and regression testing) and deployment, as well as infrastructure provisioning, is included [31].

A CI/CD pipeline is a series of automated processes that manage CI and CD of software. They are configured to run automatically, with no need for manual intervention once activated.

GitLab is a single application for the entire DevOps lifecycle, which means it performs all of the basics required to CI/CD in one environment. The documentation provided by GitLab was instrumental in enabling a comprehensive understanding of the CI/CD pipelines [32]. Pipelines are composed of a number of essential components. *Jobs* delineate the specific tasks to be accomplished, while *stages* define the sequence in which jobs are executed. In this way, stages ensure that each step takes place in the right order and make the pipeline more efficient and consistent. In the event that all jobs within a stage are successfully completed, the pipeline will automatically proceed to the subsequent stage. However, if any of the jobs fail, the flow is interrupted without proceeding.

When a pipeline is initiated, the jobs that have been defined within it are then distributed among the available runners. GitLab runners are agents within the GitLab Runner application that execute the jobs in accordance with their configuration and the available resources. They can be configured to operate on a variety of platforms, including virtual machines, containers, and physical servers. They can also be managed locally or in a cloud environment.

3.3.3 Gitbeaker

In order to gain an understanding of the Gitbeaker library, the corresponding repository, which is available on GitHub, was consulted [33].

Gitbeaker is a client library for Node.js that enables users to interact with the GitLab API. In particular, gitbeaker/rest is a specific version of the Gitbeaker package that allows users to submit requests to GitLab's Representational State Transfer (REST) API.

An API that adheres to the principles of REST architectural style is called a REST API. They facilitate communication between client and server applications using standard HTTP methods such as POST, GET, PUT, and DELETE in order to create, read, update or delete (CRUD) resources. They are stateless, meaning

that each request has to contain all the information needed for processing. These APIs can be built in different programming languages while they support multiple data formats like XML or JSON [34].

One of the most significant features of Gitbeaker is the provision of support for a range of authentication methods, including the use of personal tokens and OAuth keys. Gitbeaker provides a range of predefined methods for requesting data from the various GitLab APIs, eliminating the need for users to manually construct HTTP requests, thus greatly simplifying the integration process with GitLab.

It automatically handles errors in HTTP requests, providing meaningful error messages that help diagnose and resolve problems in a timely manner.

Finally, it is fully compatible with all of GitLab's REST APIs.

3.4 Testing

The goal was to ensure the highest possible level of quality and reliability; therefore, a comprehensive approach toward testing was followed during the development phase.

Testing is one of the stages in the cycle of software development aimed at confirming the fact that an application will work in an expected manner and meet all the specified requirements [35]. Effective testing practices in modern software development are necessary to achieve CD and guarantee the robustness of the codebase [36].

In this thesis work, a comprehensive suite was used to test the application on most of its levels in a systematic way. This not only incorporates checking the overall functionality and systematic integration but also focuses on checking the correctness of individual components. Every component introduced to develop the user interfaces was diligently tested to provide a solid, reliable application, assuring a seamless user experience.

The software testing pyramid shown in figure 3.2 is an essential strategic model in Agile software development to structure efficient testing efforts within an application. It is divided into three key layers [37]:

- **Unit testing:** it forms the base of the pyramid, involving a number of small and fast tests focused on individual components or functions. These tests are the most frequently run and the cheapest to execute, thus providing real quick feedback and making sure code quality is maintained. This efficiency helps identifying issues in the early stages of development.

- **Integration testing:** it occupies the middle layer and is concerned with the interaction between components. These tests are less in number, more complicated and expensive than unit tests.

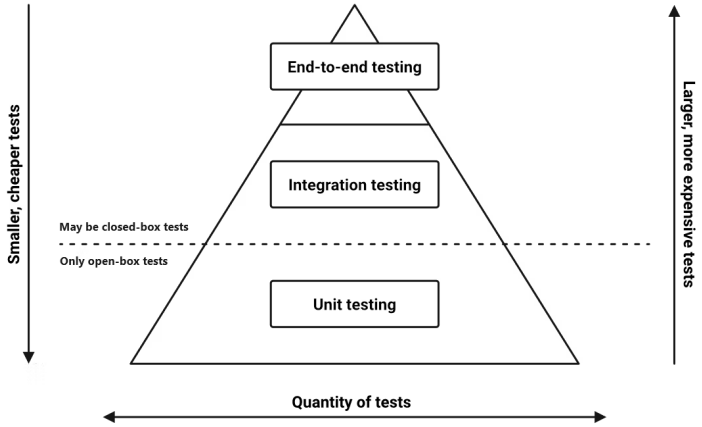

Figure 3.2: The testing pyramid [37]

- **End-to-end testing:** it sits at the top and involves very few but quite intensive tests that truly represent real user scenarios to validate the whole application workflow; they're also the most expensive and least often run (normally before major releases).

The testing pyramid structure drives a balanced approach to testing: more low-cost unit tests, fewer high-cost E2E tests, with adequate resource allocation and comprehensive test coverage. This model supports Agile principles and CI/CD workflows effectively, facilitating fast and reliable software development [37].

3.4.1 Unit testing

This subsection on unit testing is highly based on the work of Khorikov [38]. This book acts as the base source for studying the principles and practices of unit testing in depth. The unit testing methodologies, good practices, and patterns outlined have given a solid framework for understanding how to effectively implement unit testing in modern software development.

Unit testing confirms that every unit of the software is working as expected. A *unit* normally stands for a single smallest testable area of an application, like functions or methods. The simplest objective of unit testing is to see if units operate well separately from the rest of the system. This technique not only allows the detection of bugs at the earliest possible point but also makes it easier for maintenance and refactoring of code.

Unit tests are usually automated and, therefore, can be run frequently in the development process to ensure that new changes do not break existing functionality. Khorikov emphasizes that unit tests are a safety net for the developer, who can

freely modify the code, knowing that any regression in functionality will be noticed fast.

Various key principles and best practices of unit testing have been considered in the strategies of testing. They include, among others:

Isolation: any unit test should be independent of other tests. This isolation ensures that one test does not interfere with another test through its result. This quite often requires isolating the dependencies and controlling the environment of the test.

Simplicity: unit tests should be simple, focused on one single aspect of functionality at a time. This simplicity will pay out when sources of issues have to be discovered and makes the tests easy to understand and maintain.

Automated run: automated tests can be run at a very high frequency and integrated nicely into the CI pipeline. Such automation is very important for the sustenance of code quality over time through periodic efficient validation of the software.

Descriptive naming: tests should be descriptively named to explicitly tell what the purpose of each is. This improves readability, and any developer is empowered to know what the tests intend to do without even having to read the implementation at all.

Testing coverage: all varieties of scenarios should be covered under unit tests, including edge cases. Good test coverage ensures the code behaves well under all conditions.

All these principles were followed and 100% test coverage was achieved.

3.4.2 Integration testing

Unit tests may not be enough to ensure everything will work as expected in the system as a whole when the parts are composed. Integration testing fills this gap by checking that different modules or services in the application interact correctly.

An article by Katalon [39] provided the details for this subsection. Integration testing is the stage of the software development life cycle where individual modules are integrated and tested as one unit. The purpose is to test the interfaces between the units integrated and ensure that all the units work together as desired. This is very important because it detects faults occurring due to interaction of different components among themselves. These include problems in data flow, interface mismatches, and communication errors. The scope of integration testing includes the interface between modules and verifies whether they are communicating correctly

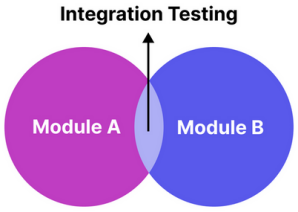

Figure 3.3: Integration testing [39]

with each other, also handling data exchanges correctly. Integration testing also ensures that correct system behaviors manifest when the components come together. This is a quite critical step where systems have complexity and require the smooth functioning of a good number of modules. The major goals of integration testing are the improvement in the quality of the software, detecting defects at early stages of development, and ensuring that all the parts of the system work together smoothly. Detecting the integration problems early allows the development team to minimize the risk of bugs that will require more resources to be spent in order to fix them later in the development cycle.

3.4.3 End-to-end testing

The information for this subsection has been sourced from an article by TechTarget [10]. End-to-end (E2E) testing is a software testing methodology that confirms the functional correctness of a software product from start to finish, guaranteeing that all constituent parts work together seamlessly in real-world scenarios. It takes into account the whole application from the user's perspective, simulating a full journey through the application while validating the system flow, integrity of data, and integration points. In this method, dependencies are highlighted and confirmed to prove that software does what is expected under real-world conditions. It is a rigorous methodology that ensures any potential faults or errors are detected before software is deployed, significantly reducing the possibility of bugs on the production environment.

It involves creating test scenarios that emulate user interactions, tending to look at data flows across the system to understand the dependencies and errors. It requires a full dedicated test environment with reproducible conditions, such as latency or traffic. Testing is usually iterative, with issues identified, resolved, and retested to ensure that everything works as intended within the system.

The challenges that exist in E2E testing include detailed test case design and possible slow times in execution, especially when Agile development environments are applied. The best practices to meet these challenges include keeping a record of testable functions, tracking data flow, and proper retest and focusing on user

experience while designing tests.

3.4.4 Acceptance testing

This section on acceptance testing is based on an article by Testsigma [40], which emphasises the critical role of this kind of testing in the software development lifecycle.

Acceptance testing is the most elaborate and, consequently, the most expensive and time-consuming stage of the software development process. The amount it costs is considerably higher because of extensive preparation, time-consuming execution, and participation by stakeholders, comprising users and customer representatives. Additionally, test counts are generally lower for the acceptance test in light of the broader scope and intricate details carried along with the tests. While acceptance testing might get slow and expensive, this is where software really shows value and the final goals it provides to the users.

There are a number of advantages associated with acceptance testing that greatly enhance the quality and user experience of the ultimate product. This step, which validates the software against predetermined acceptance criteria with input from end-users or stakeholders, is responsible for end-to-end evaluation and, in turn, the level of performance and functionality of the software in real-world conditions. One of the main benefits of the acceptance testing is that it delivers direct feedback from users to developers. As the test involves end-users, the developers can get their insights into knowing how actually the software will perform in real-life use scenarios. This way, therefore, users can easily track disparities between their expectations and performances of the software, whereas developers are equally in good positions to make modifiable informed changes to suit the user's needs easily. Such feedback helps refine the software towards the meeting or exceeding of user expectations, hence increasing user satisfaction. Furthermore, acceptance testing allows for comprehensive coverage of the test, which is another significant advantage. In this phase, a multitude of potential scenarios and conditions that users may encounter are tested, thereby facilitating the assurance of the software's functionality in a multitude of real-world contexts. It tests all aspects of software functionality, starting from the user interface to business logic and performance.

There exist acceptance testing entry and exit criteria to prove its effectiveness, showing that the software takes into consideration the required standards.

The **entry criteria** involve the following:

- All software requirements are well documented and analyzed.
- A comprehensive test plan is ready and approved.

- The test environment is completely set up and is ready to be operational.
- The test cases are designed and reviewed for completeness and accuracy.
- The necessary test data is ready.

In contrast, the **exit criteria** are as follows:

- All test cases are executed as planned.
- All the defects are identified, documented, and reported.
- All the critical defects are addressed and resolved.
- The software meets all predefined acceptance criteria.
- The software is considered ready for user acceptance and eventual deployment.

3.4.5 Jest

The implementation of unit and integration tests was conducted using Jest, a comprehensive JavaScript testing framework. Jest is designed to guarantee the correctness of any JavaScript codebase, offering an intuitive and straightforward testing environment. The Jest official documentation [41] was used to understand its features and its effective usage.

One of the significant features of Jest is its zero-configuration setup. It just works, out of the box, with most JavaScript projects and requires no upfront configuration. Secondly, it contains rich support for mocking functions, modules, and timers. This feature is very important to unit tests, isolating components and testing them independently by simulating their dependencies. Apart from that, it parallelizes test execution using worker threads, which is very useful in terms of execution speed for large codebases with large test suites. Built-in code coverage reporting also allows seeing the areas of code under test, highlighting untested areas, and hence guiding the creation of further tests in order to ensure complete coverage.

The unit tests were developed in accordance with the Arrange, Act, Assert (AAA) pattern, as outlined in the Khorikov publication [38]. The biggest advantage of this pattern is the simple uniformity structure for all tests in the suite. Any test becomes easier to read and understand, reducing the maintenance cost for the whole test suite.

This pattern splits each test into three parts:

1. *Arrange*: the state of the System Under Test (SUT), together with its dependencies, is arranged into a desired state.

2. *Act*: it involves calling methods on the SUT, passing the prepared dependencies, and capturing the output value (if any).

3. *Assert*: the outcome is checked in the final section. It may be checked by return value, final state of SUT and its collaborators, or methods that the SUT called on those collaborators.

Chapter 4

DevOps framework

4.1 Overview and objectives

At the time the thesis work started, the DTaaS application used a Jupyter Notebook to execute the DTs and Jupyter Lab to manage files and executable operations in Jupyter Notebook [5]. In order to meet the development requirements of the user interfaces, a DevOps framework was developed to support all the necessary operations. The objective was to enable interaction with the DTs via Application Programming Interface (API) calls, so that users could start, monitor and manage their DTs via the web application.

4.2 High-level design and architecture

The architectural design of the DevOps framework was intended to facilitate the management of DTs. It is based on two key elements:

- The Gitlab CI/CD infrastructure, which employs a parent-child pipeline hierarchy. The objective of this infrastructure is to enable the triggering of a pipeline of a specific DT by simply passing the necessary data as parameters, such as the name of the DT and the tag of the runner that will execute the pipeline.
- Some classes implemented in the code that utilise Gitbeaker to realise the APIs required for interaction with DTs.

As illustrated in the component diagram in figure 4.1, The infrastructure consists of three main classes: `DigitalTwin`, `LibraryAsset`, and `GitlabInstance`.

The distinction between the DigitalTwin and LibraryAsset classes was necessary to separate the full management of a DT from an asset visualised through the

library. The LibraryAsset class provides a significantly reduced set of functionality compared to the DigitalTwin, focusing only on asset visualisation.

