Bachelor Thesis

Petar Krastev

Design and implementation of a microservice for deletion of resources in the Multi-Agent Research and Simulation distributed system

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Petar Krastev

Design and implementation of a microservice for deletion of resources in the Multi-Agent Research and Simulation distributed system

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Information Engineering

at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. rer. nat. Henning Dierks
Second examiner : Prof. Dr. rer. nat. Thomas Clemen

Day of delivery July 23rd 2018

Petar Krastev

Title of the Bachelor Thesis

Design and implementation of a microservice for deletion of resources in the Multi-Agent Research and Simulation distributed system

Keywords

Distributed deletion; Distributed systems; Microservices; Multi-Agent Research and Simulation (MARS); ASP.NET Core; C#

Abstract

The work presented in this document provides a distributed deletion solution for the MARS simulation framework in the form of a microservice. MARS is a distributed system that has adopted a microservice-based architecture. The deletion service deals with issues related to concurrent access of resources by users and by other services within the system, while the deletion process takes place.

Petar Krastev

Thema der Bachelorarbeit

Design und Implementierung eines Microservices zum löschen von Ressourcen im verteilten Cloud System des MARS Frameworks

Stichworte

Verteiltes Löschen; Verteilte Systeme; Microservices; Multi-Agent Research and Simulation (MARS); ASP.NET Core; C#

Kurzzusammenfassung

Die vorliegende Arbeit demonstriert eine Lösung zum verteilten Löschen im Cloud System der MARS Gruppe auf Basis eines Microservies. MARS ist ein verteiltes System auf Basis der Microservice-Architektur. Der Lösch-Service (deletion-service) ist so gebaut dass er alle potentiellen Probleme des parallelen Zugriffs durch Nutzer oder anderer Services beachtet und sicherstellt dass der Lösch-Vorgang vollständig erfolgt.

Table of contents

List of tables

List of figures

Acronyms and abbreviations

API	Application Programming Interface
DTO	Data Transfer Object
GIS	Geographic Information System
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IoC	Inversion of Control
MSaaS	Modelling and Simulation as a Service
MARS	Multi-Agent Research and Simulation
NoSQL	Non Relational, Non SQL, Not only SQL
RPC	Remote Procedure Calls
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SQL	Standart Query Language
UI	User Interface

1 Introduction

Distributed systems play a major role in our lives. They are widely used in areas such as the Internet, healthcare, education, science, eCommerce, financial trading and others. The prime motivation for constructing and using distributed systems is the desire to share resources. The term ‘resource‘ is characterized by the range of things that can be usefully shared in a networked computer system. The definition spans from hardware components such as powerful processors and storage devices to software-defined entities such as files, databases and data objects of all kinds. It includes the stream of video frames, produced by a digital video camera, and the audio connection that a mobile phone call represents. However, there are challanges when designing and building distributed systems. A major concern is concurrency. The presence of multiple processes and users in a distributed system is a source of concurrent requests to its resources. Each resource must be designed to be safe in a concurrent environment [1].

The Multi-Agent Research and Simulation (MARS) framework, part of the Computer Science Department, Hamburg University of Applied Sciences, is designed as a tool for work groups that are considering multi-agent modelling and simulation. The framework provides a complete tool chain from data import to result visualization and analysis, which allows for large-scale model development in a web environment and execution in a high-performance cloud [2].

Since the storage capacity of any system is limited and complex simulations produce data in large volumes, the need for a solution, which removes data from the MARS system, is highly desired. This work presented in this document aims to present a distributed deletion solution, in the form of a microservice, for the MARS system.

1.1 Problem statement

The main objective of this thesis is to provide a deletion solution for the MARS system. MARS is a distributed system that has adopted the microservice architecture pattern. All of the application’s logic is separated into multiple independent services that communicate with each other via network calls. A major challenge is the concurrent usage of resources by microservices during the deletion process. Furthermore, multiple users can have access to the same resources. All factors specified impose risks of race conditions that could occur while the deletion process takes place. The current infrastructure and the workflow of the MARS system are examined in-depth for the purpose of providing a tailored solution that is most beneficial. The deletion service gives users the ability to delete any resource they have uploaded or created, which in return allows them to manage their cloud space in a more efficient manner.

2 Theoretical background

This chapter introduces the concept of distributed systems in **Section 2.1** and microservices in **Section 2.2**. The MARS system is analyzed in-depth in **Section 2.3**.

2.1 Distributed systems

A distributed system is a collection of autonomous computing elements that appear to its users as a single coherent system. This definition refers to two characteristic features of distributed systems [3].

The first characteristic is that a distributed system is a collection of computing elements, each being able to behave independently of the others. However, it must be noted that if they ignore each other, there is no use in putting them together. Modern distributed systems can consist of all kinds of computing elements, ranging from high-performance computers to small plug computers or event smaller devices. The computing elements are programmed to achieve common goals, which are realized only by exchanging messages with each other. A consequence of dealing with independent computing elements is that each one will have its own notation of time. In other words, there is no single global notation of time within a distributed system. This leads to challenges in synchronization and coordination within the system [3].

The second characteristic is that a distributed system should appear as a single coherent system. End users must believe that they are dealing with a single system and they should not notice that processes and data are dispersed across a computer network. A single coherent system, which is made of multiple computing elements, has to operate in the same way, no matter how the interaction between the user and the system takes place. However, in reality this is an extremely complex task to achieve. Since distributed systems consist of multiple autonomous computing elements, at any point in time, any of them can fail. This would leave the application running with only partial functionality, which is common to complex systems [3].

Some of the main goals and challenges worth considering when designing distributed systems is to support resource sharing, to make the distribution transparent, to achieve high openness and to achieve scalability [3].

2.2 Microservices

The microservice architecture pattern is a paradigm for programming applications by the composition of small independent services, called microservices. Each microservice runs its own process and communicates with other services via network calls. To establish the exchange of information between the microservices, each one must expose an Application Programming Interface (API). The microservice pattern is built on the concepts of Service-Oriented Architecture (SOA), which puts an emphasis on the design and development of highly maintainable and scalable software. Microservices manage growing complexity by decomposing large systems into a set of services. This approach focuses on loose coupling, high cohesion and it is beneficial in terms of modularity, maintainability and scalability [4]. Company names, such as Netlix, Amazon and others, have joined the trend of decoupling large monolithic systems into a set of independent services [5] [6] [7]. **Figure 1** shows a taxi-hailing company's infrastructure represented as a monolith application and a refactored version, which uses microservices.

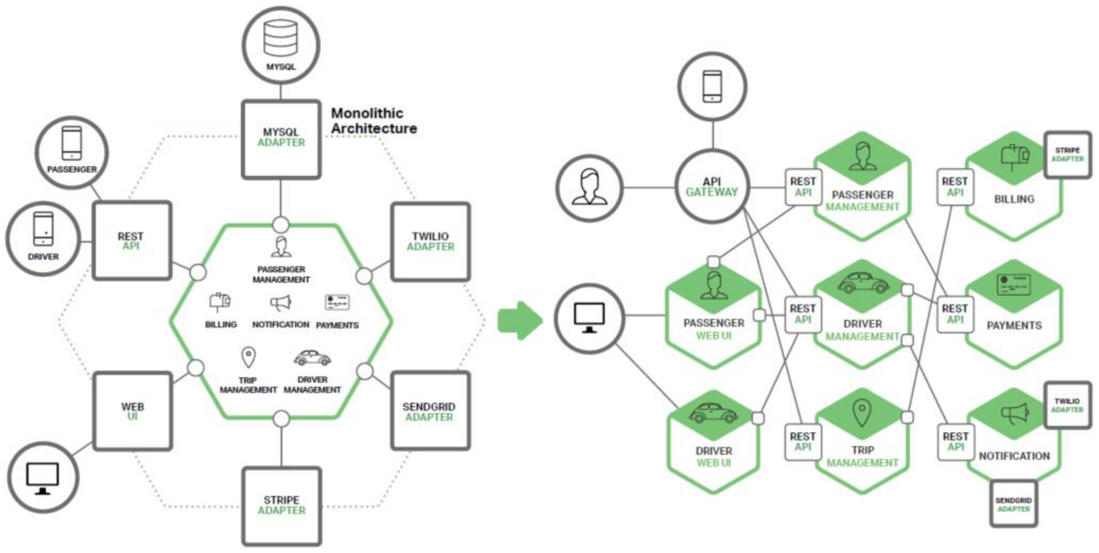


Figure 1: A taxi-hailing company's infrastructure represented as a monolith application (to the left) and as a microservice-based application (to the right) [8]

By enforcing the microservice architecture pattern, the application can leverage the following advantages.

First, the application benefits from faster deployment cycles. Individual services can be changed and deployed independently of the rest of the system, while monolith applications require a deployment of the whole system for a single change [9].

Second, the cost to replace or completely rework any of the services is lower, because of the smaller unit of work that each service has. In contrast, monolith applications tend to grow in complexity and maintenance because of large codebases and legacy code [9].

Third, the application is more resilient. Compared to monolith applications, where if one component fails, the whole system fails, microservices carry on working if a single service fails, since they are independent of each other [9].

Finally, a system composed of multiple, collaborating services, gives the developers the chance to choose the correct tools and development stack for each one. In a monolith application, the development stack is usually the same throughout the project. The technology heterogeneity advantage allows microservices to utilize different database for a specific purpose. In **Figure 1**, the microservice-based application does not show the database schema for the microservices, because each service has its own [9].

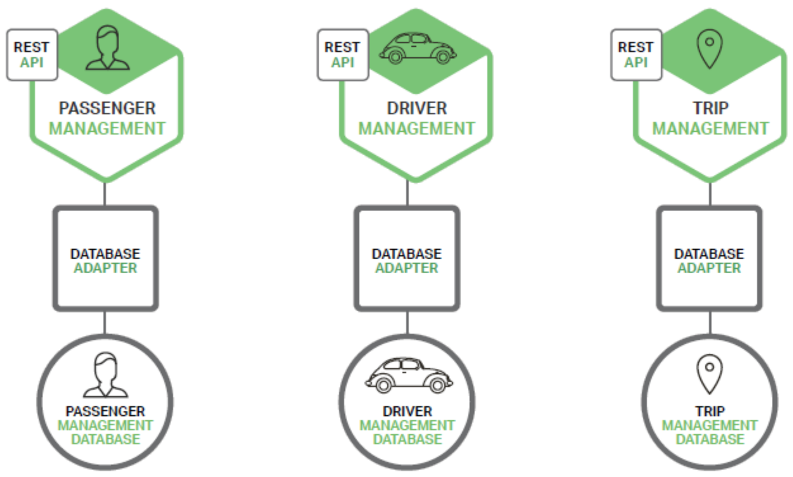


Figure 2: A taxi-hailing company's infrastructure represented by micoservices together with their database schema [8]

Figure 2 represents the microservices together with their database schema. By enforcing each service to use its own database schema, loose coupling can be achieved, which in return increases the scalability of the overall system. Scaling individual services according to the system's needs is far easier to accomplish, rather than scaling everything together at once, which is usually the case in a monolith application [9].

Even though there are many advantages gained by enforcing the microservice architecture pattern, there are also drawbacks. First, maintaining multiple services, which utilize different development stacks, can become a very complex task. Second, since each service has its own database schema, updating multiple database entries at once cannot be easily resolved. Additional functionality is needed to develop transactions that span multiple services. In a monolith application, this is not a big concern, since, usually, there is only one database for the whole application. Finally, testing a microservice-based application is quite a challenge. Automated end-to-end tests are hard to develop, since the application is composed of many parts that must work together in the testing environment [8] [9].

2.3 MARS

MARS is conceptualized as a Modelling and Simulation as a Service (MSaaS) system. It follows the trend of moving massive-scale simulations into the cloud, which enables users to gain access to all functionality of the MARS system through a user-friendly web interface [2]. MARS is a massive multi-agent platform that can support up to millions of agents per simulation [10]. The simulation models must be created in the C# programming language and they must obey the rules and constraints of the simulation engine [11].

MARS is a distributed system, which has adopted the microservice architecture pattern. All services are packaged into Docker containers, deployed and orchestrated by Kubernetes.

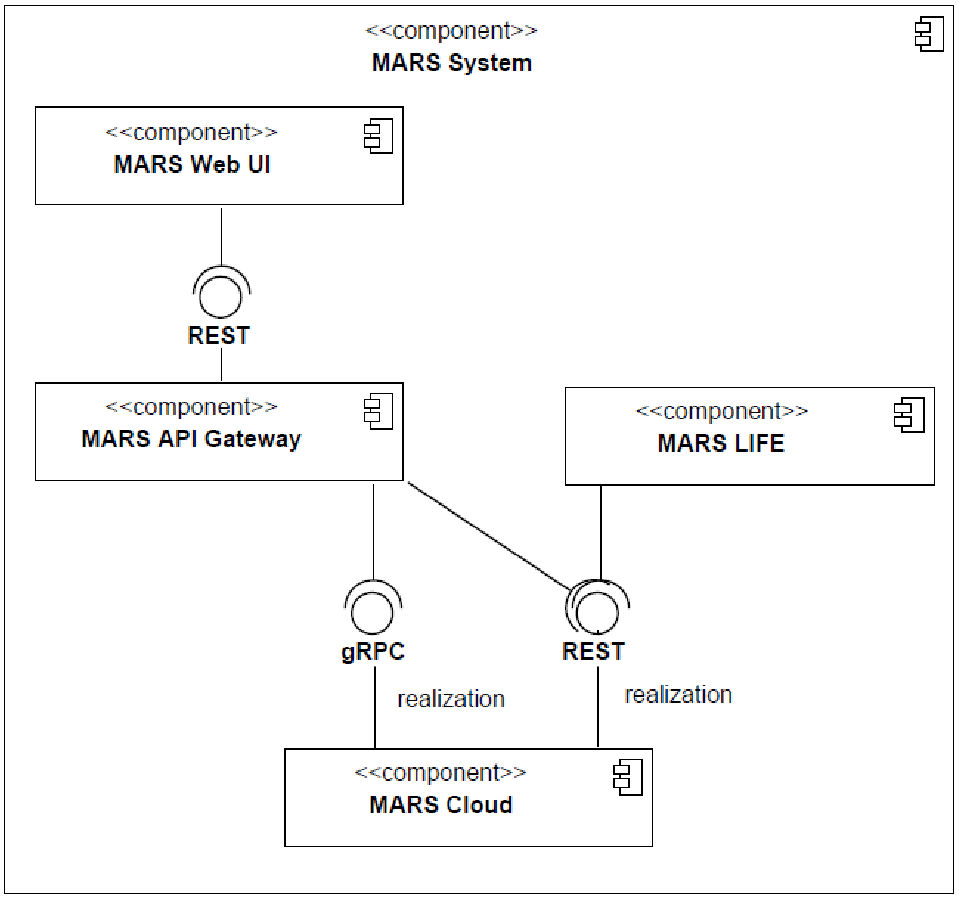


Figure 3: Component diagram representing an overview of the MARS system

Figure 3 presents an overview of the key components of the MARS system. The definition of each one is as follows:

MARS Web UI: Represents the User Interface (UI) for the whole system. It is accessible through the web browser and it acts as a client for all other components.

MARS API Gateway: The API Gateway encapsulates the internal system architecture and provides an API that is tailored for the client. In addition, it contains extra functionality such as authentication [8].

MARS Cloud: Contains all application logic, which is split into microservies.

MARS LIFE: The simulation engine, responsible for running simulations.

2.3.1 MARS Cloud

The MARS Cloud is built from multiple microservices, which encapsulate the application's logic. Each service exposes an API, either REST (Representational State Transfer) or gRPC (Remote Procedure Calls), which all other services use to establish communication. At the time of writing, there are no restrictions on the inner-service communication. Any service can call any service. **Figure 4** illustrates some of the services, together with their database connection and the type of API that they expose. For reasons of simplicity, only the services relevant to the deletion process are included.

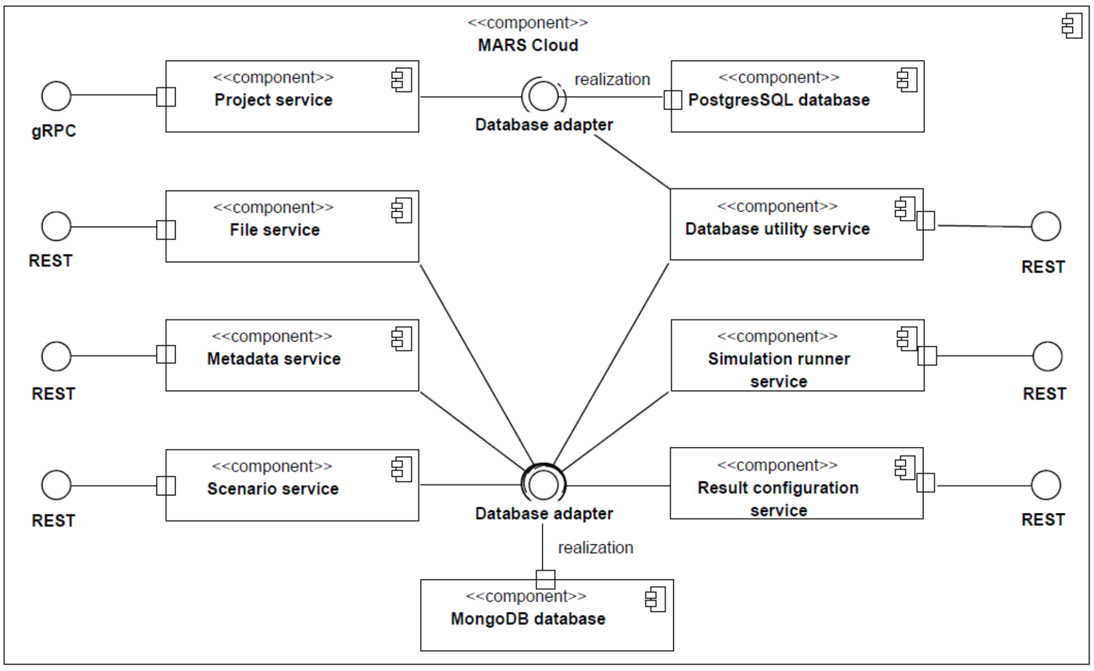


Figure 4: Component diagram representing a simplified version of the MARS Cloud

An overview of each service is as follows:

Project service: Contains all project related logic.

File service: Responsible for manipulating uploads of type Model or Data Layer (TimeSeries, Geographic Information System (GIS), etc.). Depending on the type, the upload is stored into a specific database.

Metadata service: Responsible for manipulating meta-information belonging to an upload. Metadata is generated as soon as an upload functionality is started.

Scenario service: Contains logic for manipulating scenarios.

Result configuration service: Contains all result configuration related logic.

Simulation runner service: Contains logic for manipulating simulation plans and simulation runs. In addition, the service is responsible for managing the simulation processes.

Database utility service: Provides an automation tool, which enables tasks such as backups, for various databases used in the system, such as MongoDB and PostgreSQL. In addition, it contains logic for manipulating result data collections.

MongoDB database: MongoDB is an open-source, non-relational (NoSQL) database that is powerful, flexible and scalable. The database offers users ease of use and ease of scaling without sacrificing performance [12]. These are among some of the factors why MongoDB is the most widely used database in the MARS system.

PostgresSQL database: PostgresSQL is an open-source, relational database that supports the Standard Query Language (SQL). The database is reliable and performant with support for transactions [13].

2.3.2 Related technologies

This section presents a brief introduction, together with the usage of some of the technologies used within the MARS system.

REST

Representational State Transfer (REST) is an architectural style defined to help create and organize distributed systems. It is a set of constraints, most commonly associated with the Hypertext Transfer Protocol (HTTP). REST is a resource-based architecture, where a resource is accessed via a common interface based on the HTTP standard methods [14]. Most services in the MARS Cloud expose a REST interface, which all collaborating services use to exchange information.

gRPC

gRPC is a modern, open-source, high performance Remote Procedure Calls (RPC) framework, which can run in any environment. The framework is used to connect services within data centers, distributed computing environments and many more [15]. gRPC uses protocol buffers as the Interface Definition Language (IDL) for describing the service interface and the structure of the payload messages [16]. Only a few services, namely project and user, expose a gRPC interface.

Docker

Docker is an open-source engine that automates the deployment of applications into containers. It is designed to provide a lightweight and fast environment that enables developers to build, run, collaborate and deploy different programs seamlessly into different environments – local, test, production, etc. [17] All services inside the MARS system are packaged into Docker containers.

Kubernetes

Kubernetes is an open-source project and an orchestration tool for containerized applications. It helps organizations deal with some of the major operations and management concerns such as resource utilization, high availability, updates, patching, networking, service discovery, monitoring and logging [18]. Kubernetes is used to orchestrate all microservices within the MARS system.

3 Requirement analysis

3.1 Functional requirements

The deletion service would give users the possibility to delete any resource they have uploaded or created. **Figure 5** illustrates the main functional requirements of the deletion service.