Intermediate classes have been introduced to ensure a clear separation of file management responsibilities: `DTAssets` and `LibraryManager`. These classes implement the necessary logic to mediate between a DigitalTwin or LibraryAsset and the `FileHandler` class. The FileHandler class has a single responsibility: to make API calls to files via GitBeaker. This design allows for the separation of high-level logic from low-level file operations.

The infrastructure requires that the DigitalTwin class and the LibraryAsset class include an instance of `GitlabInstance`. This composition relationship emphasizes the dependency between these classes, where a DigitalTwin or a LibraryAsset instance cannot function independently without a GitlabInstance. The GitlabInstance class provides the essential services required for interacting with GitLab, including API integrations and pipeline management.

The GitlabInstance class serves as the interface to the realized CI/CD infrastructure. By utilizing the Gitlab class imported from GitBeaker and initialized as its attribute, GitlabInstance facilitates the execution of pipelines and other CI/CD-related tasks. This architecture ensures that the infrastructure remains modular and adheres to the principles of single responsibility and clear dependency management.

4.3 Requirements and specifications

The functional requirements of the system include the automation of pipelines and the management of DTs via APIs. Consequently, the framework was designed to facilitate the comprehensive automation of the DT lifecycle, with the objective of minimising the necessity for manual intervention. The system must be capable of managing the dynamic configuration of pipelines, utilising variables that permit the customisation of pipeline behaviour according to the data provided by the user, such as the designation of the DT. Integration with GitLab is another fundamental requirement. The framework must be able to interact with GitLab to execute CI/CD pipelines via API calls, using Gitbeaker as a wrapper. Users must be able to authenticate themselves via GitLab's OAuth mechanism, and the system must automatically manage the authentication tokens and trigger tokens needed to start pipelines. In addition, the system must be able to automatically retrieve key information from the user's GitLab repository, such as the list of available DTs.

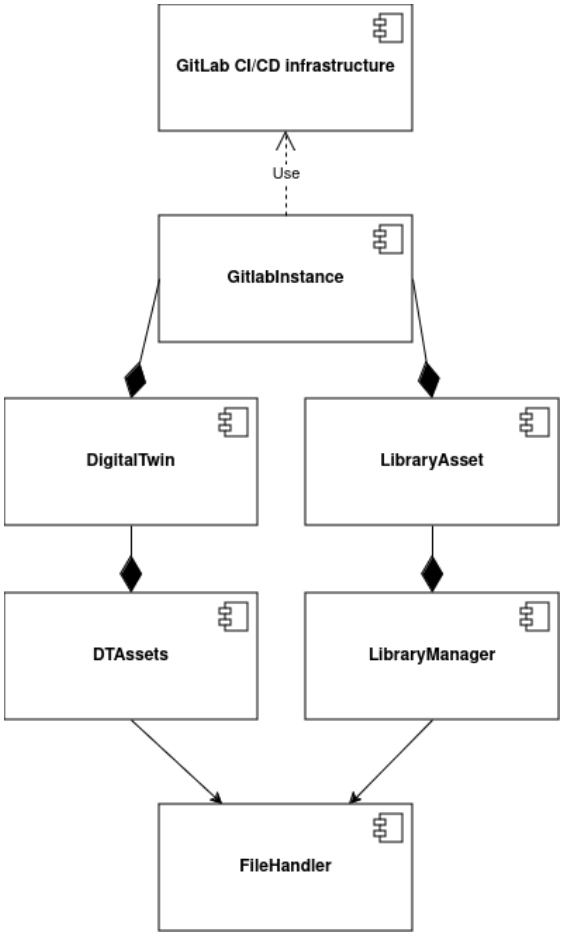

Figure 4.1: High-level design of the DevOps framework

4.4 GitLab CI/CD infrastructure

Given that files in the Library are stored in a Git repository, the approach employed was that of GitLab's parent-child pipelines. In this context, a parent pipeline initiates the execution of another pipeline within the same project, the latter of which is known as the child pipeline [42].

4.4.1 Parent pipeline

The parent pipeline was configured as a top-level element. There is a single stage called *triggers*, which is responsible for triggering other child pipelines.

On the `.gitlab-ci.yml` file triggers are managed for DTs that are inside the user repository. Each trigger is connected with one distinct DT and it becomes

active when the corresponding value of *DTName* variable is given by the API call. The *RunnerTag* variable is used to specify a custom runner tag that will execute each job in the DT's pipeline.

Below is the explanation of the keywords used in the CI/CD pipeline configuration [43]:

Image: it specifies the Docker image, like *fedora:41*, providing the environment for the pipeline execution.

Stages: it defines phases in the pipeline, such as triggers, organizing tasks sequentially.

Trigger: it initiates another pipeline or job, incorporating configurations from an external file.

Include: it imports configurations from another file for modular pipeline setups.

Rules: it sets conditions for job execution, based on variables or states.

If: a condition within rules that specifies when a job should run, based on the value of a variable.

When: it specifies the timing of job execution, such as *always*.

Variables: it defines dynamic variables, like *RunnerTag*, used in the pipeline.

```
image: fedora:41

stages:
  - triggers

trigger_mass-spring-damper:
  stage: triggers
  trigger:
    include: digital_twins/mass-spring-damper/.gitlab-ci.yml
  rules:
    - if: '$DTName == "mass-spring-damper"'
      when: always
  variables:
    RunnerTag: $RunnerTag
```

Figure 4.2: Parent pipeline

4.4.2 Child pipelines

Within the *digital_twin* folder of a user's GitLab file system, there is a folder for each DT. To automate the lifecycle of the corresponding DT, a child pipeline has been incorporated into each of these folders.

Each job within the YAML file encompasses specific stages, corresponding to distinct tasks. Regardless of the image provided in the parent pipeline, each child pipeline will use its own specified image or Ruby's default image.

With the DT mass spring damper serving as a point of reference in figure 4.2, the stages in question are designed to facilitate the creation, execution, and termination of the DT simulation, as well as the cleaning and restoration of the environment to ensure its readiness for future executions.

The following are the explanations of the keywords used within the CI/CD child pipeline [43]:

Stage: it defines the steps that happen in a pipeline sequentially, for example, create, execute and clean, to make sure that tasks occur in a specific order.

Script: it lists commands to be run at each step; for example, changing directories, modifying permissions, or running lifecycle scripts.

Tags: it specifies which runner should run the jobs, thereby providing an additional control over where and how the jobs are run.

4.4.3 API call

Once a trigger token for the pipeline has been created, a pipeline can be triggered [44].

It is possible to manually trigger a DT's pipeline using an API call, setting the *DTName* variable to the desired DT name and the *RunnerTag* to specify the GitLab runner. The call will be executed in the *main* branch.

The values to be supplied are as follows:

- <access_token>: the user GitLab trigger token.
- <digital_twin_name>: the name of the DT (e.g. mass-spring-damper).
- <runner_tag>: the specific tag of the GitLab runner that the user wants to use.
- <project_id>: the ID of the GitLab project, displayed in the project overview page.

```
image: ubuntu:20.04

stages:
    - create
    - execute
    - clean

create_mass-spring-damper:
    stage: create
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/create
        - lifecycle/create
    tags:
        - $RunnerTag

execute_mass-spring-damper:
    stage: execute
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/execute
        - lifecycle/execute
    tags:
        - $RunnerTag

clean_mass-spring-damper:
    stage: clean
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/terminate
        - chmod +x lifecycle/clean
        - lifecycle/terminate
    tags:
        - $RunnerTag
```

Figure 4.3: Child pipeline of "mass spring damper" DT

4.5 Implemented classes

In order to facilitate the management of the lifecycle of DTs via the web application interfaces, it was necessary to develop specific code within the project client. The code was designed to facilitate efficient API calls through the use of Gitbeaker as a wrapper, as this approach simplifies interactions with GitLab's REST API and reduces the complexity of the project code.

```
curl --request POST \
  --form "token=<access_token>" \
  --form ref=main \
  --form "variables[DTName]=<digital_twin_name>" \
  --form "variables[RunnerTag]=<runner_tag>" \
  "https://maestro.cps.digit.au.dk/gitlab/api/v4/projects/<
project_id>/trigger/pipeline"
```

Figure 4.4: Multine cURL command to trigger a pipeline

After testing the correctness of the API by implementing a `GitlabDriver` class, the APIs were integrated into the front-end. The latter was done by wiring up API endpoints to the front-end components, ensuring a seamless data flow. Unit and integration testing was done to ensure the coverage of all functional requirements and solve all problems regarding data consistency, performance, or user experience.

4.5.1 GitlabInstance

The GitlabInstance class was created in order to manage the APIs and information related to the GitLab profile, the project and the user-specific data stored in their account.

The username and the token required to instantiate the Gitbeaker *Gitlab* component, which is required for making the API calls, are retrieved from the session storage, taking the *access_token* of the user already logged into the DTaaS application.

The initialisation of the GitlabInstance object is concluded with the execution of the init() method, which enables the retrieval and storage of the projectId and triggerToken attributes. The *projectId* is a unique identifier for projects in GitLab and it is essential for subsequent API calls. For example, it is passed to the method that retrieves a *trigger token*, which is used to trigger CI/CD pipelines in GitLab.

The objective of the getDTSubfolders method was to retrieve the names and corresponding descriptions of the DTs of the user, so that these could be shown at the front-end interface. This approach would obviate the user from having to input the name of a DT; hence, saving the user from possible error and inefficiencies arising from manual input. The user interface makes it easier for the user to deal with DTs by automatizing their selection and manages them more accurately. This implementation also eliminates the necessity for manual input from users for the access token and the username, which are automatically provided via the GitLab

OAuth login.

Furthermore, logs maintained in the `GitlabInstance` class improve awareness and transparency over the operations conducted. The final three methods are employed in conjunction to oversee the execution of a DT. In particular, individual logs are saved for each job in the pipeline, and the status of the latter is monitored so that, once the entire pipeline is complete, the results can be displayed in details within the user interface. In this way, all statuses of each operation are logged for better debugging and performance analysis, including possible errors. Having trace logs exposed to the user means troubleshooting will be more effective and insight into execution and management of DTs will be gained from improving system reliability and user confidence.

4.5.2 DigitalTwin

The DigitalTwin class was created in order to manage the APIs and information related to a specific DT.

The creation of a DigitalTwin object requires a pre-existing GitlabInstance to be associated with the object. It was determined that matching a different GitlabInstance for each DigitalTwin would be the optimal approach to ensure the maintenance of independence between the various DTs. The *api* attribute of GitlabInstance facilitates the execution of Gitbeaker APIs pertinent to the DT.

The class allows a pipeline to be started and stopped, thus giving the user full control of the execution. The execute() method uses the previous methods internally. This approach ensures that there are no errors due to missing design information during the execution of the pipeline. Responsibilities have been divided into smaller methods in order to make the code more modular, facilitating debugging and testing. In both execute() and stop(), the status of operations executed on the DT is monitored, keeping track of them via the logs attribute of GitlabInstance. Errors are identified and tracked, providing a complete view and the ability to monitor performance.

The descriptionFiles, lifecycleFiles and configFiles attributes are used to keep track of the files within the corresponding GitLab folder of the DT, thus enabling the read and modify features.

The create() method enables the creation of a DT and saves all its files in the user's corresponding GitLab folder. Additionally, if the DT is configured as *common*, it is also added to GitLab's shared folder, making it part of the Library and accessible to other users.

Similarly, the delete() method removes a DT from GitLab. If the DT was part of the Library, it is also removed from the shared folder.

A crucial aspect of these two methods is their integration with the DevOps

infrastructure. When a DT is created or deleted, the `.gitlab-ci.yml` file of the parent pipeline is updated to add or remove the *trigger_DTName* section associated with the DT. This ensures that a user-created DT can be executed via the web interface without requiring manual updates to pipeline configuration files on GitLab. Instead, these files are automatically updated, providing an effortless user experience and maintaining alignment with the infrastructure.

4.5.3 LibraryAsset

The LibraryAsset class was created in order to manage the APIs and information related to a specific library asset.

It is similar to the DigitalTwin class, but contains only the methods required to display files. This focused design reflects its limited scope and ensures simplicity and clarity for use cases involving the library.

4.6 Prerequisites for the correct functioning of the framework

In order for the DevOps framework to function as intended, the following prerequisites must be met:

- The GitLab account used as OAuth provider must have a *DTaaS* group, a project under the username of the user, and a *digital_twins* folder which contains the DTs.
- Within their GitLab profile, the user needs to configure a *gitlab personal access token*, selecting all scopes for the access token, and a *pipeline trigger token*. The former is used as a security requirement to access the user's profile, the latter is used to trigger pipelines.
- In addition to the configuration of the CI/CD pipeline in the GitLab project, the user must configure at least one project runner with *linux* as runner tag, install it and integrate it with the GitLab project.

Chapter 5

Standard compliant user interfaces for Digital Twins

5.1 Overview and objectives

The UIs for the DTaaS software platform were designed to adopt DevOps practices. This implementation was made possible by the adoption of GitLab's CI/CD pipelines through Gitbeaker and GitLab APIs, making the platform interface directly to the GitLab account of the user. The design of the platform from the start adopted DevOps practices by automation of various aspects associated with lifecycles, run-time monitoring, and deployment. This automation reduced the need for human intervention and further streamlined operational processes to ensure users could efficiently manage their DTs through user-friendly interfaces.

The main focus on the UIs was to make the user experience as intuitive as possible, simultaneously addressing the most common drawbacks of the DT platforms analyzed in chapter 2.

The key goals with consideration of other interfaces were:

Complexity balanced with usability: unlike other platforms, which can be complex with a strong learning curve or weak performance because of heavy-weight tools, the DTaaS UIs were designed to balance complexity with usability.

Standardized dashboard: while some platforms offer a high degree of customization, it often comes at the cost of increased complexity for developers, making it burdensome to implement changes. In contrast, the architecture of the DTaaS UIs prioritizes accessibility and simplicity. Customization is intentionally not permitted to avoid overloading end-users with additional complexity,

particularly those with low technical expertise. Instead, a standardized dashboard is provided to ensure a consistent and user-friendly experience, allowing users to efficiently manage their DTs without needing to adapt to ongoing modifications or complex setups.