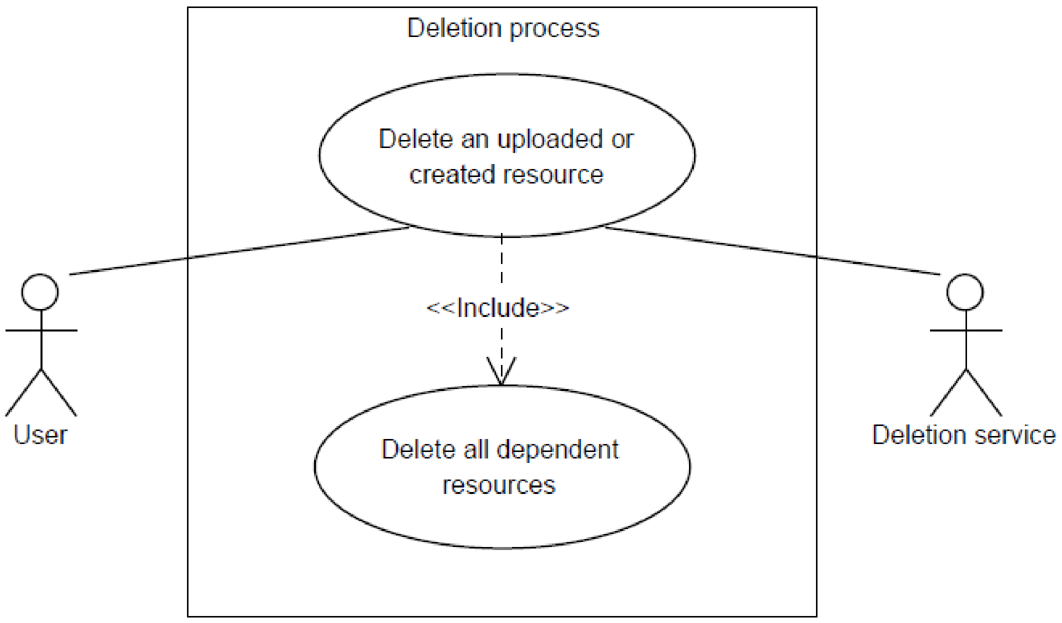

Deletion process
Delete an uploaded or created resource
<<Include>>
Delete all dependent resources
User
Deletion service

Figure 5: Use case diagram for the main functional requirements of the deletion process

Proper planning of deletion in any system requires specifying what kind of resources does the system produce and use. For simplicity, only the resources relevant for the deletion process are taken into consideration.

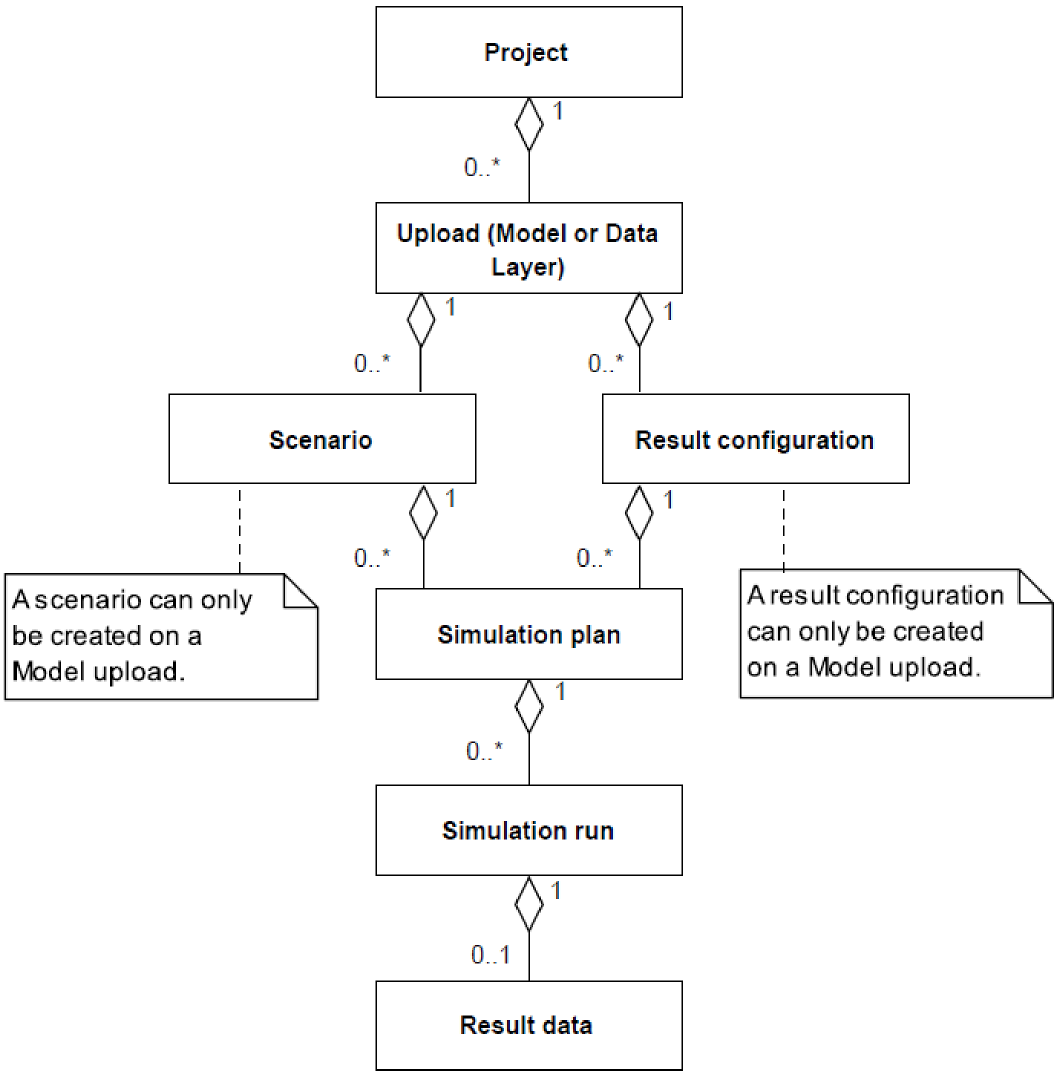

Project
1
0..*
Upload (Model or Data Layer)
1
0..*
1
0..*
Scenario
Result configuration
1
0..*
1
0..*
A scenario can only be created on a Model upload.
Simulation plan
A result configuration can only be created on a Model upload.
1
0..*
Simulation run
1
0..1
Result data

Figure 6: MARS resource dependency and relation diagram

The resources, together with their hierarchy and relations, are shown in **Figure 6**. The definition for each is presented in **Table 1**.

Resource	**Definition**
Project	Provides access to all project contents to members.
Upload	**Model upload**: Provides the definition of the simulation. **Data Layer upload**: Provides data for the creation of the model, it can be of different types (GIS, TimeSeries, etc.).
Scenario	Gives the opportunity to set and alter the simulation parameters. It can be created based on a Model upload only.
Result configuration	Acts as a filter for the simulation result data. It can be created based on a Model upload only.
Simulation plan	Combines an upload, scenario and result configuration into one entity, which can be run as a simulation.
Simulation run	Produces result data based on a simulation plan.
Result data	Represents the result of the simulation run.

Table 1: Resources produced by the MARS system together with definitions for each one

All specified resources have a dependency on each other. A scenario cannot be created without an upload; a simulation plan cannot be created without a scenario and so on. When deleting any of the resources, the deletion process must ensure that the desired resource is deleted together with all dependent resources. Otherwise, the system will be cluttered with data that cannot be used.

3.1.1 Atomic deletion

A crucial aspect that must be considered when designing the deletion service is the distributed nature of the MARS system. Since all MARS services run on multiple machines, there is a chance that any service could be killed and rescheduled on a different machine at any time. Reasons for this could be workload, optimization of resources, machine failure, etc. This imposes risks of partial completion for a deletion process. Partial deletion would produce inconsistencies in the MARS system, because of the resource dependencies.

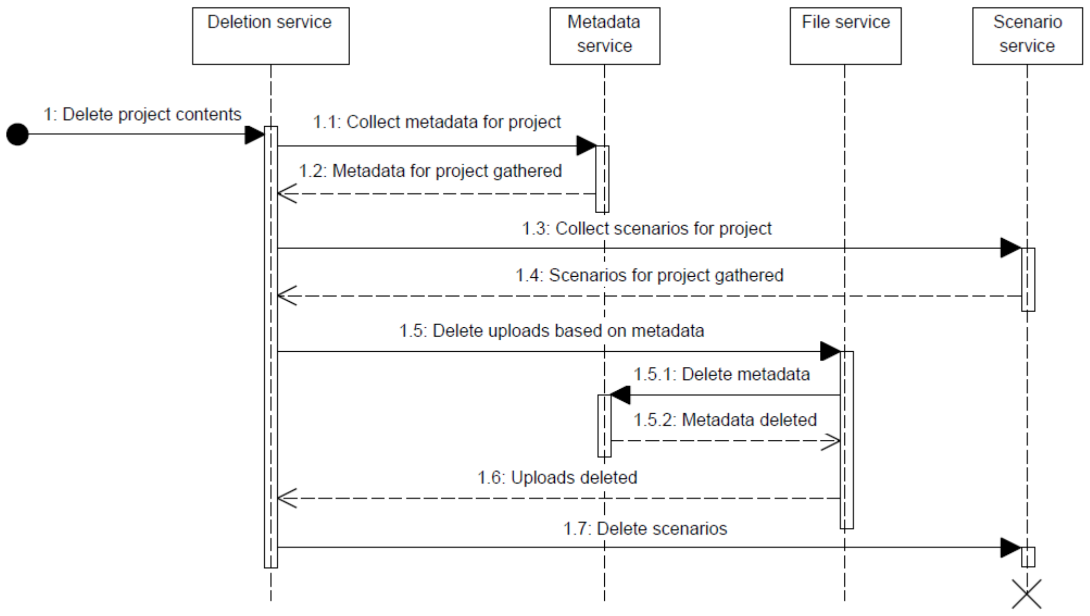

Deletion service
Metadata service
File service
Scenario service
1: Delete project contents
1.1: Collect metadata for project
1.2: Metadata for project gathered
1.3: Collect scenarios for project
1.4: Scenarios for project gathered
1.5: Delete uploads based on metadata
1.5.1: Delete metadata
1.5.2: Metadata deleted
1.6: Uploads deleted
1.7: Delete scenarios

Figure 7: Sequence diagram representing a partial deletion process

Figure 7 presents a scenario where the deletion process is partially successful in deleting resources belonging to a project. The scenario service is killed during the deletion process, which does not give the deletion service the chance to delete all resources. Since scenarios have a dependency on uploads, they cannot be used (refer to **Figure 6**). At some point in time, Kubernetes will reschedule the scenario service and the deletion of the pending resources would be possible.

The deletion process must be executed as an atomic one. The process must succeed or fail as a complete unit; it must never be partially complete.

3.1.2 Avoiding race conditions

Multiple users can have access to the same project and its resources. Furthermore, there are no restrictions on the usage of resources within the system, multiple services can use the same resource at the same time. All listed factors lead to the possibility of race conditions in certain cases during the deletion process.

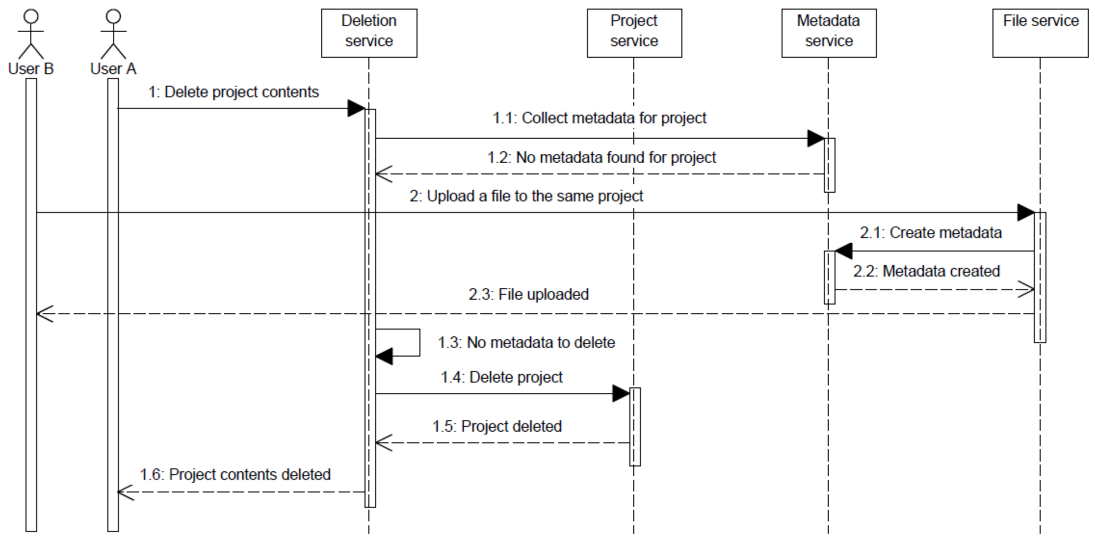

User B
User A
Deletion service
Project service
Metadata service
File service
1: Delete project contents
1.1: Collect metadata for project
1.2: No metadata found for project
2: Upload a file to the same project
2.1: Create metadata
2.2: Metadata created
2.3: File uploaded
1.3: No metadata to delete
1.4: Delete project
1.5: Project deleted
1.6: Project contents deleted

Figure 8: Sequence diagram representing a race condition during the deletion process

Figure 8 illustrates a race condition in which a new resource is created based on a resource that will be deleted. The assumption is made that the deletion service gathers all dependent resources first and starts to delete them afterwards. This leads to the result that the newly created resource is not deleted. If this occurs, the newly created resource cannot be used, because its dependent resources are deleted. Such race conditions must be avoided and a solution must be found that restricts the usage of resources by multiple services at the same time during the deletion process.

3.1.3 Integration into the system

For the purpose of usability, the deletion service must be integrated into the MARS system. Appropriate UI elements, in the form of buttons, must be added to the MARS Web UI. Furthermore, to make the deletion service accessible from the MARS Web UI, the service must also be integrated into the MARS API Gateway.

3.2 Non-functional requirements

3.2.1 Robustness against sudden changes in the service life cycle

Since the deletion service runs in a distributed environment, where at any point in time, any service can be killed and rescheduled, the processes of the service must be designed to recover in such events. **Figure 7** illustrates a case where an external service, namely scenario, is killed during the deletion process. However, it is possible that the deletion service is killed and rescheduled instead of the scenario service. The deletion service must be robust against sudden changes in the life cycle of the service.

3.2.2 High maintainability

At the time of writing, MARS has more than 20 services. Each one utilizes a different programming language, framework and external libraries. Some even have a custom database schema. Maintenance of the whole system grows in complexity with each new feature and service added. This is the prime reason why the deletion service must be highly maintainable. The development stack and tools must be familiar to the MARS developers, because they will be the ones maintaining the service in the future.

3.2.3 High performance output

The deletion service must be highly performant. Appropriate software techniques must be considered when designing and building the service. A higher performance for the service, would yield a better user experience.

4 Planning & software design

This section presents the different approaches considered for assuring safe deletion in **Section 4.1** and **Section 4.2**. The software design phase is discussed in-depth from **Section 4.3** to **Section 4.6**, together with the utilized design patterns in **Section 4.7**.

4.1 Event-driven approach

The event-driven architecture enforces microservices to publish events when something notable happens such as a change in a resource. The term ‘resource’ is defined as any data entity stored into a database. Other microservices subscribe to these events. When a microservice receives an event, it can update its own resources, which might lead to publishing more events. All events are transferred through a message broker. The events can be used to implement transactions that span multiple services if two conditions are met. First, each service must atomically update its database and publish an event. Second, the message broker must guarantee that the events are delivered at least once to each subscribed service [8].

Atomicity could be achieved in a number of ways, but focus is given to event sourcing. Instead of storing the current state of a resource, the application will store a sequence of state-changing events. Whenever a resource is changed, a new event will be stored to the collection of events. The current state of a resource entity can be reconstructed by replaying the events. Since saving an event is a single operation, it is also an atomic one [8].

All events are persisted to an event store, which exposes an API for adding and retrieving events for any resource. The event store behaves similarly to the message broker; it provides logic for subscribing and delivering events to all subscribers [8].

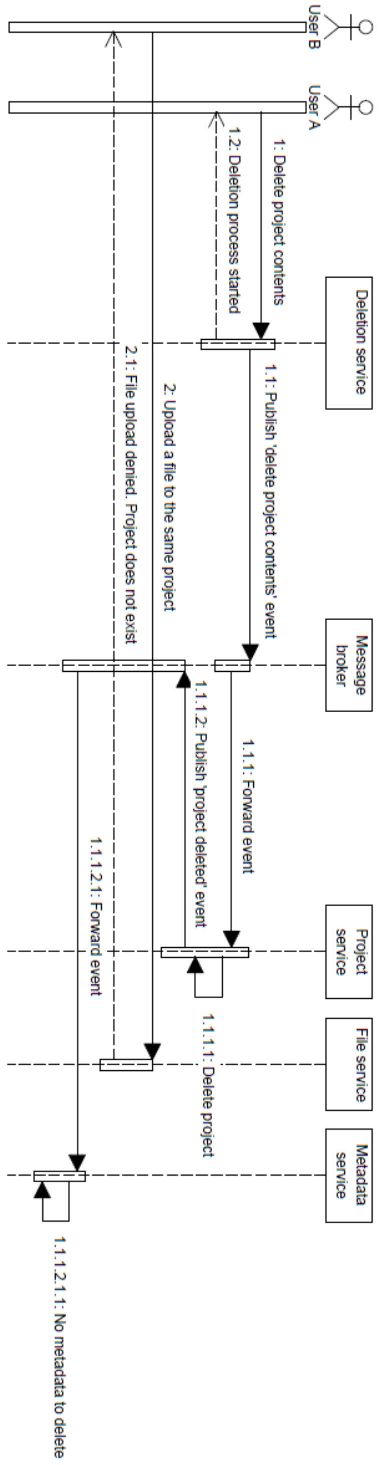

User A
User B
Deletion service
Message broker
Project service
File service
Metadata service
1: Delete project contents
1.2: Deletion process started
1.1: Publish 'delete project contents' event
1.1.1: Forward event
1.1.1.1: Delete project
1.1.1.2: Publish 'project deleted' event
1.1.1.2.1: Forward event
1.1.1.2.1.1: No metadata to delete
2: Upload a file to the same project
2.1: File upload denied. Project does not exist

Figure 9: Sequence diagram representing a deletion process using the event-driven approach

Figure 9 presents the workflow of the deletion process, specified in **Figure 8**, using the event-driven approach. The deletion process deletes resources starting from the project and ending with the result data, according to the dependency tree in **Figure 6**. The order of deleting resources is crucial for the success of the deletion process, since new resources cannot be created on already deleted ones. This strategy is used to avoid the race conditions indicated in **Figure 8**. The message broker is responsible for passing events among the different services. Once a service receives an event, it can execute some logic and publish a new event.

4.1.1 Infrastructure changes

A tool, which can be used to realize the message broker, already exists in the MARS system. However, most of the services must be changed completely for this approach to work. Instead of storing the current state of each resource, each service must store the state-changing events for the resource entity. In addition, each service must include logic for publishing events to the message broker and reacting to them accordingly when subscribing. An event model must be introduced and integrated in the whole application.

4.1.2 Summary

The event-driven approach enforces services to publish events when a resource entity has changed and to react accordingly when an external event is received. This approach would allow transactions that can span multiple services. The event-driven approach would be a perfect solution for the deletion process. It would even solve issues related to data consistency in the MARS system. However, it requires massive changes to the infrastructure of the MARS Cloud. The time limitation and complexity of this approach led to the search of another, which would take bigger advantage of the current architecture of the MARS Cloud.

4.2 Marking approach

The marking approach requires the introduction of a ‘to be deleted’ flag to each resource definition, specified in **Figure 6**. The flag would be an indication for all services whether the resource could be used. The resources must be marked first before they are deleted. The deletion process would mark resource according to the dependency tree (refer to **Figure 6**), starting with projects and ending with result data. This method provides assurance that if the services obey the flags, the race conditions defined in **Figure 8** can be avoided.

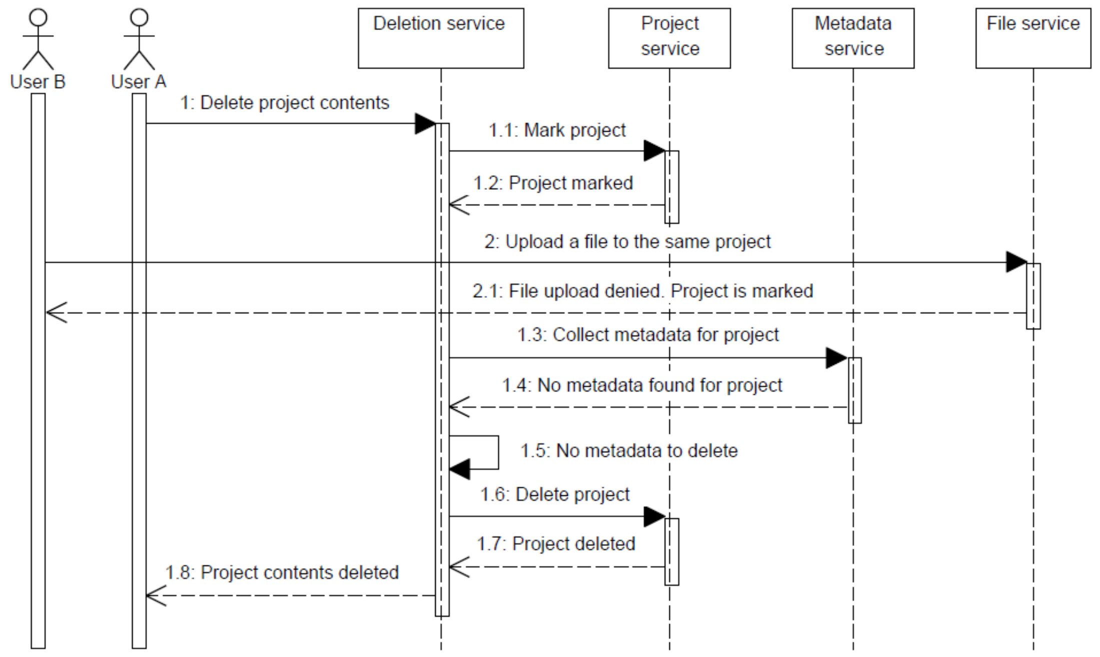

User B
User A
Deletion service
Project service
Metadata service
File service
1: Delete project contents
1.1: Mark project
1.2: Project marked
2: Upload a file to the same project
2.1: File upload denied. Project is marked
1.3: Collect metadata for project
1.4: No metadata found for project
1.5: No metadata to delete
1.6: Delete project
1.7: Project deleted
1.8: Project contents deleted

Figure 10: Sequence diagram representing the deletion process using the marking approach

Figure 10 presents the workflow of the deletion process, specified in **Figure 8**, using the marking approach. Since a mark is placed on the project resource and all other services obey the mark, the race conditions are avoided and the deletion process is successful in execution.