Integration with DevOps: in contrast to interfaces that encounter difficulties when incorporating intricate functionalities (for instance, the restricted support for dynamic assets offered by AASX Package Explorer), the DTaaS platform effectively incorporates DevOps methodologies via Gitbeaker and GitLab. This integration not only facilitates the automation of DT lifecycle management but also guarantees the preservation of security, scalability, and user governance, thereby enhancing the accessibility and manageability of sophisticated DT simulations.

In addition, the DTaaS platform leverages the security and access management capabilities provided by GitLab, avoiding possible vulnerabilities. Consequently, this guarantees that users are able to manage their DTs in a secure and reliable manner.

5.2 High-level design and architecture

The website consists of a single-page React application, providing simple means to interact with DTs. This architecture facilitate effortless interaction through dynamic content loading and live updating without the necessity for explicit full-page reloads.

The integration with GitLab's CI/CD pipelines, using Gitbeaker, enables triggering and monitoring directly from the interface. This ensures that DT management benefits from established software development workflows to enhance reliability and reproducibility of simulations. This close API integration with GitLab enables not only automation of these processes but also real-time feedback to be provided to the users, who can then monitor and control the simulations from an easily accessible web interface.

The component diagrams in figure 5.10 and 5.2 show how the components of the two main pages are organised. Instead, the component diagrams of the different tabs within the Digital Twins page have been added to the appendix.

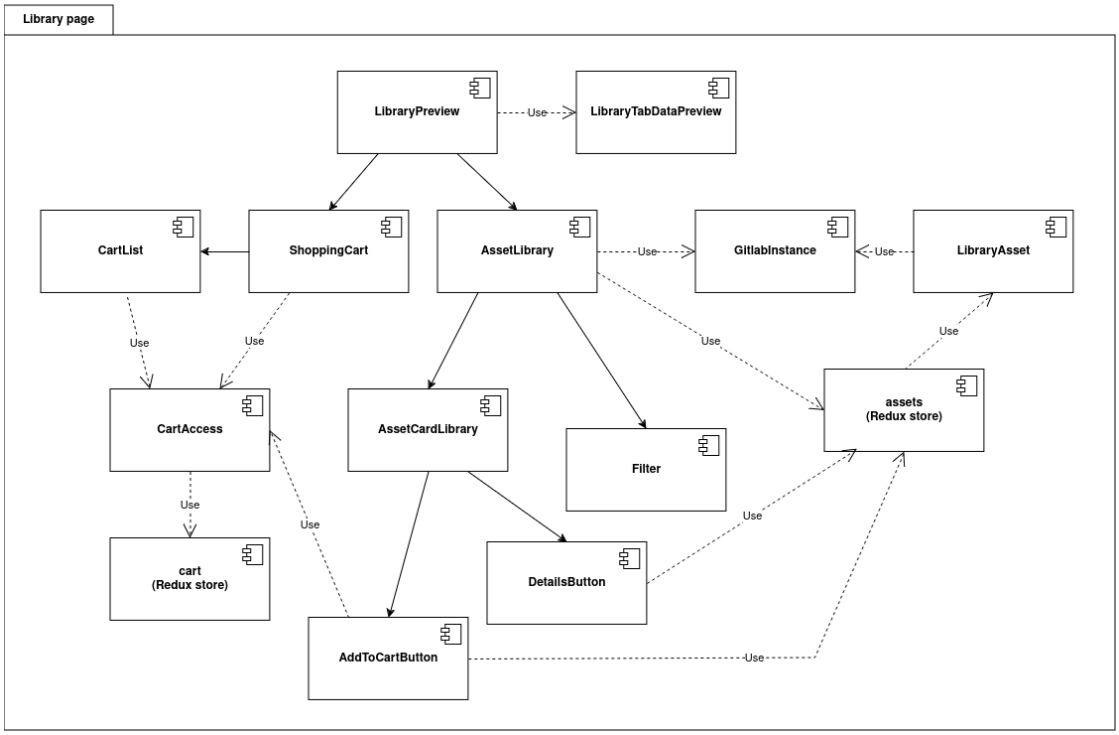

Figure 5.1: Library page - component diagram

5.3 Requirements and specifications

5.3.1 DT lifecycle

The DTaaS documentation website [5] provides a complete overview of the lifecycle related to a DT. It explains the various stages a DT undergoes in correlation with its physical counterpart, along with the way they are managed and controlled within the DTaaS platform.

As representend in figure 5.3, the DT lifecycle is composed of eight basic steps:

1. **Explore**: it comprises asset identification. This stage is vital for identifying and assessing various assets to ensure that they meet the criteria that are required to the specific DT configuration.

2. **Create**: settings are defined for the configuration of the DT. In the situation in which DT already exists, this can be skipped and, in turn, made an

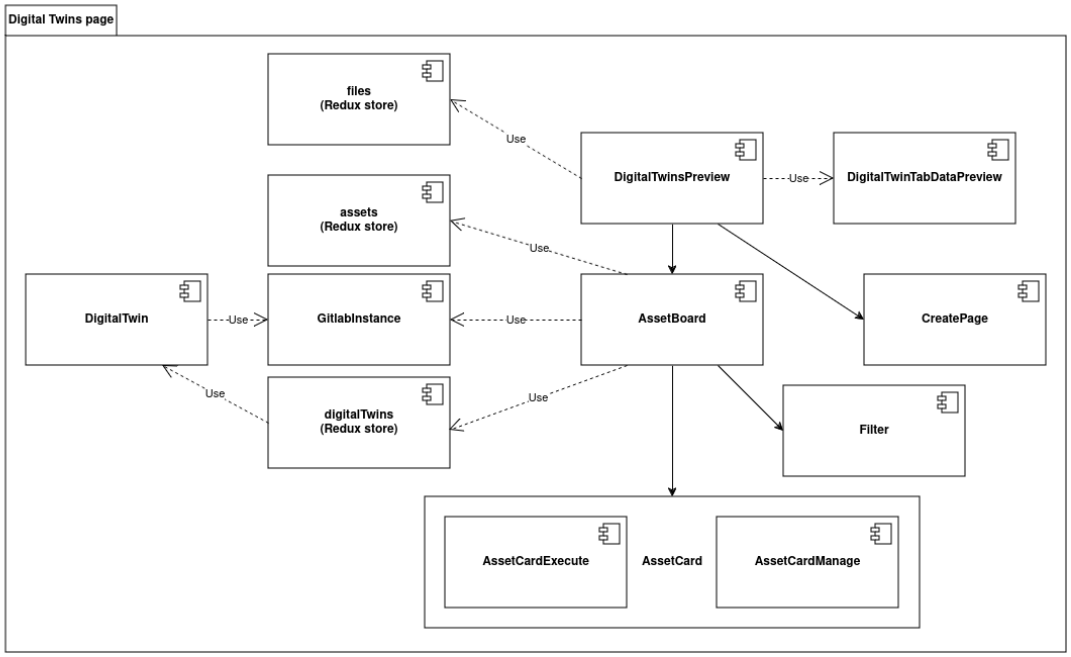

Figure 5.2: Digital Twins page - component diagram

opportunity for reuse or modification of an existing DT. It involves defining the characteristics and the parameter settings of the DT and the setting of initial conditions for the DT operation.

3. **Execute**: it refers to the actual process where the DT is executed automatically or manually, depending on the configuration of the DT. This is the stage during which the DT is supposed to simulate or monitor the performance of its physical counterpart and derive output.

4. **Analyse**: it makes output produced by the DT. This could generally be text files, visual dashboards, or any other output form. The analysis is done through interpreting the results subject to smart decision-making or modification in the configuration of the DT.

5. **Evolve**: it reconfigures DTs based on the insights derived during the analysis phase. It will perform the improvement processes, adjustments in DT to enhance functions or reach new requirements, and aims at refining DT to meet the evolving needs or conditions.

6. **Save**: it involves storing the current state of the DT to enable future recovery or resumption. This functionality is essential for preserving the DT's status and facilitating its start if needed. It allows users to pause and resume operations or recover the DT state after an interruption.

7. **Terminate**: it stops the run of the DT and possibly takes some final actions, like resource clean-up or final assessment, before finally decommissioning the DT.

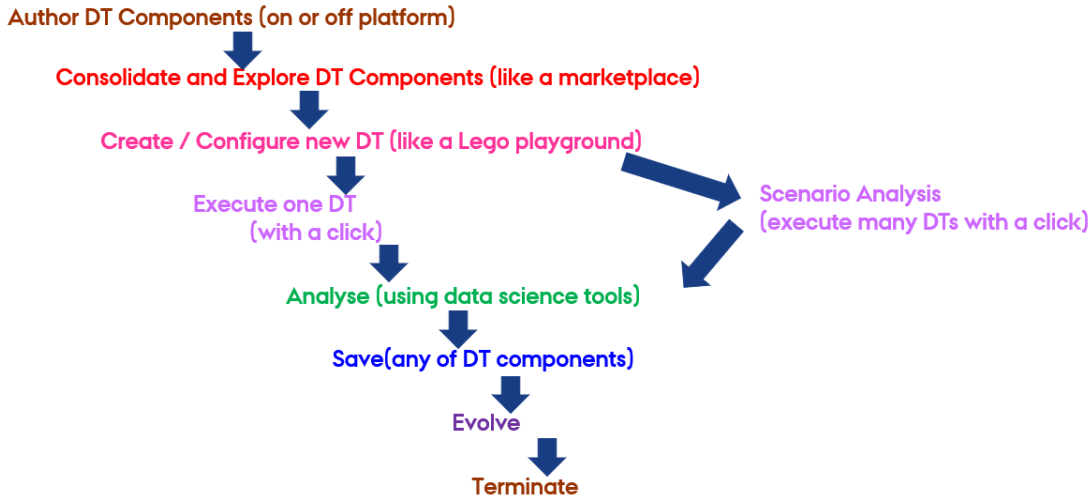

Figure 5.3: DT lifecycle [5]

The DTaaS platform intends to manage the different phases of the DT lifecycle and, therefore, provides dedicated interfaces and tools for the purpose. Figure 5.4 shows this structured kind of approach aligned with the Build-Use-Share model of DTaaS, which can facilitate the management of DTs.

For efficiently supporting the entire DT lifecycle, the DTaaS software platform needs to be embodied with a variety of indispensable functionalities. These functionalities are identified below [6]:

Authoring tools : the platform shall provide native, comprehensive tools and frameworks for authoring DT assets within the platform.

Consolidation : users require an organized system to consolidate and manage the available DT assets and authoring tools. The platform should include a discovery mechanism that facilitates easy navigation through a library of reusable assets, enhancing the efficiency of asset management and retrieval.

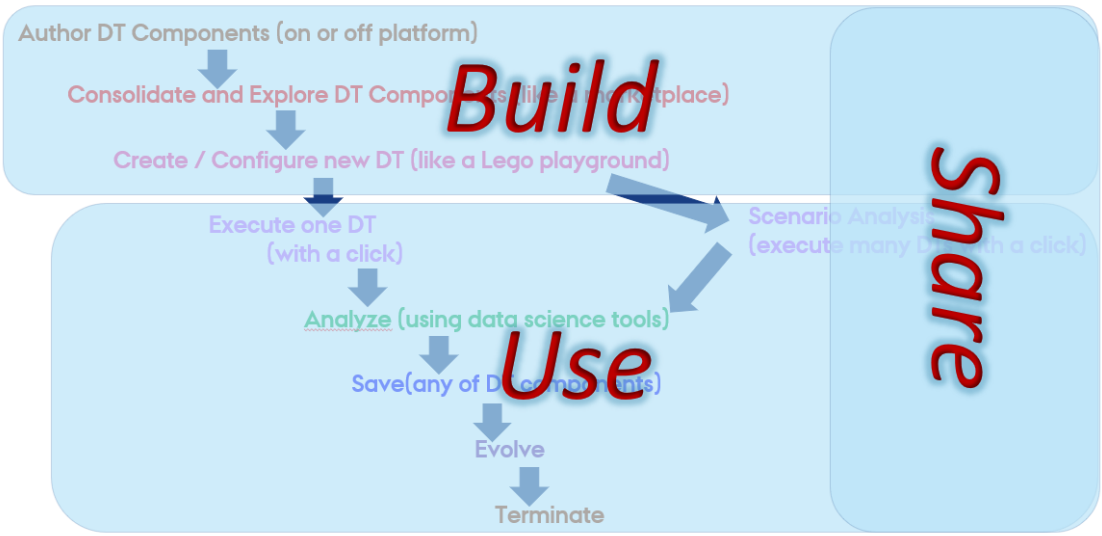

Figure 5.4: DT lifecycle and Buid-Use-Share model of DTaaS [5]

Configuration : the platform has to support the selection and configuration of DTs, including the validation of configurations.

Execution infrastructure : for DT execution, the on-demand provisioning of the computing infrastructure needs to be supported.

Exploration : the platform must allow for interactive exploration of DTs, facilitating inspection of results both in and outside the platform. This feature should also support analytic capabilities to be able to derive meaning from the DT data with an ability that helps in informed decision-making.

State management: the platform should have support to save the state of a DT during the execution phase. On-demand saving and re-spawning of DT are required for users to be able to pause, resume, or re-create the state of DT.

Sharing: users will be able to share their DTs with other users within the organization. This will help in ensuring others get involved and participate in effectively collaborating on the DT.

What-if analysis and DT evolution are not currently supported by the platform. Integration with data science tools for this purpose is beyond the scope of this work. Therefore, while what-if analysis remains a valuable feature for future iterations of the platform, it is excluded from the current implementation.

5.4 Mock-ups

The following mock-ups were the starting point for the implementation of the interfaces. All changes were based on iterative design improvements and feedback from users and stakeholders. These changes aimed to improve alignment with user needs and platform objectives.