4.2.1 Infrastructure changes

Table 2 summarizes all infrastructure changes required by the marking approach.

Service name	Required changes
Project	A 'to be deleted' flag must be added to the project resource definition.
Metadata	The metadata resource definition already contains a 'to be deleted' state, no changes are required here.
File	Must deny the upload of any files to a marked project
Scenario	A 'to be deleted' flag must be added to the scenario resource definition. Must deny the creation of a scenario based on a marked metadata.
Result configuration	Must deny the creation of a result configuration based on a marked metadata. The result configurations will reuse the 'to be deleted' state of the metadata resource definition, no additional flag is needed here.
Simulation runner	A 'to be deleted' flag must be added to the simulation plan and simulation run resource definitions. Must deny the creation of a simulation plan based on a marked metadata, scenario or result configuration. Must deny the start of a simulation run based on a marked simulation plan.
Database utility	A 'to be deleted' flag must be added to the result data resource definition.

Table 2: Infrascture changes for the MARS Cloud required by the marking approach

In addition, appropriate API endpoints must be introduced to each service for retrieving and changing the 'to be deleted' flag, wherever needed.

4.2.2 Summary

The marking approach introduces the concept of marking resources. Prior to deleting a resource, it must be marked as 'to be deleted'. The marks can be used as an indication for all services whether the resources can be used or not. Due to the lower complexity and fewer infrastructure changes, the marking approach has been chosen to limit the usage of resources by multiple services and to avoid the race conditions that could occur during the deletion process.

4.3 Separation of concerns

Following the marking approach in **Section 4.2**, the deletion service would consist of two main processes: marking all resources first and then deleting them. However, marking all resources and guaranteeing that no other service can alter them at the same time could possibly be utilized by other services as well. At the time of writing, the archive service [19] is being developed in parallel and would require similar functionality. This is the main reason why the marking logic has been separated from the deletion service into a separate one called the marking service.

4.4 Development framework

The ASP.NET Core framework is chosen to realize the marking and deletion services, because the MARS LIFE simulation engine is also implemented using the same framework. The conclusion can be drawn that MARS developers are familiar with the C# framework and they will be able to maintain it in the future. Furthermore, the framework is rich in utilities for development and testing purposes.

4.5 Marking service

4.5.1 Additional requirements

The introduction of a marking service leads to uncertainties of the requirements and specifications for the service. This section aims to address these issues.

The main responsibility of the marking service is to find and mark all dependent resources belonging to a root resource. Since the marking service places the marks on all the resources, it must also pay attention to them. This is of high importance, because it removes the possibility that two external services can manipulate the same resources at the same time.

Furthermore, while the marks are in place, any changes to the marked resources should be restricted. This is less relevant for the deletion service, but the archive service [19] demands this restraint. In the current MARS system, users are able to alter the scenarios and result configurations. An additional requirement to the scenario and result configuration services is that while the marks are in place, any changes to the scenarios and result configurations must be denied.

4.5.2 Workflow

As stated above, the main responsibility of the marking service is to find and mark all dependent resources belonging to a root resource. In order to achieve this goal, the marking service must go through the dependency tree. If an already marked resource is encountered, the current marking process must be aborted and the marks reverted. In this case, the implication can be made that some other service is already using the marked resource and only it can alter the resource at any given time. The workflow of the marking process is described in **Figure 11**.

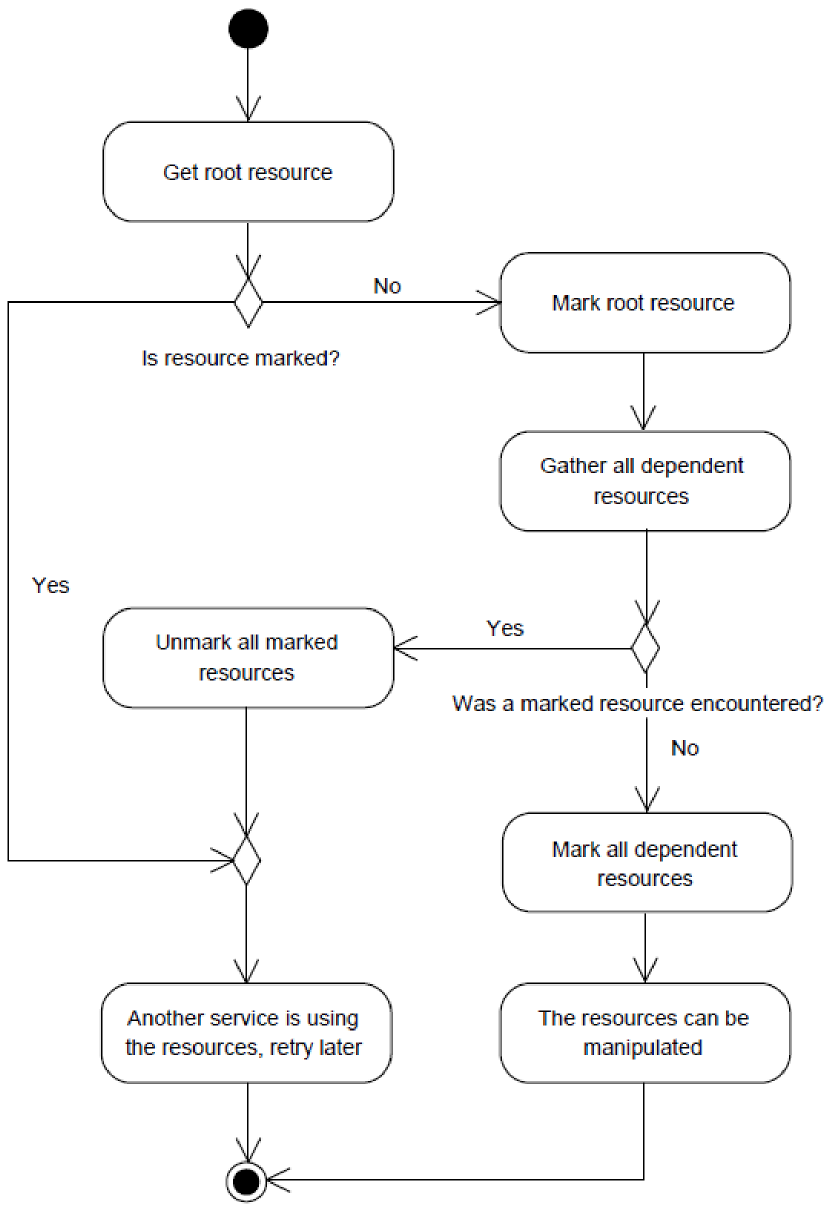

Get root resource
No
Mark root resource
Is resource marked?
Gather all dependent resources
Yes
Unmark all marked resources
Yes
Was a marked resource encountered?
No
Mark all dependent resources
Another service is using the resources, retry later
The resources can be manipulated

Figure 11: Activity diagram representing the workflow of the marking service

Figure 12 presents all services that are involved in the marking process. Each step is represented by a call to an external service, where the response is awaited. Once the service responds, the program execution continues. However, if the response is awaited forever, there could be possible deadlocks. For this reason, the interfaces used for establishing the inner-service communication have a timeout for the requests. If the external service fails to reply within the given time, an exception is thrown.