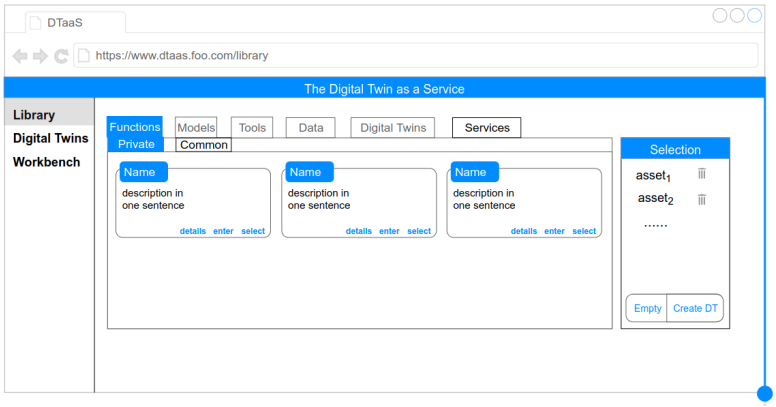

Figure 5.5: Library mock-up - Library page [45]

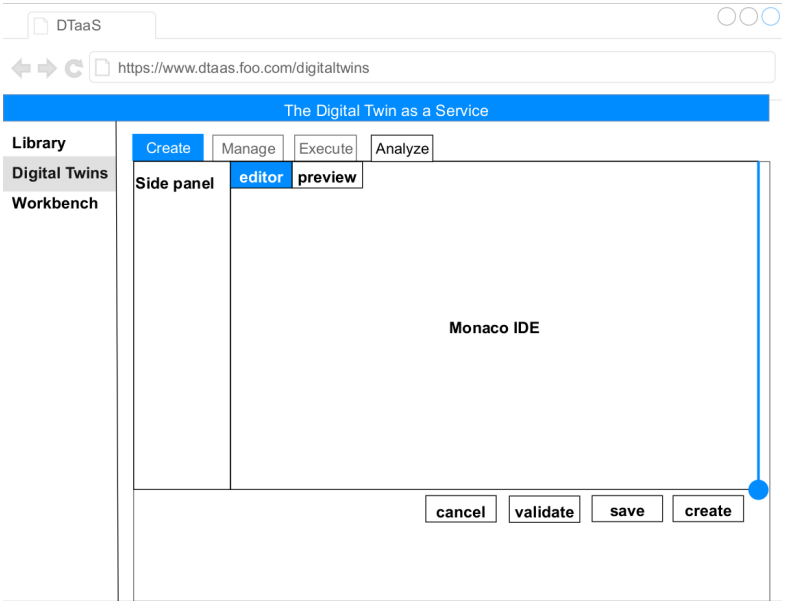

Figure 5.6: Create tab mock-up (editor section) - Digital Twins page [45]

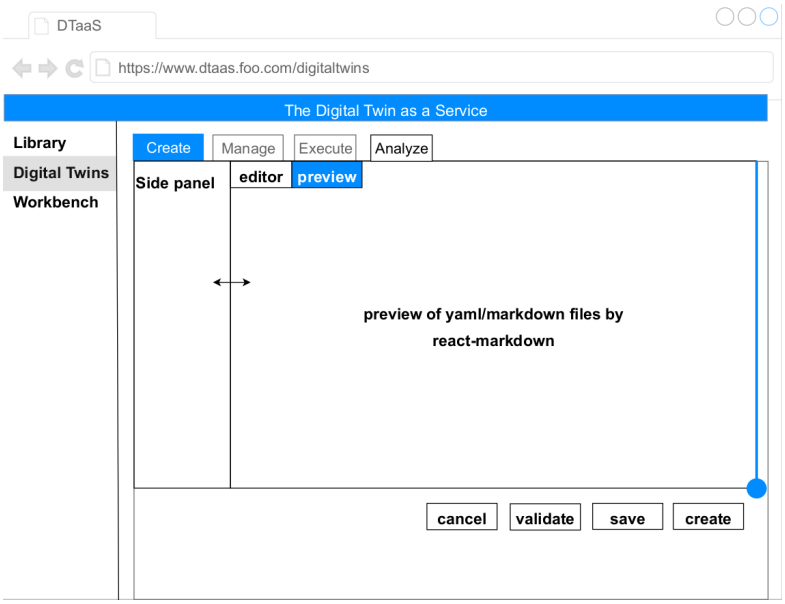

Figure 5.7: Create tab mock-up (preview section) - Digital Twins page [45]

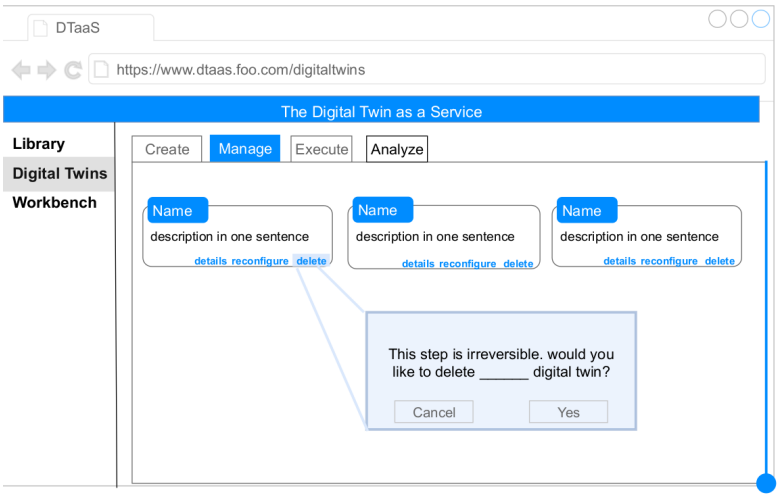

Figure 5.8: Manage tab mock-up - Digital Twins page [45]

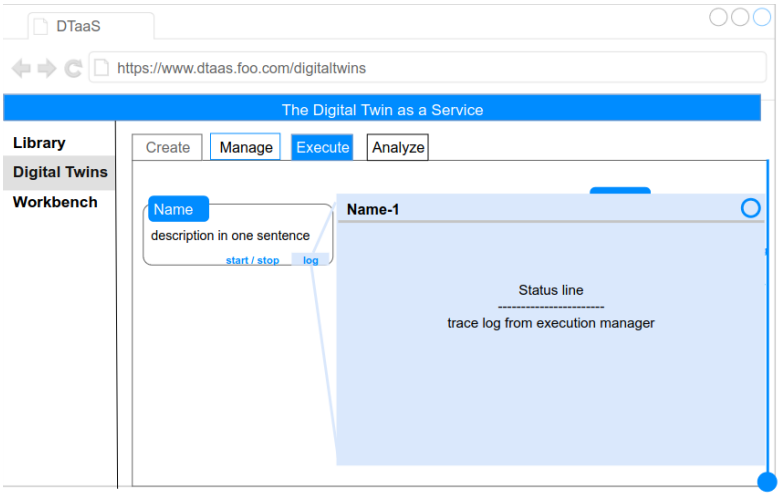

Figure 5.9: Execute tab mock-up - Digital Twins page [45]

5.5 Library page

The figure 5.10 shows the *Library* page, a central component of the system as it is based on the Build-Use-Share model. It gives users access to the database of assets that can be used to create DTs, with the ability to reconfigure them according to their needs. Assets include ready-to-use functions, data, templates, tools and DTs.

Users can view assets shared by other users as well as private assets. For each asset, there is a short description (if available) and a Details button that provides access to the contents of the associated `README.md` file, if available.

Users can add or remove assets from the cart, a dedicated area where selected assets are stored for use during the creation phase on the Create page.

5.6 Digital Twins page

The *Digital Twins* page serves as a comprehensive interface for displaying all the DTs associated with the logged-in user's GitLab profile, thereby enabling the management of their entire lifecycle. It provides a centralized view, not only listing the DTs but also equipping users with tools to monitor and manage how each DT progresses from creation through execution to analysis. It includes the following tabs: Create, Manage, and Execute.

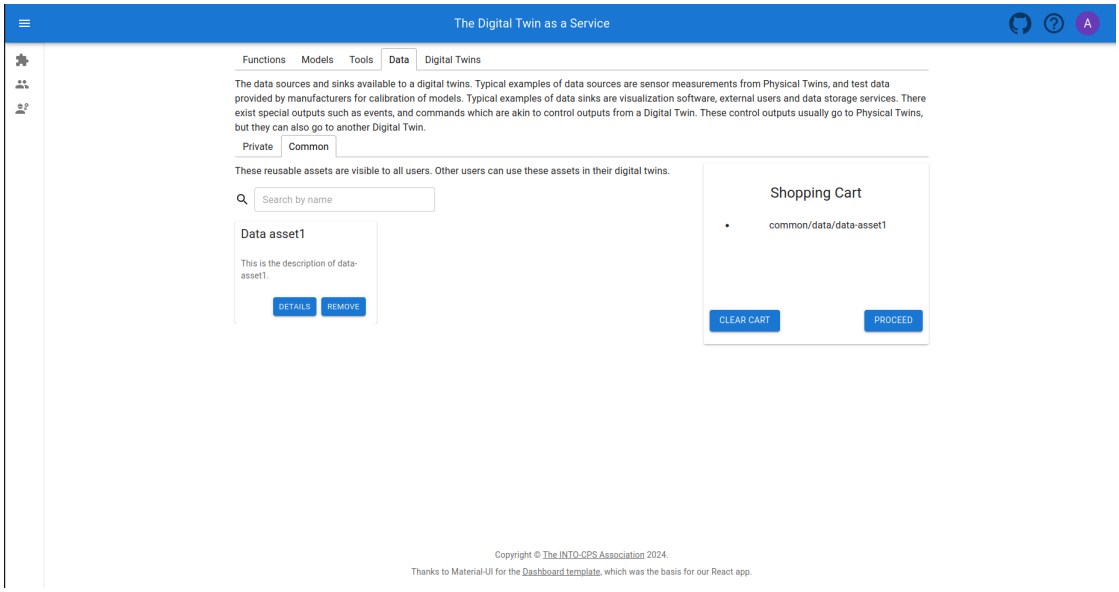

Figure 5.10: Library page

5.6.1 Create

The *Create* tab, shown in figure 5.11 allows users to create new DTs and save them to their GitLab profile. Two creation modes are provided:

- Create from scratch, for a fully customised DT.
- Reuse assets from the Library, previously added to the cart on the relevant page.

In both modes, the Create page has a sidebar divided into sections for managing DT files:

- **Description**: contains files in Markdown format, useful for describing the DT.
- **Configuration**: contains configuration files in .json, .yaml and .yml format needed to set up the DT.
- **Lifecycle**: contains all other types of files, such as bash scripts used during the pipeline to define the instructions of the different jobs, each corresponding to a phase of the DT lifecycle.

When assets have been added to the cart, additional sections are created, one for each asset selected. These contain the asset configuration files, which can be

modified to suit the new DT.

When a new DT is created, three files are automatically generated:

- `description.md`: a short description of the DT, visible on the Digital Twin's personal page and in the Library in the DT's card on the board. This file is not mandatory and can be deleted.
- `README.md`: a detailed description of the DT. This file is also not mandatory and can be deleted.
- `.gitlab-ci.yml`: the child pipeline needed to execute the DT. This file is mandatory.

Users can add, rename and delete files, which are automatically organised into their respective sections. Users can also preview the contents of files in the editor's Preview tab, which supports rendering by file format, as shown in figure 5.12.

Finally, the user can give the DT a name and click on the Save button. At this point, a confirmation dialogue will appear to complete the creation, with a snack bar informing the user of the result of the operation.

The DT created will be visible in the boards on the Digital Twins page and in the private section of the Library in the DT category. If the user has chosen to make the DT public, it will also be stored in the shared GitLab database, making it available to other users in the common section of the Library.

This configuration guarantees a high level of customisation, being fully integrated with the intended use of the Library.

5.6.2 Manage

The *Manage* tab was designed to enable the administration, modification and deleting of existing DTs, showing one card for each DT for the user's account. Each DT card shows a brief description of the DT and allows three main actions: *details*, *reconfigure*, and *delete*.

As presented in the image 5.13, the details button opens a pop-up displaying the `README.md` file, which includes the complete description of the DT, thus explaining the goals and configuration of the twin. The correct markdown formatting is displayed, as well as images, tables and mathematical formulas.

The reconfigure button opens the Reconfigure Dialog shown in figure 5.14, a pop-up window that allows a user to view and edit the files in the corresponding

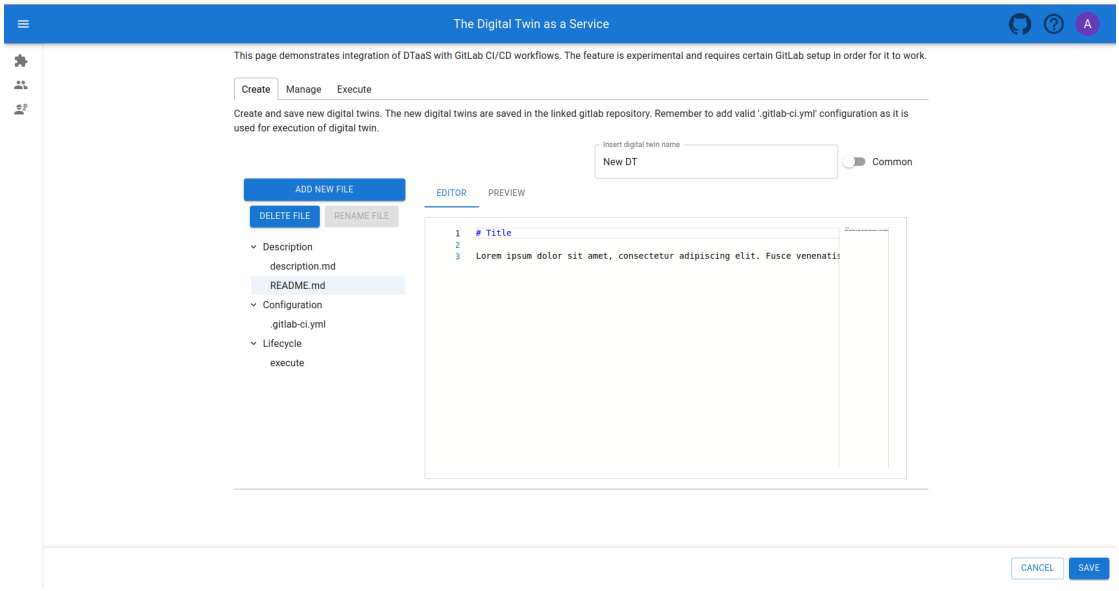

Figure 5.11: Create tab - Digital Twins page

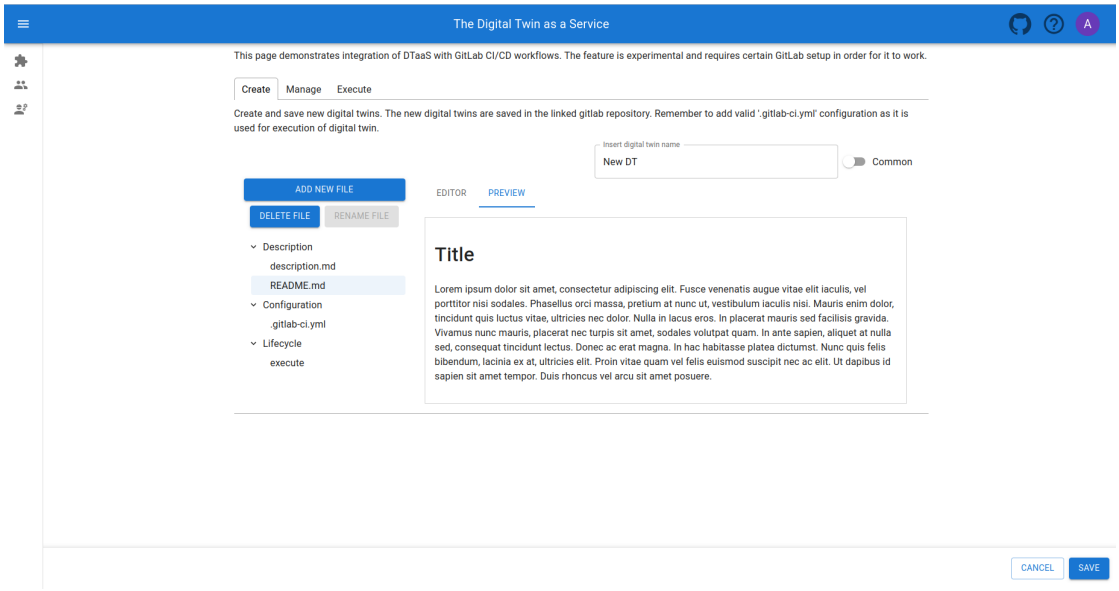

Figure 5.12: Create tab showing the `README.md` preview - Digital Twins page

GitLab folder of the DT. This corresponds to the Editor shown in the Create tab, but with functionality only for editing existing files.