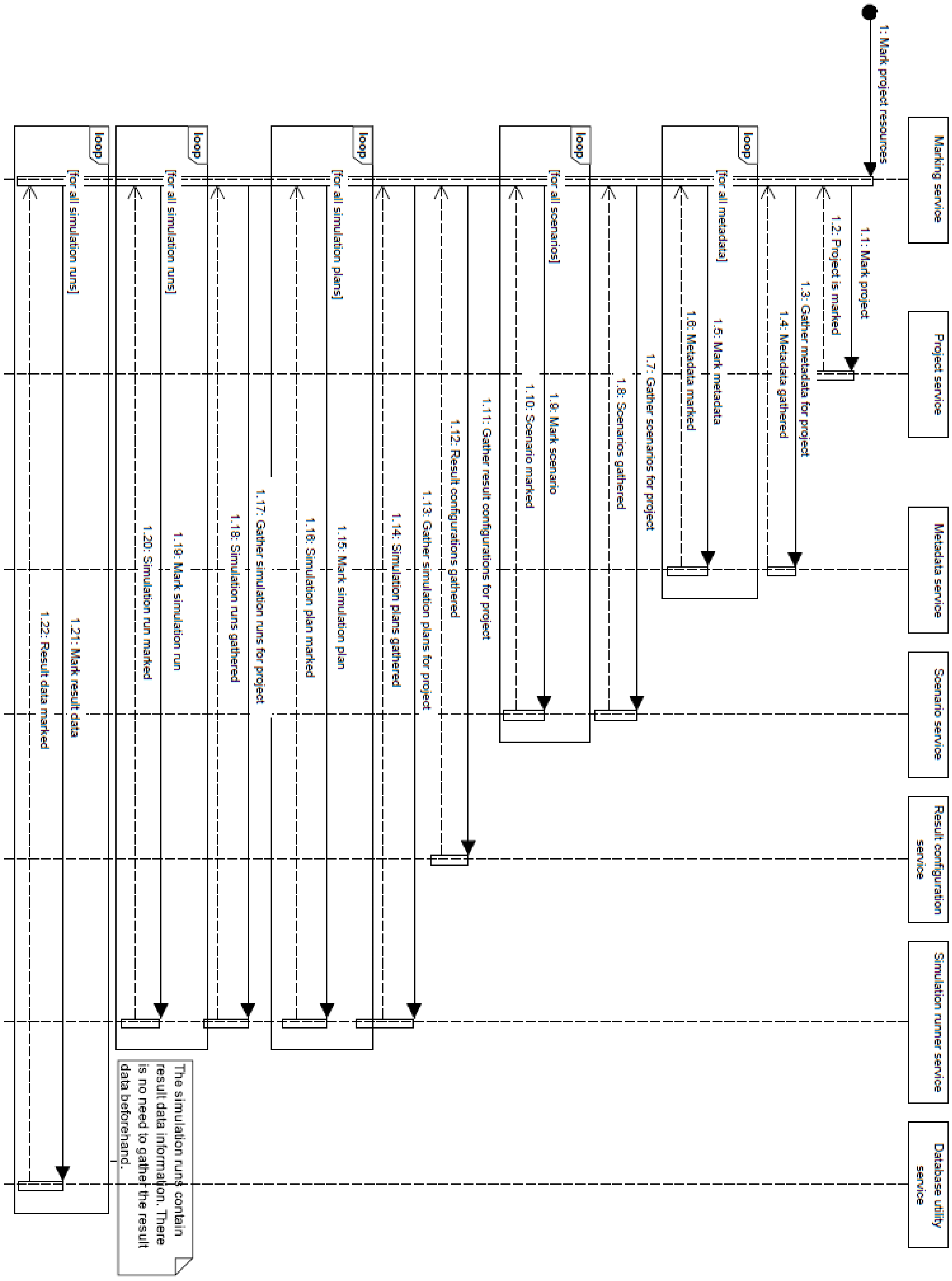

Marking service
Project service
Metadata service
Scenario service
Result configuration service
Simulation runner service
Database utility service
1: Mark project resources
1.1: Mark project
1.2: Project is marked
1.3: Gather metadata for project
1.4: Metadata gathered
loop
[for all metadata]
1.5: Mark metadata
1.6: Metadata marked
1.7: Gather scenarios for project
1.8: Scenarios gathered
loop
[for all scenarios]
1.9: Mark scenario
1.10: Scenario marked
1.11: Gather result configurations for project
1.12: Result configurations gathered
1.13: Gather simulation plans for project
1.14: Simulation plans gathered
loop
[for all simulation plans]
1.15: Mark simulation plan
1.16: Simulation plan marked
1.17: Gather simulation runs for project
1.18: Simulation runs gathered
loop
[for all simulation runs]
1.19: Mark simulation run
1.20: Simulation run marked
loop
[for all simulation runs]
1.21: Mark result data
1.22: Result data marked
The simulation runs contain result data information. There is no need to gather the result data beforehand.

Figure 12: Sequence diagram representing the workflow of the marking service

The marking process is done in steps. If the user would like to mark the contents of a project, the project must be marked first, after that the uploads, then scenarios and so on according to the dependency tree, specified in **Figure 6**. This logic together with the changes from **Section 4.2.1** and **Section 4.5.1** prevent the race conditions defined in **Figure 8**. While the marks are in place, the marked resources cannot be altered and their usage is restricted to other services.

Most services have a database connection, which allows them to persist their current progress. In the event that the service is killed and rescheduled, the service can continue or abort the previous process. A similar strategy is used for the marking service, which has a connection to the MongoDB database. MongoDB is chosen, because of its simplicity, high scalability and wider usage within the MARS Cloud.

The service creates a mark session, which is persisted into the MongoDB database and it collects all marked dependent resources. In order to judge the completeness of a marking process, the following states are introduced to the mark session. **Figure 13** summarizes the states of the mark session and their transitions.

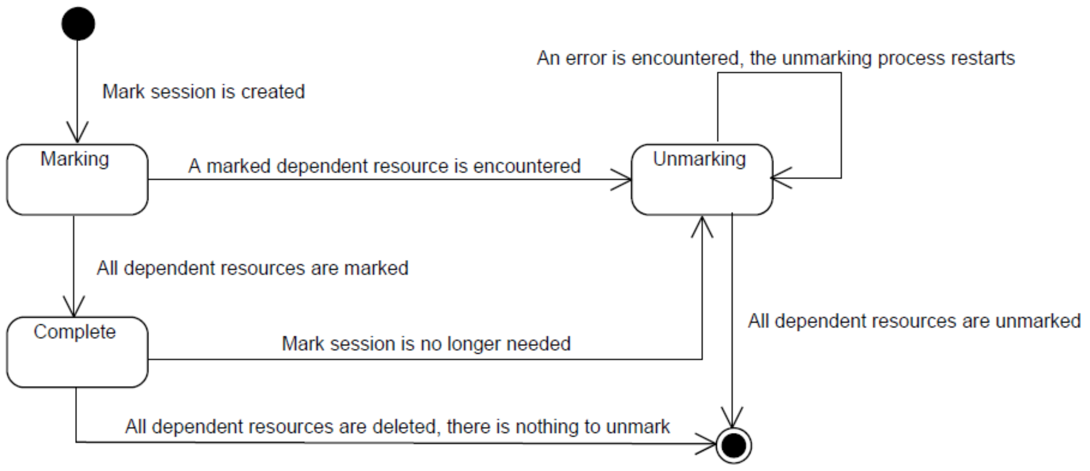

An error is encountered, the unmarking process restarts
Mark session is created
Marking
A marked dependent resource is encountered
Unmarking
All dependent resources are marked
All dependent resources are unmarked
Complete
Mark session is no longer needed
All dependent resources are deleted, there is nothing to unmark

Figure 13: State diagram representing all states of a mark session and their transitions

Each step of the marking process (refer to **Figure 12**) is immediately persisted into the mark session and the MongoDB database. A similar strategy is used for the unmarking process, but there as soon as a resource is unmarked; it is removed from the mark session and the MongoDB database.

Figure 14 shows a class diagram of the MarkSessionModel class together with the marked resources, which are represented by the DependentResourceModel class.

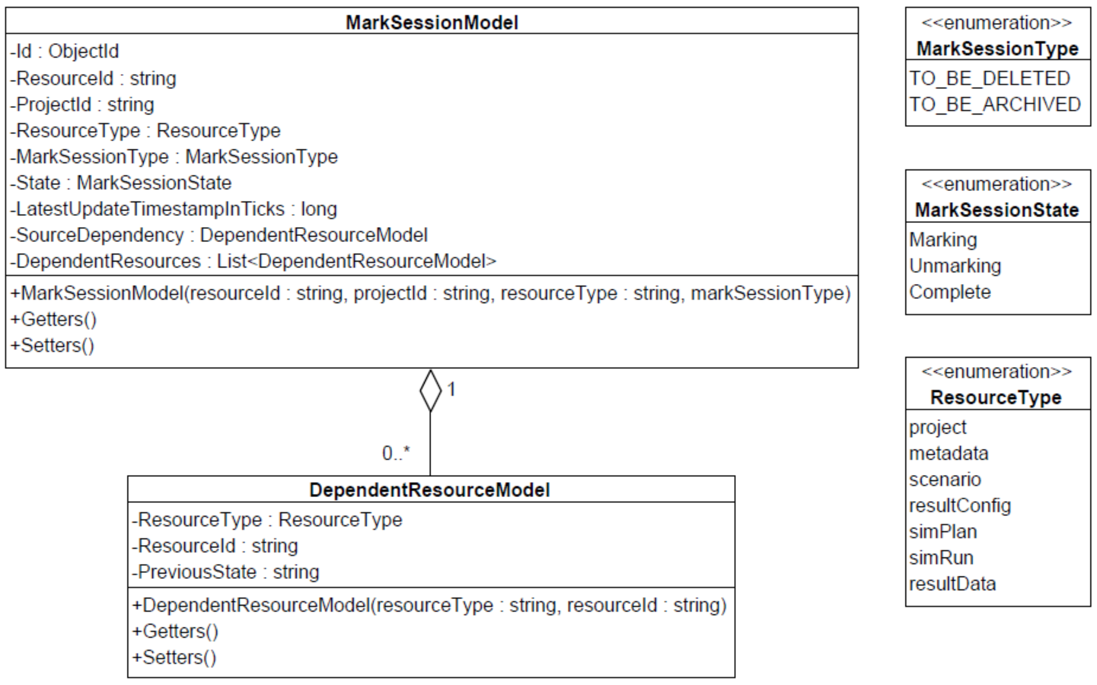

MarkSessionModel
-Id : ObjectId
-ResourceId : string
-ProjectId : string
-ResourceType : ResourceType
-MarkSessionType : MarkSessionType
-State : MarkSessionState
-LatestUpdateTimestampInTicks : long
-SourceDependency : DependentResourceModel
-DependentResources : List<DependentResourceModel>
+MarkSessionModel(resourceId : string, projectId : string, resourceType : string, markSessionType)
+Getters()
+Setters()
1
0..*
DependentResourceModel
-ResourceType : ResourceType
-ResourceId : string
-PreviousState : string
+DependentResourceModel(resourceType : string, resourceId : string)
+Getters()
+Setters()
<<enumeration>>
MarkSessionType
TO_BE_DELETED
TO_BE_ARCHIVED
<<enumeration>>
MarkSessionState
Marking
Unmarking
Complete
<<enumeration>>
ResourceType
project
metadata
scenario
resultConfig
simPlan
simRun
resultData

Figure 14: Class diagram representing the MarkSessionModel and DependentResourceModel classes

The marking service incorporates a hosted service (refer to **Section 4.7.1**). The hosted service will verify if any mark sessions are incomplete using the 'State' and the 'LastesUpdateTimestampInTicks' attributes, part of the MarkSessionModel class. The logic specified so far is used to resolve all terminated marking processes due to a sudden change in the life cycle of the marking service.

Figure 15 demonstrates a case, where the marking service is suddenly killed during a marking process. Upon starting the service again, the hosted service removes the incomplete mark session and the marked dependent resources are unmarked as well.

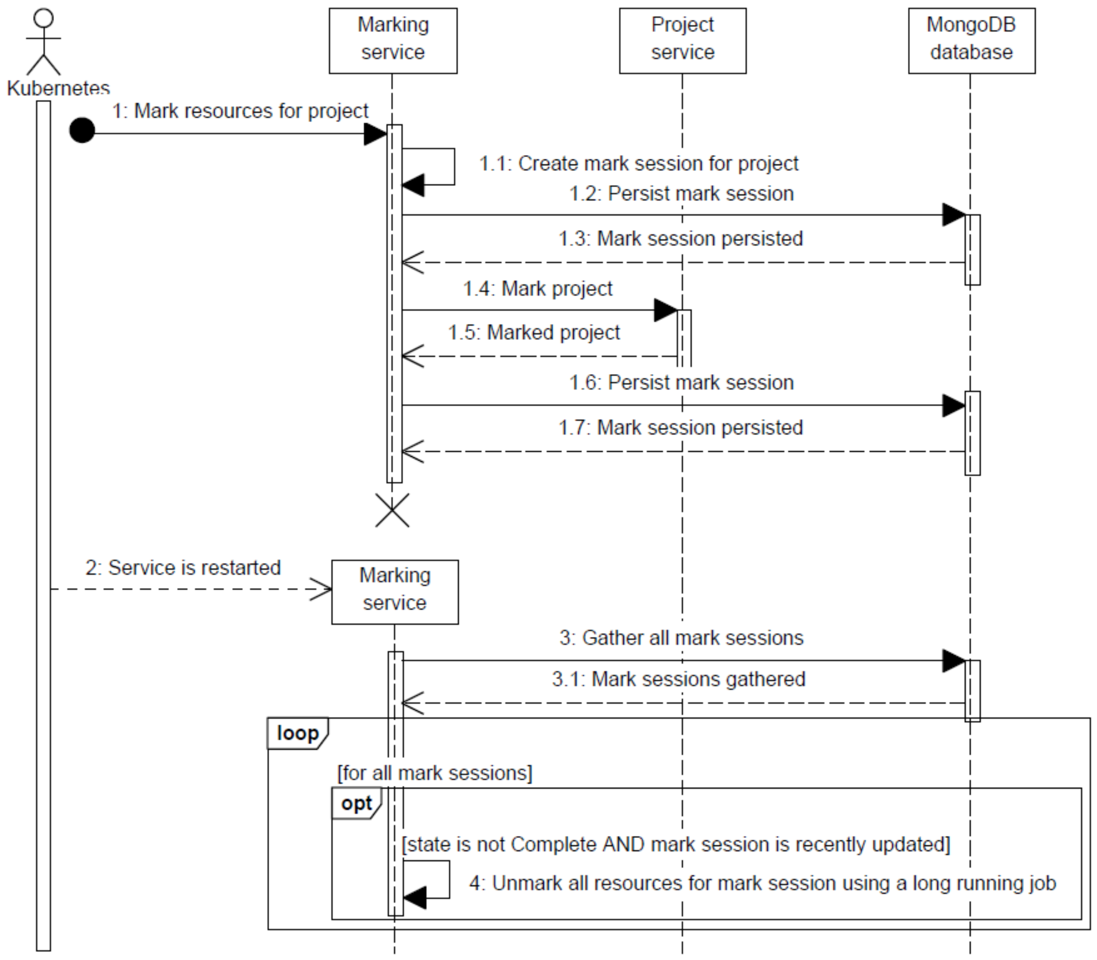

Kubernetes
Marking service
Project service
MongoDB database
1: Mark resources for project
1.1: Create mark session for project
1.2: Persist mark session
1.3: Mark session persisted
1.4: Mark project
1.5: Marked project
1.6: Persist mark session
1.7: Mark session persisted
2: Service is restarted
Marking service
3: Gather all mark sessions
3.1: Mark sessions gathered
loop
[for all mark sessions]
opt
[state is not Complete AND mark session is recently updated]
4: Unmark all resources for mark session using a long running job

Figure 15: Sequence diagram representing a stopped marking process and its recovery after a sudden change in the marking service life cycle

In addition, the marking service uses long running background jobs (refer to **Section 4.7.2**) for the unmarking process. At any point in time, any of the services in the MARS Cloud could be unavailable. Since the marks impose restrictions on the usability of resources in the MARS system, they must be removed if they are not in use. The long running background jobs make continuous attempts to remove the marks until the job is successful.

Furthermore, the marking service creates mark sessions according to the 'first come, first serve' policy. This means that the service, which first requests resources, would always get them. All other services would have to retry later if they would like to get access to the same resources. In most common cases, this policy is sufficient. However, there are scenarios where this policy is unfair at allocating resources to some services.

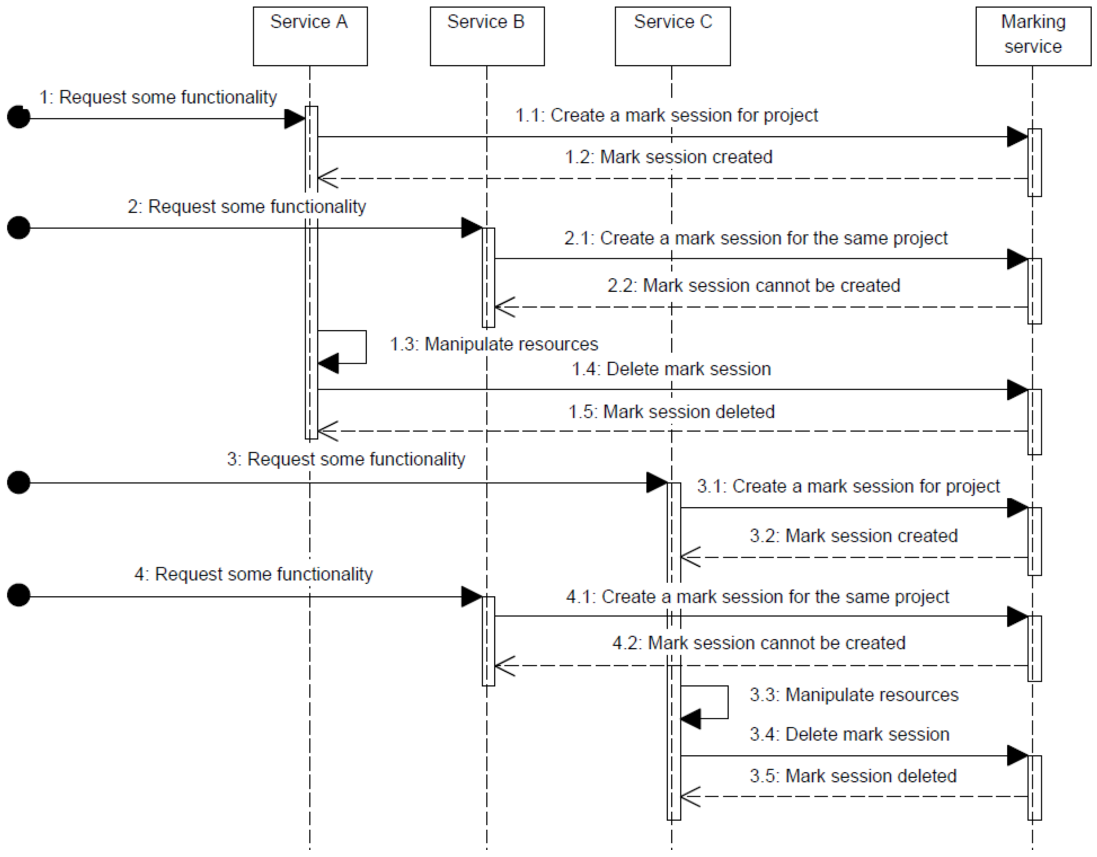

Service A
Service B
Service C
Marking service
1: Request some functionality
1.1: Create a mark session for project
1.2: Mark session created
2: Request some functionality
2.1: Create a mark session for the same project
2.2: Mark session cannot be created
1.3: Manipulate resources
1.4: Delete mark session
1.5: Mark session deleted
3: Request some functionality
3.1: Create a mark session for project
3.2: Mark session created
4: Request some functionality
4.1: Create a mark session for the same project
4.2: Mark session cannot be created
3.3: Manipulate resources
3.4: Delete mark session
3.5: Mark session deleted

Figure 16: Sequence diagram representing unfair allocation of resources by the marking service

Figure 16 presents a scenario where multiple services are competing to create a mark session for the same project. The specific order of requests, does not allow Service B to perform at all. The conclusion can be drawn that the ‚first come, first serve‘ policy is unfair in this precise case. To solve this issue, a distributed queue data structure can be used, which would store the denied services. The creation of mark sessions would be coordinated with respect to the services in the queue. With this policy, Service C would be denied and placed on the queue, then Service B would get a chance to exetuce the requested funationality (refer to **Figure 16**). However, this approach raises further issues. A queue utlity built in mind for distribution must be used. Otherwise, questions regarding the manipulation of the queue arise by multiple replicas of the marking service. Furthermore, most requests in the system are triggerred upon user interaction. There is no quarantee that a certain functionality would be requested once again after it is denied, it all depends on the user. This means that the queue should remove the services, which are not requesting further allocation of resources.

Due to the complexity of realizing a fair policy, the time limitation and the fact that a scenario, like the one defined in **Figure 16** is extremely rare to appear in real life curcumstances, this topic is no longer investigated and it will be proposed as future work.

4.5.3 Optimizations

The marking service goes through the resource dependency tree at least once in order to find and mark all resources belonging to a root resource. By passing the mark session, which contains all marked resources, to the caller services, the guarantee can be made that no other service will have to go through the dependency tree once again. This optimization saves unnecessary calls to multiple services.

In addition, since most of the attributes are used internally by the marking service (refer to **Figure 14**), a data transfer object (DTO) [20] is used to hide all specific implementation details of the mark session from the external services.

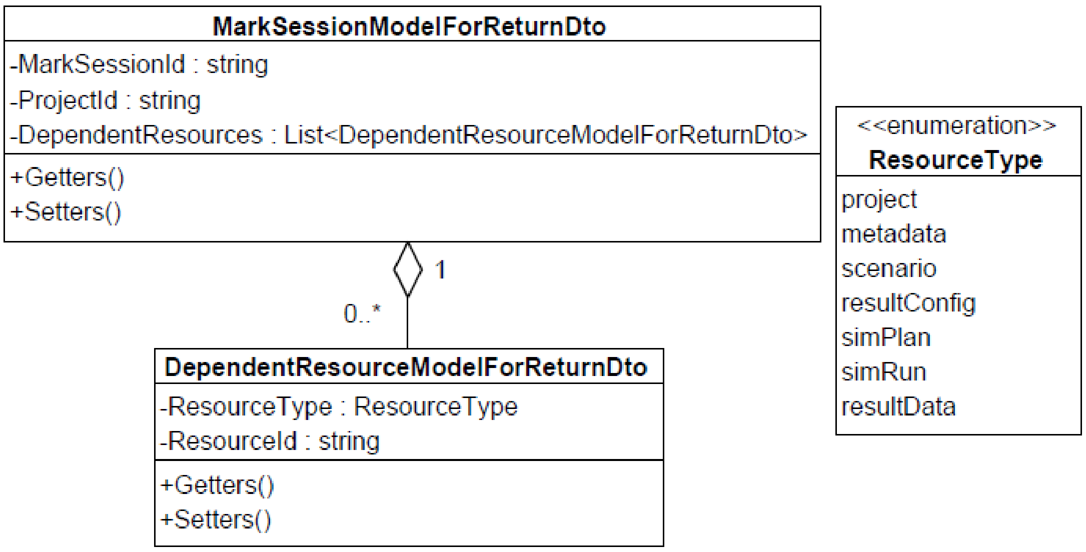


Figure 17: Class diagram representing the MarkSessionForReturnDto and DependentResourceModelForReturnDto classes

Furthermore, the performance of the service is enhanced by performing all mark requests in parallel (refer to **Figure 12**), rather than waiting for the previous one to finish and then performing the next.