The user can select a file that requires editing, which is then displayed in the Editor tab. It is possible to edit one or more files. The Save button save only those

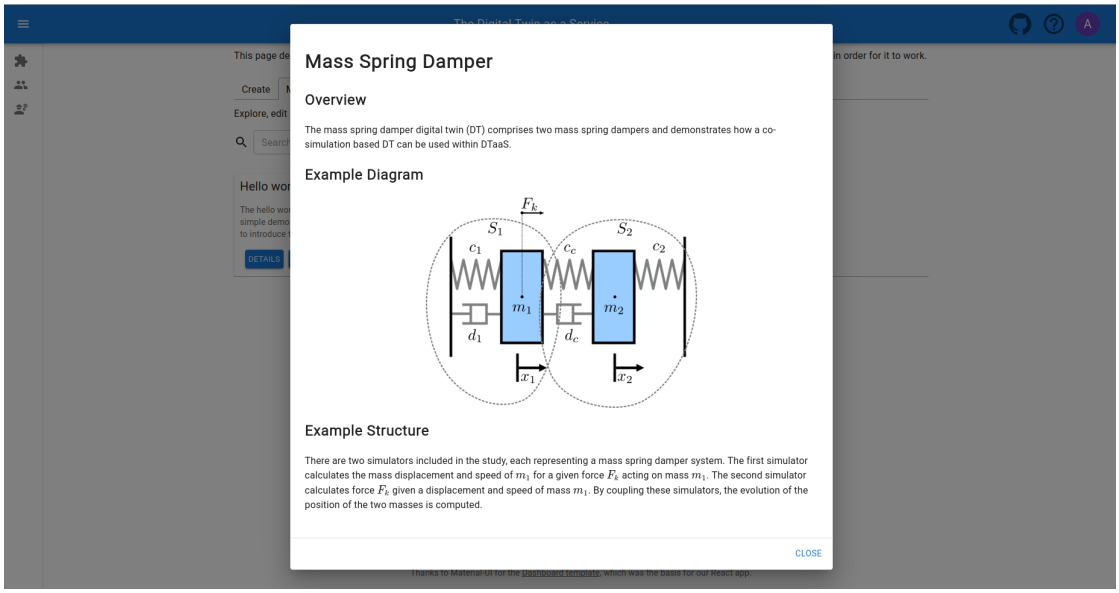

Figure 5.13: Details for the Mass Spring Damper DT - Manage tab - Digital Twins page

files that have been modified in the editor, thus avoiding unnecessary saving of all files and reducing the load on the system.

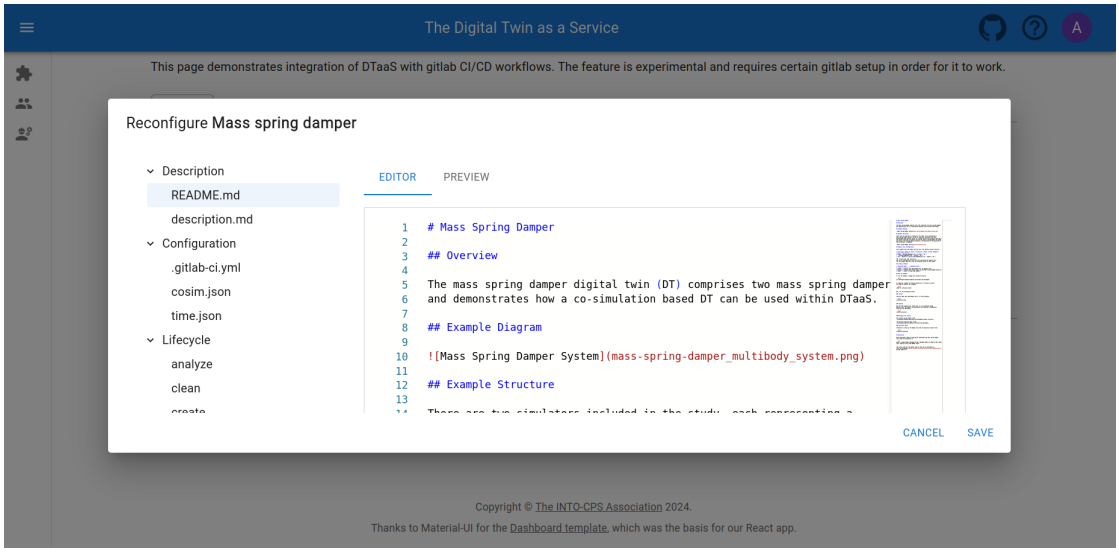

Figure 5.14: Reconfigure for the Mass Spring Damper DT - Digital Twins page

The delete button is used to remove a DT from the workspace with all its associated data.

Both reconfiguration and deletion are permanent processes, so there is a confirmation dialog box that appears advising the user to confirm their intention to reconfigure or remove the DT. This proves to be a necessary safeguard against data loss due to accidents and further ensures that users are duly informed of the implications of their action before they proceed.

5.6.3 Execute

As depicted in figure 5.15, the execute tab shows one card for each DT for the user's account. It displays the DT name and a brief description, along with start/stop execution options and a log button. When opening the page, the log button is disabled.

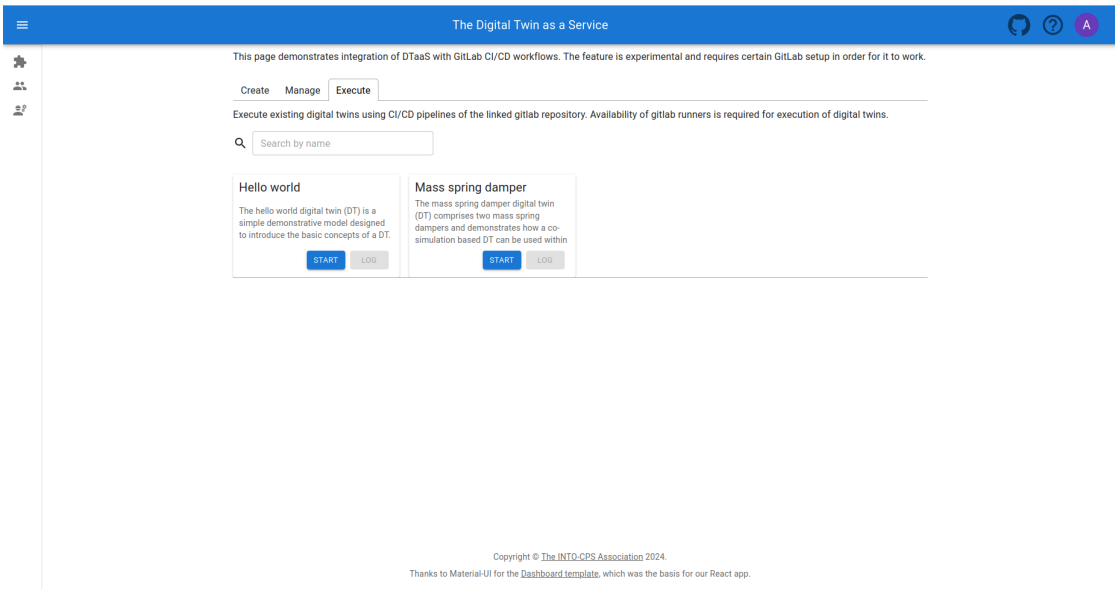

Figure 5.15: Execute tab - Digital Twins page

The Snackbar component is used to update the user on the status of operations or to report any errors, as shown in figures 5.16 and **??**.

In addition, when the pipeline is in loading status, as depicted in figure 5.16, the user has another visual feedback offered by the circular progress within the relevant card. During this process, the button initially used to start the pipeline changes state, allowing the user to stop the operation by clicking on it, with the label updated to 'Stop'.

At the end of the pipeline execution, the log button is enabled. Upon clicking on it, the corresponding LogDialog is opened, which displays the comprehensive

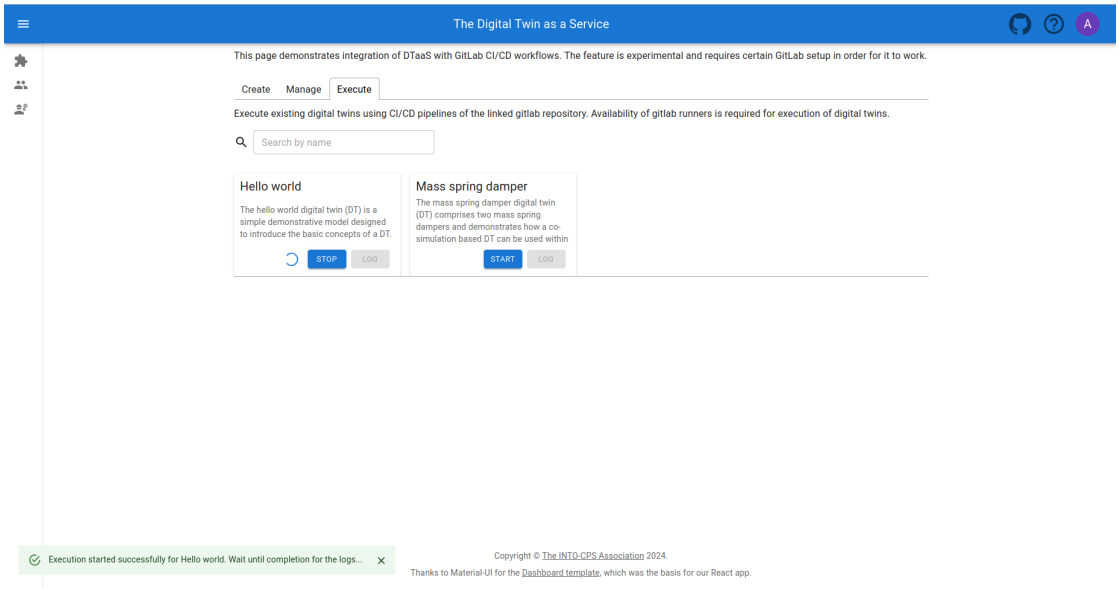

Figure 5.16: Start pipeline - Execute tab - Digital Twins page

logs of all the constituent jobs within the pipeline.

Some advantages of this approach include centralized management, which enables users to manage multiple DTs through a single interface. It facilitates real-time control, allowing for immediate command over the execution of the DT. Moreover, operation logging enhances transparency and debugging, providing users with trace logs that improve troubleshooting and performance analysis.

Chapter 6

User testing and results

A general introduction on acceptance testing was provided in Chapter 3. This chapter will describe how the acceptance tests were conducted, presenting and analysing the results.

6.1 User acceptance tests

User Acceptance Testing (UAT) is a specific type of acceptance testing. It is performed at the end of the development phase from the end-user perspective. It focuses on verifying that a system meets the requirements and delivers business value to its intended users [46].

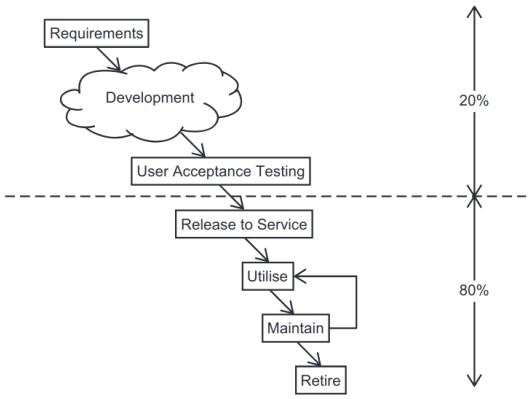

Figure 6.1: UAT in the lifecycle of a software product [47]

Although the software has already undergone unit and integration testing, once it has reached this stage it may still not meet business requirements. In fact, the former type of testing is designed to verify that the software meets technical

requirements, whereas UATs focus on the system as a whole and in its context of use [47]. Hence the importance of involving end users, who can provide their perspective in identifying any gaps or discrepancies between implemented functionality and actual needs. The results obtained allow defects to be identified and mitigated before the software is released [46].

6.1.1 Process

The approach outlined by Hambling and van Goethem [47] was adopted during the UAT stage of this thesis work.

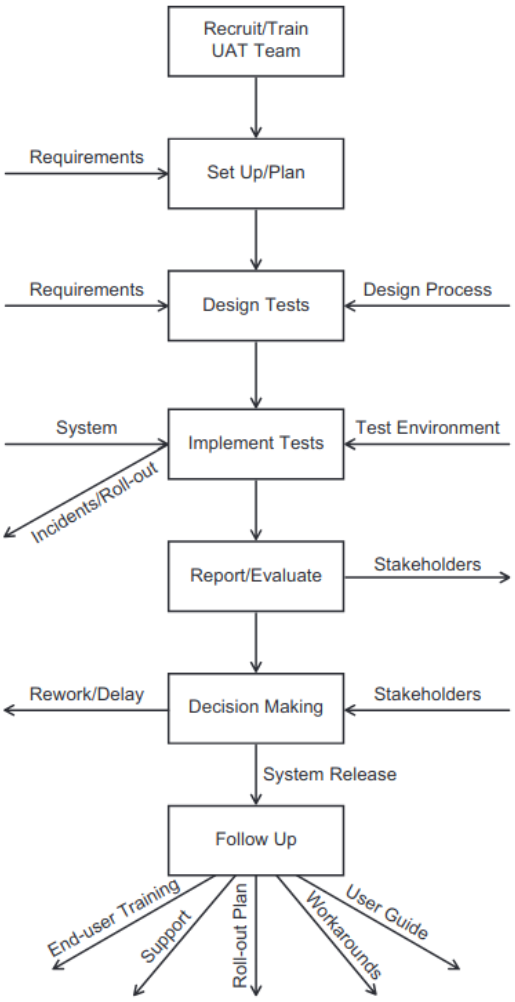

Figure 6.2: The process of UAT [47]

The process involves the following steps:

1. *Recruit/Train UAT Team*: recruit the team responsible for planning and executing the AUT process and train the testers.
2. *Set Up/Plan*: define the goals and objectives of the UAT, establish a basis for testing, and organise the next steps.
3. *Design Tests*: derive tests from the requirements in order to achieve the main objective.
4. *Implement Tests*: collect data on the status of the system for analysis and potential resolution, to provide a working test environment for the effective execution of all tests.
5. *Report/Evaluate*: assess whether the acceptance criteria have been met and, if not, assess the extent of the failure and identify potential solutions.
6. *Decision Making*: evaluate the results of the UAT and the deployment decisions made.

6.1.2 Criteria

As outlined by Hambing and van Goethem in their work [47], UAT is based on three key criteria:

1. **Organised method**: tests are carried out according to a formal and well-structured process that ensures systematic execution.
2. **User-centricity**: the main objective is to ensure that the system fully meets the needs of the users and complies with the requirements.
3. **Compliance with acceptance criteria**: it is essential that the system meets the standards that the users consider appropriate for their approval.

6.2 Steps of the UAT process in this thesis

This section describes the steps of the UAT process in the context of the thesis work.

6.2.1 Recruit/train UAT team

The AUT team consisted only of the author of this thesis. Therefore, there was no need for recruitment, but a study of the AUT process. On the other hand, the AUT testers were end-users, so it was necessary to provide them with an overview of the software application and its business intent. The testers' tasks included executing

the test scripts, highlighting any problems they encountered and providing feedback on the user experience.

6.2.2 Set up/plan

It was necessary to identify business requirements in order to determine what had to be tested during the UAT. A business requirement describes what the system should do in clear and simple language. They should collectively represent the business goals and objectives. In this way a test basis was created, i.e. the documentation used to generate the tests.

Reference	Description
U_01	A user can view assets, both private and common, through the Library page
U_02	A user can add one or more assets to the shopping cart
U_03	A user can create a new DT
U_04	A user can read the details (`README.md` file) of both their DTs and library assets
U_05	A user can edit the files of their DTs
U_06	A user can delete their DTs
U_07	A user can start execution of their DTs
U_08	A user can stop the execution of their DTs
U_09	A user can view logs, i.e., the result of the execution of their DTs
S_01	The system must allow the execution of DTs created by users
S_02	The system must not allow execution of DTs deleted by users

Table 6.1: Business requirements

The business intent is the fundamental purpose for which a software product is created, representing the goals and benefits to achieve once the system is operational. It serves as a reference point for all stakeholders during the development and UAT phase. In the case of end-users, it helps clarify to them the goal to be achieved and the value to be created [47].
The business intent of this thesis project was to simplify the management and use of DTs and make them accessible through an integrated software platform based on DevOps principles. Its main objective was to meet the needs of both technical and non-technical users through automated tools and intuitive interfaces that enable the creation, use and sharing of DTs throughout their lifecycle.

Acceptance criteria are used to decide when to stop testing. These are a set of conditions that functionality or software must meet to be considered complete and acceptable. Ideally, the system should work correctly, have no defects and be ready for release. However, if defects are identified during the AUT process, they can be rated according to their criticality and severity. A distinction can also be made between essential, important but not essential and cosmetic (improving the user experience) functionality. These factors can then be evaluated at the end of the AUT to make a more realistic decision about the release of the software product [47].
In this case, the planning involved covering all the required functionalities through testing and then evaluating the potential defects identified.

6.2.3 Design tests

Coverage is the key factor to be used in assessing the effectiveness of tests, which is not actually related to the number of tests per se. In particular, each requirement must have at least one associated test, and critical tests may have more than one. Documenting the tests makes it possible to verify their correctness and reduce failures due to errors in the tests themselves [47].

The first step was to identify test cases describing inputs, expected outputs, preconditions and postconditions. From these, test scripts were written. These scripts translate the test cases into detailed instructions to follow in order to validate or invalidate the tests. The test scripts used have been included in the appendix of this thesis.

6.2.4 Implement tests

This phase involves planning the tests and recording the results obtained, which will be used to evaluate the system [47].

First, the order of the tests was planned to optimise the process. The testers were given the test scripts and carried out the tests according to the instructions, with the task of noting any difficulties and reporting incidents if the results differed from those expected.

6.2.5 Report/evaluate

Each tester ran the test scripts until the requirements were covered so that the acceptance criteria were met. No incidents or defects were found, so no changes were required.

6.2.6 Decision making

The feedback received from the testers was positive and the business intent was considered to have been achieved. The execution of the tests did not reveal any particular defects or results that differed from those expected. The release can therefore be considered safe.

6.3 SUS questionnaires

With the aim of obtaining usability feedback, each user was asked to complete a System Usability Scale (SUS) questionnaire at the end of the usability tests.

The SUS is an evaluation tool that measures the usability of a system from the user's point of view. It is a simple method to administer as it consists of only 10 questions, added to the appendix of this thesis, using a scale from 1 ("strongly disagree") to 5 ("strongly agree"). It is able to provide a quantitative measure of usability by giving users the opportunity to express their opinion, thus helping to understand the weaknesses of the system [48].

The SUS score is calculated by converting the responses into an overall score between 0 and 100. The average SUS score is 68, and a score of 70 or more is considered good, while a score of 80 or more is considered excellent [48].

6.3.1 SUS results

The results of the SUS questionnaires are shown in the table 6.2:

Tester ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SUS Score
1	5	1	5	1	5	1	5	1	5	2	95
2	4	1	4	2	5	1	5	1	4	1	77.5
3	4	1	4	1	4	2	5	1	4	1	77.5
4	5	2	4	2	5	1	5	1	5	2	85
5	5	1	5	1	4	1	5	1	4	1	87.5

Table 6.2: SUS results

The average SUS score is 84.5. This indicates excellent usability according to SUS conventions [48].

Chapter 7

Conclusion

This thesis presents a comprehensive framework that integrates DevOps practices for the management of DTs. The solution enabled the streamlining of lifecycle management, thus allowing users to create, monitor, and manage DTs in an effective manner. The implementation of user-friendly interfaces and automated CI/CD pipelines enhanced accessibility for both technical and non-technical users, thereby improving usability and efficiency. Moreover, the user testing phase validated the framework's effectiveness. This work has demonstrated the potential of combining DevOps methodologies with DTs to address industry challenges and establish a foundation for future developments.

7.1 Evaluation of the thesis objectives

This section analyses whether the objectives set out in the introduction have been achieved in the course of the thesis work. The objectives are then reported again, followed by an evaluation.

- **Design of an automated infrastructure**: the proposed framework successfully implemented a fully automated infrastructure using GitLab CI/CD pipelines. The design minimised manual intervention by integrating parent-child pipelines.
- **DevOps integration**: DevOps practices were successfully integrated into the framework, with a particular focus on CI/CD pipelines and automation. The use of Gitbeaker simplified interactions with GitLab APIs.
- **Lifecycle management of DTs**: the system provided comprehensive support for the entire lifecycle of DTs. Users are able to create DTs either from scratch or by utilising the library, as well as modify and delete them directly through

the web interface. Additionally, the platform allows users to initiate and stop the execution of DTs. Key features such as file management and pipeline automation were validated, ensuring that all aspects of the DT lifecycle could be efficiently managed.

- **Intuitive user interfaces**: user-friendly web interfaces were developed with the objective of enabling interaction with DTs even for non-technical users. The focus was on simplicity and usability, addressing the need for intuitive interaction and minimising the learning curve. Feedback from user acceptance tests confirmed that the interfaces met the established usability expectations.

- **Evaluation with end users**: The user acceptance testing process validated the framework's functionality and usability. End-user feedback, collected via SUS questionnaires, demonstrated high satisfaction with the system's ease of use and efficiency. Areas for future improvement were also identified, providing a roadmap for further development.

7.2 Future work

In terms of future improvements, several areas offer promising opportunities.

Firstly, the testing was conducted with a limited dataset, comprising a small number of DTs and assets in the Library. For this reason, it would be beneficial to expand the scope of testing to include larger datasets and more use cases. This would help validate the scalability and robustness of the system, ensuring its capacity to handle increased data volumes and complex interactions in real-world scenarios.

Furthermore, the incorporation of AI-driven decision-making could significantly enhance the platform's functionalities. An *Analyze* page could be incorporated in the Digital Twins page, where users could leverage AI to analyse the data associated with their DTS. This could provide valuable insights, predictions, and recommendations based on the behavior and performance of the DTs, helping users make more informed decisions.

Moreover, users have proposed the addition of a feature enabling the creation of individual assets directly within the Digital Twins page. This proposed enhancement would provide greater flexibility in the creation and sharing of assets, making the platform more adaptable to the diverse needs of its users.

7.3 Personal outcomes

This thesis was a great opportunity for both technical and personal growth.

The project allowed me to deepen my understanding of DevOps principles and gain significant hands-on experience with key technologies, such as GitLab CI/CD. One of the most rewarding aspects was integrating the DevOps framework with user-friendly interfaces, automating the lifecycle of DTs.

In addition, the testing phase was particularly valuable as it gave me a deeper understanding of user experience and the importance of gathering user feedback. Conducting user acceptance tests and interpreting the results sharpened my skills in evaluating software from an end-user perspective.

On a personal level, working on this project allowed me to refine my ability to manage complex tasks and collaborate with supervisors and end users. It has also highlighted the importance of clear communication. As well as broadening my technical skills, this thesis has given me the confidence to take on large projects and the ability to think critically about the integration of different technologies and methodologies.

Appendix A

Component diagrams

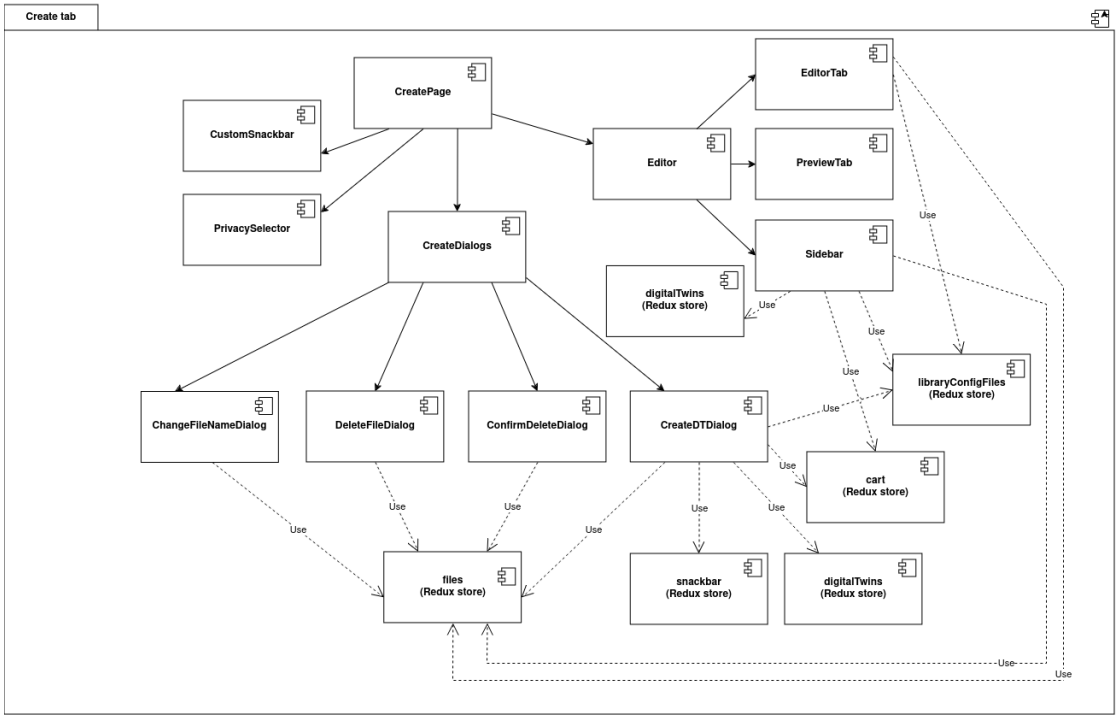

Figure A.1: Create tab subsystem

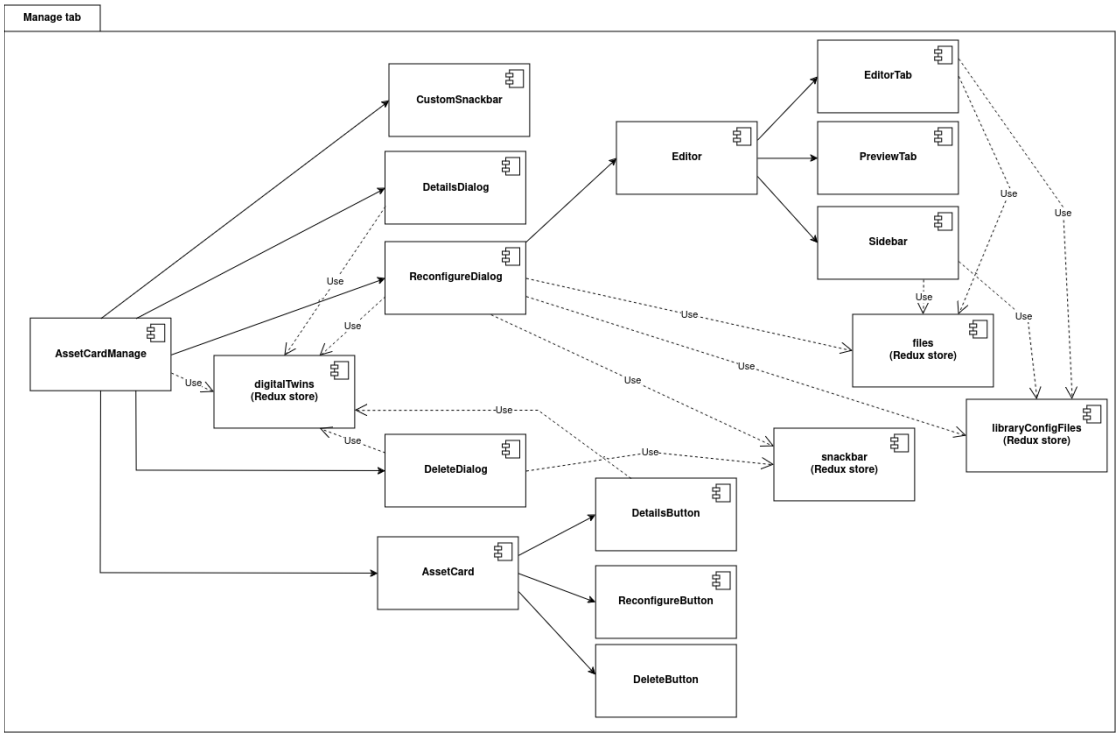

Figure A.2: Manage tab subsystem

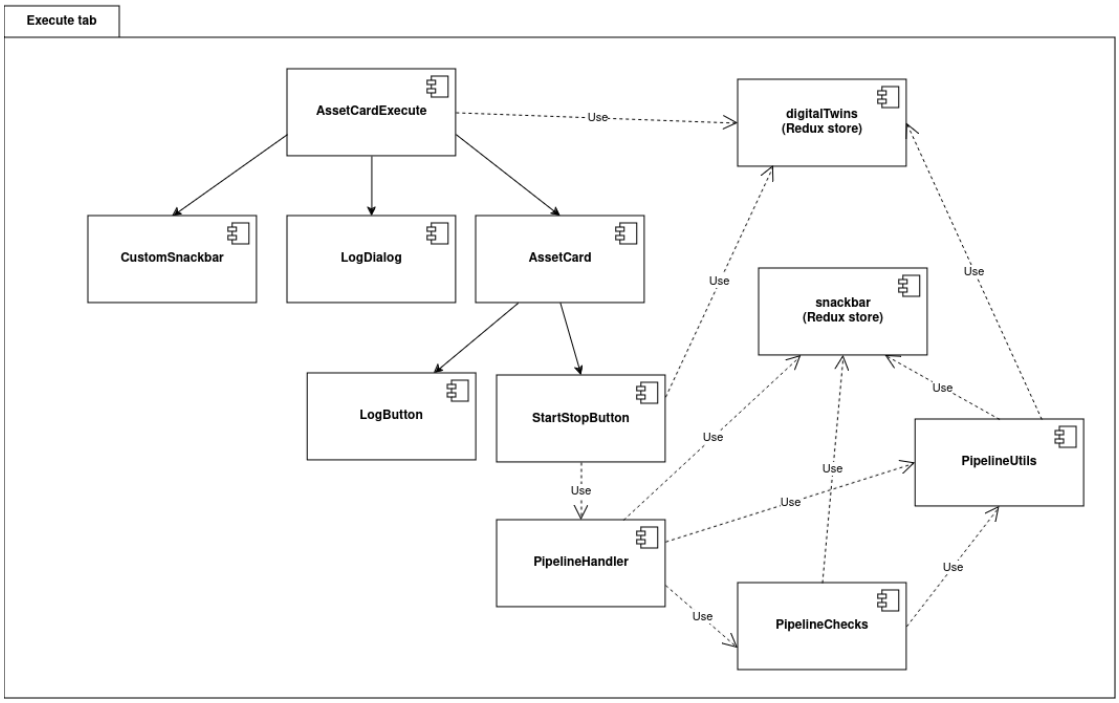

Figure A.3: Execute tab subsystem

Appendix B

Sequence diagrams

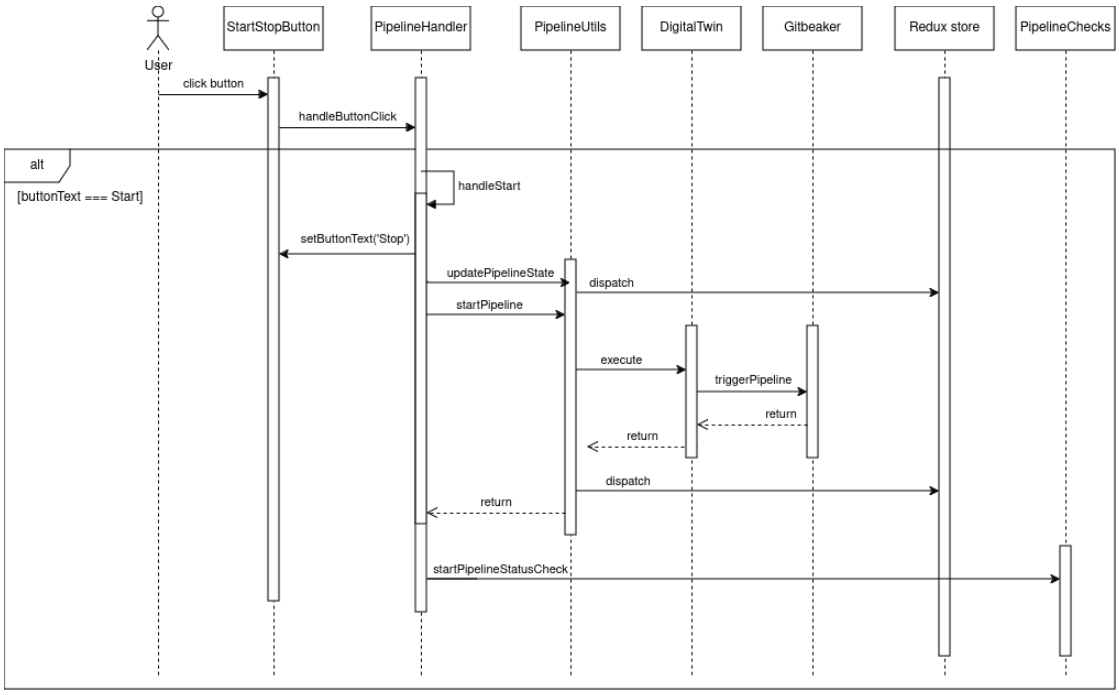

Figure B.1: User start execution of a DT

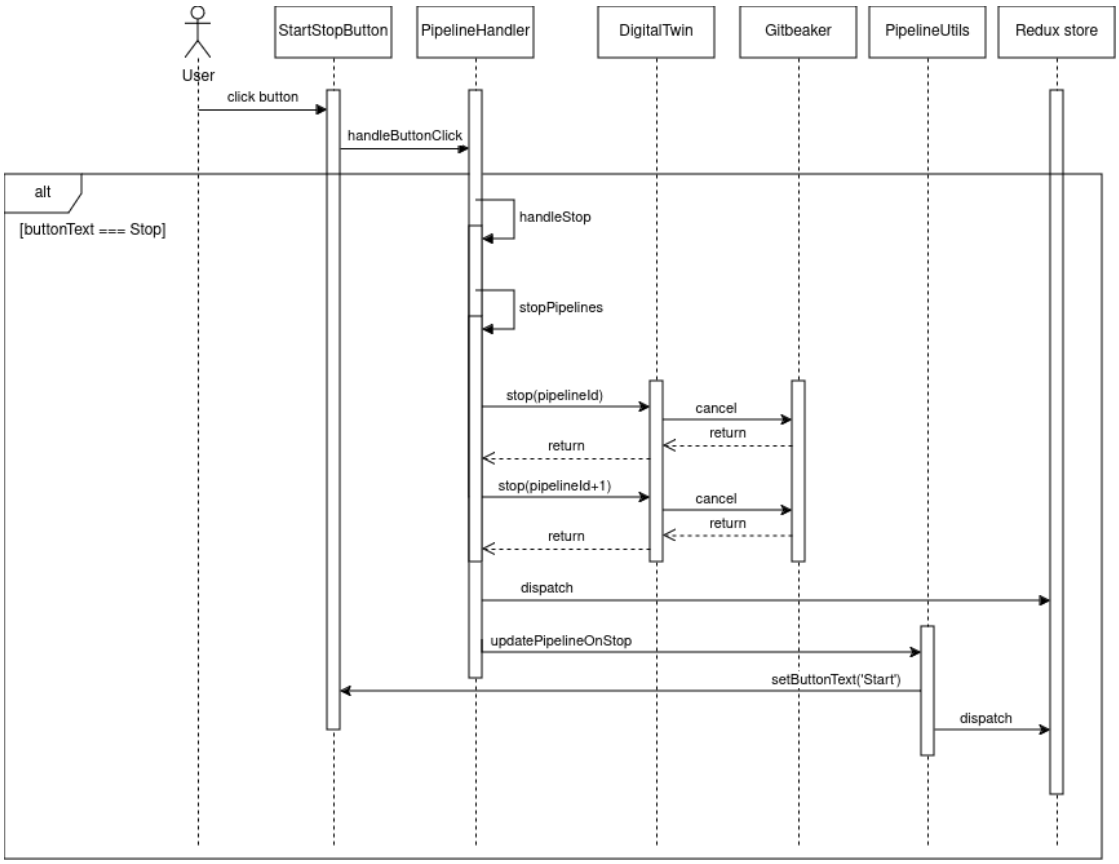

Figure B.2: User stop execution of a DT

Appendix C

UAT - Test scripts

Assets visualization in Library page	
Requirement ref.	U_01
Purpose	Verify that a user can view both private and common assets on the Library page.
Precondition	User must be logged in.
Test data	A set of private and common assets associated with the user account.
Process steps	1. Log into the system with valid user credentials. 2. Navigate to the Library page from the Workbench. 3. Verify that all private and common assets are displayed on the Library page. 4. Check that the details for each asset are correct, such as asset type (function, model, tool, data or digital twin), asset name and asset privacy (private or shared).

Successful creation of a DT	
Requirement ref.	U_03 and S_01
Purpose	Verify that a DT is successfully created.
Precondition	User must be logged in. The Create tab is displayed.
Test data	Valid file set in the sidebar. DT name: *Example-DT*.

Process steps	1. Open the Create tab. 2. Prepare the files in the sidebar and ensure there are no empty files. 3. Enter a valid name for the DT (*Example-DT* for this test). 4. Set the DT to Private. 5. Verify that the *Save* button is enabled after entering the DT name. 6. Click on the *Save* button. 7. Confirm the action in the pop-up dialog. 8. Verify that the system confirms the successful creation of the DT. 9. Navigate to the Manage tab and verify that the DT *Example-DT* appears in the board. 10. Navigate to the Execute tab and verify that the DT *Example-DT* also appears in the board. 11. Navigate to the Library page and verify that the DT *Example-DT* appears in the private section of the Digital twins tab. 12. Ensure that the corresponding section for the DT is added to the `.gitlab-ci.yml` file (parent pipeline) in the GitLab user account. 13. **Alternative Flow (Common DT):** (a) Set the DT to Common. (b) Verify that the DT *Example-DT* appears in the common section of the Digital twins tab.

Add assets to shopping cart	
Requirement ref.	U_02
Purpose	Verify that a user can add one or more assets to the shopping cart.
Precondition	User must be logged in.
Test data	A set of assets visible in the Library page.
Process steps	1. Navigate to the Library page from the Workbench. 2. Choose an asset and click *Add*. 3. Verify that the selected asset appears in the shopping cart. 4. Repeat the process for multiple assets and confirm that all are added to the cart.

Initial setup of the sidebar in the Create tab	
Requirement ref.	U_03
Purpose	Verify that the sidebar always shows three sections (Description, Configuration, Lifecycle) with default files when the Create tab is opened.
Precondition	User must be logged in. User opens the Create tab with no assets added to the cart.
Test data	None required (default setup).
Process steps	1. Navigate to the Digital Twins page from the Workbench. 2. Verify that the Create tab is opened. 3. Verify that the Sidebar contains the following sections: • **Description**: includes `README.md` and `description.md`. • **Configuration**: includes `.gitlab-ci.yml`. • **Lifecycle**: empty by default. 4. Confirm that the default files are editable and correctly positioned in their respective sections.

Display assets files in the sidebar from the cart	
Requirement ref.	U_03
Purpose	Verify that the configuration files of the assets added to the cart are displayed in the sidebar under separate sections when the Create tab is opened.
Precondition	User must be logged in. User has added one or more assets to the cart before opening the Create tab.
Test data	Assets with configuration files (.json, .yml, .yaml) in the cart.
Process steps	1. Add assets to the cart from the Library page. 2. Open the Digital Twins page. 3. Verify that the sidebar contains the initial setup with the default files. 4. Verify that the sidebar contains a separate section for each asset, named *<asset-name> configuration*. 5. Confirm that the configuration files of each asset are editable and listed under the respective sections.

Create DT with empty files	
Requirement ref.	U_03
Purpose	Verify that the system prevents the creation of a DT with empty files.
Precondition	User must be logged in. The DT Create tab is displayed, with at least one empty file.
Test data	An empty file in the sidebar and the DT name.
Process steps	1. Enter the DT name to enable the *Save* button. 2. Click *Save* to start trying to create the DT. 3. Confirm the action in the pop-up dialog. 4. Verify that the DT is not created and an error message prompts the user to edit the empty files, displaying their names.

Add a new file to the sidebar	
Requirement ref.	U_03
Purpose	Verify that a new file can be added and displayed in the correct section of the Sidebar.
Precondition	User must be logged in. The DT Create tab is displayed.
Test data	File names and extensions (.md, .json, .yaml, .yml, others).
Process steps	1. Click *Add new file*. 2. Digit the file name (including its extension). 3. Verify that the file appears in the correct section of the Sidebar: • **Description**: Files with *.md*. • **Configuration**: Files with *.json*, *.yaml*, or *.yml*. • **Lifecycle**: All other file types. 4. Try adding a file with the same name as an existing file in the Sidebar. 5. Verify that the system displays an error message and does not add the file.

Delete a file from the sidebar	
Requirement ref.	U_03
Purpose	Verify that a file can be deleted from the sidebar, except `.gitlab-ci.yml`.
Precondition	User must be logged in. The Create tab is displayed.
Test data	A list of files, including `.gitlab-ci.yml`.
Process steps	1. Select a file in the sidebar and click *Delete file*. 2. Confirm the deletion in the pop-up dialog. 3. Verify that the file is removed from the Sidebar. 4. Check that the *Delete* button is not displayed in case of `.gitlab-ci.yml`.

Rename a file in the sidebar	
Requirement ref.	U_03
Purpose	Verify that a file can be renamed, except default file names (`README.md`, `description.md`, `.gitlab-ci.yml`).
Precondition	User must be logged in. The Create tab is displayed with at least one file in the Sidebar added by the user.
Test data	File names, including default ones.
Process steps	1. Select a file in the Sidebar and click *Rename file*. 2. Enter a new valid name and confirm. 3. Verify that the file is renamed successfully. 4. Check that the *Rename file* button is not displayed in case of the default files. 5. Try renaming a file with an already existing name in a section between Description, Configuration and Lifecycle and verify that the system prevents this with an error message.

View Preview in Editor	
Requirement ref.	U_03
Purpose	Verify that the user can view a formatted preview of a file by clicking on the *Preview* tab.
Precondition	User must be logged in. The Create tab is displayed.
Test data	Files with different formats (e.g., .md, .json).
Process steps	1. Select a file from the sidebar to open it in the Editor. 2. Enter the file content in the *Editor* tab. 3. Click on the *Preview* tab. 4. Verify that the file is displayed in the appropriate formatted view based on its type.

Read DT and library asset details	
Requirement ref.	U_04
Purpose	Verify that a user can read the details (`README.md` file) of their DTs and library assets, or be informed if the file does not exist.
Precondition	User must be logged in. The DTs and assets may or may not have associated `README.md` files.
Test data	DTs and Library assets with valid `README.md` files containing formatted content (e.g., text, images, tables). DTs and Library assets without README.md files.
Process steps	1. Click *Details* button of a DT or library asset from the board. 2. Verify the following scenarios: (a) If the `README.md` file exists: • A pop-up opens displaying the file's content, formatted correctly. • Verify that text, images, and tables (if present) are rendered properly. (b) If the `README.md` file does not exist: • A pop-up opens with the message: "There is no README.md file."

Edit the files of a DT	
Requirement ref.	U_05
Purpose	Verify that a user can edit the files of their DTs.
Precondition	User must be logged in. The DT to be modified must already exist and contain editable files.
Test data	A DT with some files. Library assets with configuration files, if applicable.
Process steps	1. Navigate to the Manage tab and select the DT to modify. 2. Click the *Reconfigure* button to open the Editor. 3. Verify that the files associated with the DT are displayed in the Editor, similar to the Create tab. 4. Click on a file in the Editor to view its content. 5. Verify that the content of the file is displayed correctly in the Editor. 6. Modify the content of the file as needed. 7. If there are assets from the Library, verify that their configuration files are displayed in the Editor. 8. Modify the configuration files of the library assets if needed. 9. Click on the *Preview* tab to view the formatted preview of the file. 10. Verify that the file is rendered correctly. 11. After making changes, click the *Save* button and confirm the action in the pop-up dialog. 12. Verify that the DT is successfully updated with the new content.

Delete a DT	
Requirement ref.	U_06 and S_02
Purpose	Verify that a user can delete their DTs.
Precondition	User must be logged in. The DT to be deleted must exist and be visible in the Manage tab.
Test data	A DT with valid files and configuration.
Process steps	1. Navigate to the Manage tab. 2. Click the *Delete* button of the DT to delete it. 3. Confirm the deletion action in the pop-up dialog. 4. Verify that the DT is removed from the list in the Manage tab. 5. Check that the DT no longer appears in the system. 6. Check that the section for the deleted DT is removed from the `.gitlab-ci.yml` file (parent pipeline) in the GitLab user account. 7. If the DT was common, check that it was also deleted from the Library.

Start execution of a DT	
Requirement ref.	U_07
Purpose	Verify that a user can start the execution of their DTs.
Precondition	User must be logged in. The DT must exist and be visible in the Execute tab.
Test data	A DT that can be executed.
Process steps	1. Navigate to the Execute tab. 2. Click the *Start* button of the DT to execute. 3. Verify that a message appears indicating that execution has started: "Execution successfully for (DTName). Wait until completion for the logs...". 4. Wait for the logs to become available (*Log* button enabled).

Stop execution of a DT	
Requirement ref.	U_08
Purpose	Verify that a user can stop the execution of their DTs.
Precondition	User must be logged in. The DT must be actively executing in the Execute tab.
Test data	A DT that is currently executing.
Process steps	1. Navigate to the Execute tab and ensure that the DT is currently executing. 2. Click the *Stop* button. 3. Verify that a message appears indicating that the execution has been stopped, such as "Execution stopped successfully for (DTName)." 4. Confirm that the DT no longer shows the "Running" status and is now marked as "Stopped."

View execution logs of a DT	
Requirement ref.	U_09
Purpose	Verify that a user can view the logs of their DTs, i.e., the result of the execution.
Precondition	User must be logged in. The DT must have been executed.
Test data	A DT that has been executed and generated logs.
Process steps	1. Navigate to the Execute tab and start the execution of the DT. 2. Wait until the execution is completed. 3. When it is enabled, click on the *Log* button associated with the DT. 4. Verify that the logs of the executed pipeline job are displayed.

Appendix D

SUS questionnaire

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well-integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Bibliography

[1] Qinglin Qi, Fei Tao, Tianliang Hu, Nabil Anwer, Ang Liu, Yongli Wei, Lihui Wang, and Andrew YC Nee. «Enabling technologies and tools for digital twin». In: *Journal of Manufacturing Systems* 58 (2021), pp. 3–21 (cit. on p. 1).

[2] Stefan Mihai et al. «Digital twins: A survey on enabling technologies, challenges, trends and future prospects». In: *IEEE Communications Surveys & Tutorials* 24.4 (2022), pp. 2255–2291 (cit. on p. 1).

[3] *The Power of Digital Twins in the Age of Industry 4.0 – Applications & Benefits*. `https://www.xavor.com/blog/the-power-of-digital-twins-in-the-age-of-industry/` (cit. on p. 2).

[4] *DevOps*. `https://about.gitlab.com/topics/devops/` (cit. on p. 2).

[5] *DTaaS Website*. `https://into-cps-association.github.io/DTaaS/version0.3/index.html` (cit. on pp. 2, 16, 25, 36, 38, 39).

[6] Prasad Talasila, Cláudio Gomes, Peter Høgh Mikkelsen, Santiago Gil Arboleda, Eduard Kamburjan, and Peter Gorm Larsen. «Digital twin as a service (DTaaS): a platform for digital twin developers and users». In: *2023 IEEE Smart World Congress (SWC)*. IEEE. 2023, pp. 1–8 (cit. on pp. 2, 8, 38).

[7] Francis Bordeleau, Ali Motamedi, and Érik Poirier. «A DevOps Approach for the Systematic Development and Evolution of Built Assets Digital Twins». In: () (cit. on p. 5).

[8] Gong Chao. «Human-computer interaction: process and principles of human-computer interface design». In: *2009 International Conference on Computer and Automation Engineering*. IEEE. 2009, pp. 230–233 (cit. on pp. 6, 7).

[9] Alan Dix. *Human-computer interaction*. Pearson Education, 2003 (cit. on pp. 6, 7).

[10] Brad Myers. «Challenges of HCI design and implementation». In: *interactions* 1.1 (1994), pp. 73–83 (cit. on pp. 6, 7, 21).

[11] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human-computer interaction*. Addison-Wesley Longman Ltd., 1994 (cit. on pp. 6, 7).

[12] Detlef Zuehlke and Nancy Thiels. «Useware engineering: a methodology for the development of user-friendly interfaces». In: *Library Hi Tech* 26.1 (2008), pp. 126–140 (cit. on p. 6).

[13] Mariana Segovia and Joaquin Garcia-Alfaro. «Design, modeling and implementation of digital twins». In: *Sensors* 22.14 (2022), p. 5396 (cit. on pp. 7, 8).

[14] Rafael Duque, Crescencio Bravo, Santos Bringas, and Daniel Postigo. «Leveraging a visual language for the awareness-based design of interaction requirements in digital twins». In: *Future Generation Computer Systems* 153 (2024), pp. 41–51 (cit. on pp. 7, 8).

[15] Barbara Rita Barricelli and Daniela Fogli. «Digital twins in human-computer interaction: A systematic review». In: *International Journal of Human–Computer Interaction* 40.2 (2024), pp. 79–97 (cit. on p. 8).

[16] Xin Liu, Du Jiang, Bo Tao, Feng Xiang, Guozhang Jiang, Ying Sun, Jianyi Kong, and Gongfa Li. «A systematic review of digital twin about physical entities, virtual models, twin data, and applications». In: *Advanced Engineering Informatics* 55 (2023), p. 101876 (cit. on p. 8).

[17] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. «A survey on digital twin: Definitions, characteristics, applications, and design implications». In: *IEEE access* 7 (2019), pp. 167653–167671 (cit. on p. 8).

[18] Santiago Gil, Peter H Mikkelsen, Cláudio Gomes, and Peter G Larsen. «Survey on open-source digital twin frameworks–A case study approach». In: *Software: Practice and Experience* 54.6 (2024), pp. 929–960 (cit. on pp. 8, 11).

[19] *AWS IoT TwinMaker*. `https://aws.amazon.com/iot-twinmaker/` (cit. on p. 8).

[20] *AWS IoT TwinMaker Is Now Generally Available*. `https://aws.amazon.com/blogs/aws/aws-iot-twinmaker-is-now-generally-available/` (cit. on p. 9).

[21] *Azure Digital Twins*. `https://azure.microsoft.com/en-us/products/digital-twins` (cit. on p. 9).

[22] *What is Azure Digital Twins?* `https://learn.microsoft.com/en-us/azure/digital-twins/overview` (cit. on p. 10).

[23] *Eclipse Ditto*. `https://eclipse.dev/ditto/` (cit. on p. 10).

[24] *Ditto Explorer User Interface*. https://eclipse.dev/ditto/user-interface.html (cit. on p. 11).

[25] *Eclipse AASX Package Explorer*. https://github.com/eclipse-aaspe/package-explorer (cit. on pp. 11, 12).

[26] Daniel A Magües, John W Castro, and Silvia T Acuna. «HCI usability techniques in agile development». In: *2016 IEEE International Conference on Automatica (ICA-ACCA)*. IEEE. 2016, pp. 1–7 (cit. on p. 14).

[27] Tiago Silva da Silva, Milene Selbach Silveira, and Frank Maurer. «Usability evaluation practices within agile development». In: *2015 48th Hawaii International Conference on System Sciences*. IEEE. 2015, pp. 5133–5142 (cit. on p. 14).

[28] *Agile Manifesto*. https://agilemanifesto.org/ (cit. on p. 14).

[29] Muhammad Usman Tariq. «User Centered Human-Computer Interaction and Agile Development: A Systematic Model for Useable Product Case Study». In: *International Business Information Management Association Conference*. 2020 (cit. on pp. 14, 15).

[30] *What is prototyping and why is mid fidelity its unsung hero in rapid testing?* https://thegood.com/insights/what-is-prototyping/ (cit. on p. 15).

[31] *What is CI/CD?* https://about.gitlab.com/topics/ci-cd/ (cit. on p. 17).

[32] *CI/CD pipelines*. https://docs.gitlab.com/ee/ci/pipelines/ (cit. on p. 17).

[33] *Gitbeaker*. https://github.com/jdalrymple/gitbeaker (cit. on p. 17).

[34] *What is a REST API?* https://www.ibm.com/topics/rest-apis (cit. on p. 18).

[35] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003 (cit. on p. 18).

[36] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (cit. on p. 18).

[37] *The testing pyramid: Strategic software testing for Agile teams*. https://circleci.com/blog/testing-pyramid/ (cit. on pp. 18, 19).

[38] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020 (cit. on pp. 19, 23).

[39] *What is Integration Testing?* https://katalon.com/resources-center/blog/integration-testing (cit. on pp. 20, 21).

[40] *Acceptance Testing : What, Why, Types & How to Do?* https://testsigma.com/guides/acceptance-testing/ (cit. on p. 22).

[41] *Jest*. https://jestjs.io/docs/getting-started (cit. on p. 23).

[42] *Downstream pipelines*. https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html/ (cit. on p. 27).

[43] *CI/CD YAML syntax reference*. https://docs.gitlab.com/ee/ci/yaml/ (cit. on pp. 28, 29).

[44] *Trigger pipelines by using the API*. https://docs.gitlab.com/ee/ci/triggers/ (cit. on p. 29).

[45] *Client design - DTaaS GitHub repo*. https://github.com/INTO-CPS-Association/DTaaS/wiki/Client-design/ (cit. on pp. 40–42).

[46] *User Acceptance Testing: Complete Guide with Examples*. https://www.functionize.com/automated-testing/acceptance-testing-a-step-by-step-guide (cit. on pp. 49, 50).

[47] Brian Hambling and Pauline Van Goethem. *User acceptance testing: a step-by-step guide*. BCS Publishing, 2013 (cit. on pp. 49–53).

[48] *System Usability Scale (SUS) Practical Guide for 2024*. https://blog.uxtweak.com/system-usability-scale/ (cit. on p. 54).