4.5.4 Scalability

Higher scalability is achieved for the service by enforcing the 'ResourceId' attribute, part of the MarkSessionModel class (refer to **Figure 14**), to be unique. Even if there are multiple replicas of the service running, only one of them can create a mark session for a specific resource at a time. Furthermore, the long running background jobs are also persisted to the MongoDB database. However, everything regarding this process is handled internally by the library; no additional functionality is needed.

4.5.5 Overview

The marking service is responsible for gathering and marking all dependent resources for a given root resources. It persists a mark session, which contains all information about a marking process to a MongoDB database. It integrates techniques such as hosted services and long running jobs to recover from sudden changes in the service life cycle.

The marking service exposes a REST API, which all other services use to communicate. REST is chosen over gRPC, because of its wider usage within the MARS system.

Figure 18 represents the available API endpoints for creating a mark session using the marking service. If the requested resources are available, a mark session is returned to the caller service. Otherwise, the caller service would have to retry at a later time.

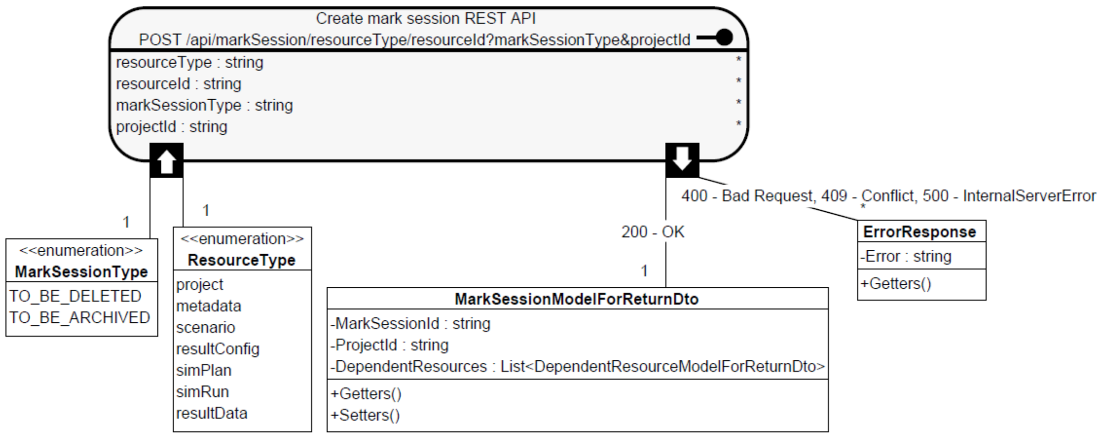


Figure 18: Class diagram representing the POST methods of the REST API associated with the mark sessions

Figure 19 presents the available API endpoints for deleting a mark session. Since the marks impose a restriction on the usability of resources within the system, they must be removed as soon as the resources are not needed. The deletion process for a mark session is started as a long running background job, which unmarks all dependent resources. However, since the deletion service deletes all dependent resources, an extra endpoint is added for deleting empty mark sessions. An empty mark session is simply removed from the MongoDB database.

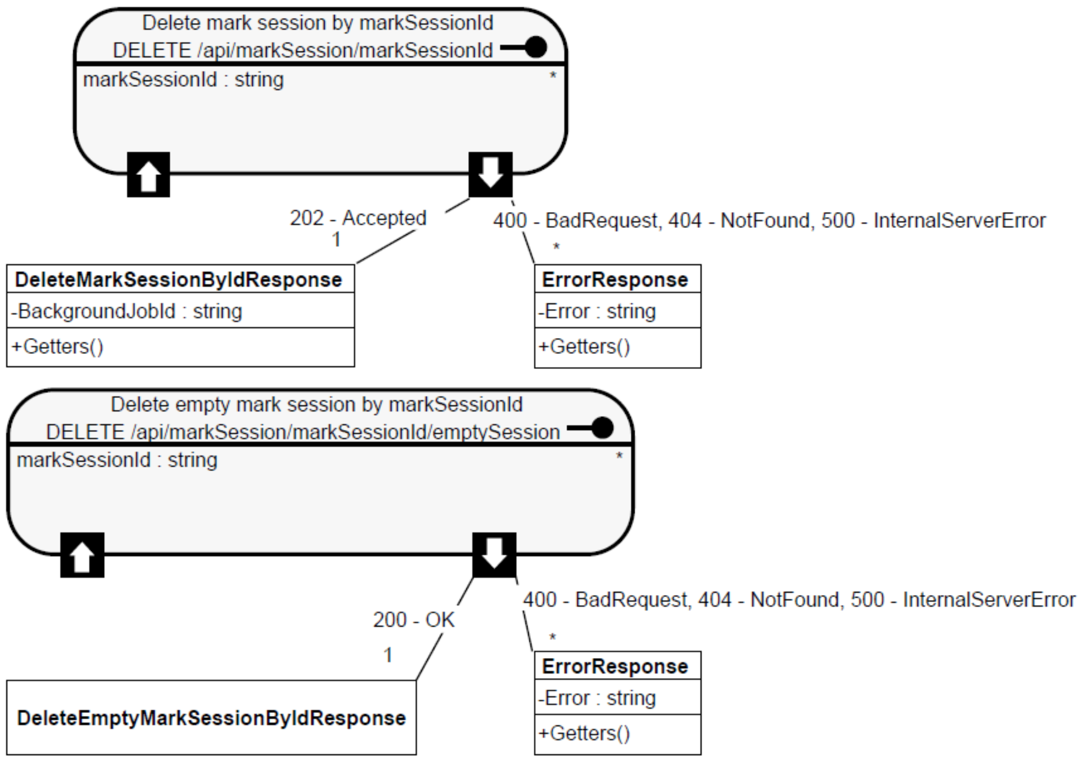

Delete mark session by markSessionId
DELETE /api/markSession/markSessionId
markSessionId : string
*
202 - Accepted
1
400 - BadRequest, 404 - NotFound, 500 - InternalServerError
*
DeleteMarkSessionByIdResponse
-BackgroundJobId : string
+Getters()
ErrorResponse
-Error : string
+Getters()
Delete empty mark session by markSessionId
DELETE /api/markSession/markSessionId/emptySession
markSessionId : string
*
200 - OK
1
400 - BadRequest, 404 - NotFound, 500 - InternalServerError
*
DeleteEmptyMarkSessionByIdResponse
ErrorResponse
-Error : string
+Getters()

Figure 19: Class diagram representing the DELETE methods of the REST API associated with the mark session

Furthermore, the marking service exposes API endpoints for retrieving and updating a mark session, together with an API endpoint for retrieving state information about a background job (refer to **Section 4.7.2**).

Table 3 presents a summary of the status codes, together with the possible reason to receive them from the marking service.

Status code	Reasoning
200 – OK	The desired operation has successfully been executed.
202 – Accepted	The desired operation is being processed, immediate feedback cannot be given.
400 – Bad Request	A required parameter is not specified. A wrong value is given for a required parameter.
404 – Not Found	The desired resource does not exist.
409 - Conflict	A mark session already exists for the resource. A marked resource is encountered during the marking process.
500 – Internal Server Error	A service, used internally by the marking service, is currently unavailable. An error has occurred within the marking service.

Table 3: Status codes together with their reasoning, produced by the marking service

4.6 Deletion service

4.6.1 Additional requirements

An additional requirement for the deletion service is that projects must not be deleted. This restriction is imposed by the archive service, which is developed in parallel. The archive service relies heavily on the project resource definition to perform archives and later archive restores. The user resource definition does not contain any information about the projects that each user has access to. This leads to difficulties for restoring user access to projects once they are deleted. Since the deletion service is restricted to deleting projects, it will remove the project contents instead. This would give users the ability to reuse their projects [19].

4.6.2 Workflow

The main purpose of the deletion service is to delete all dependent resources gathered in a mark session. Refer to **Figure 20** for an overview of the deletion process.

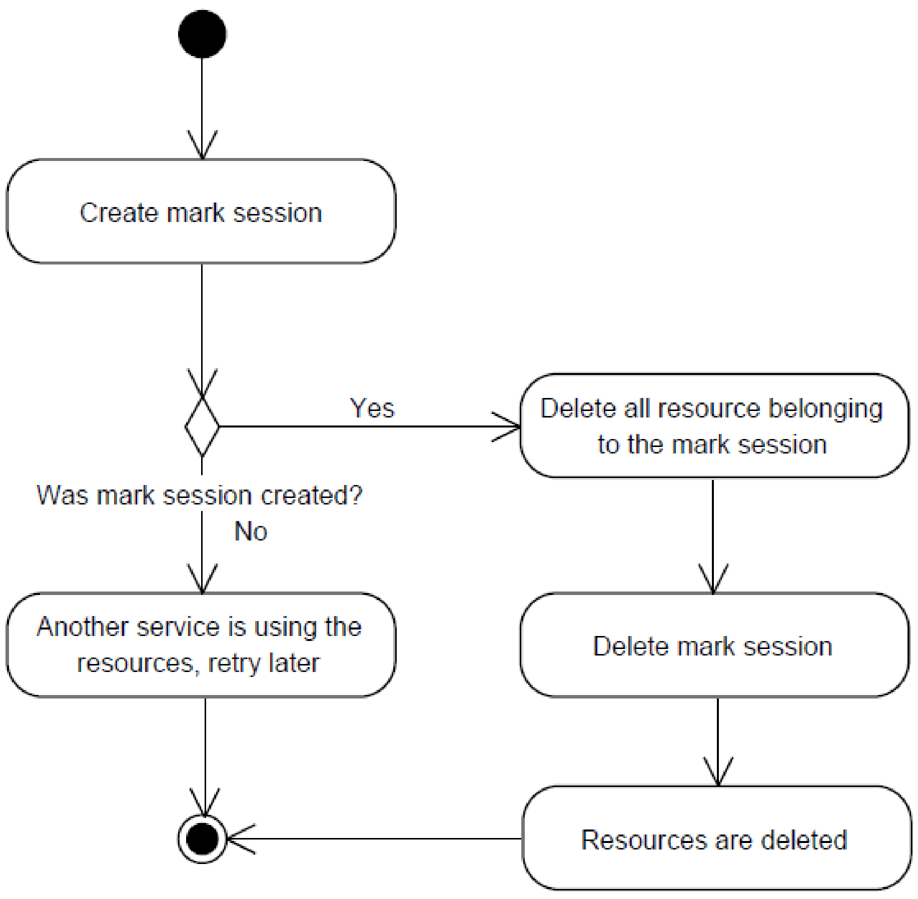

Create mark session
Yes
Delete all resource belonging to the mark session
Was mark session created?
No
Another service is using the resources, retry later
Delete mark session
Resources are deleted

Figure 20: Activity diagram representing the workflow of the deletion service

Figure 21 presents all services that are involved in the deletion process. Each step is represented by a call to an external service, where the response is awaited. Once the service responds, the program execution continues. However, if the response is awaited forever, there could be possible deadlocks. For this reason, the interfaces used for establishing the inner-service communication have a timeout for the requests. If the external service fails to reply within the given time, an exception is thrown.

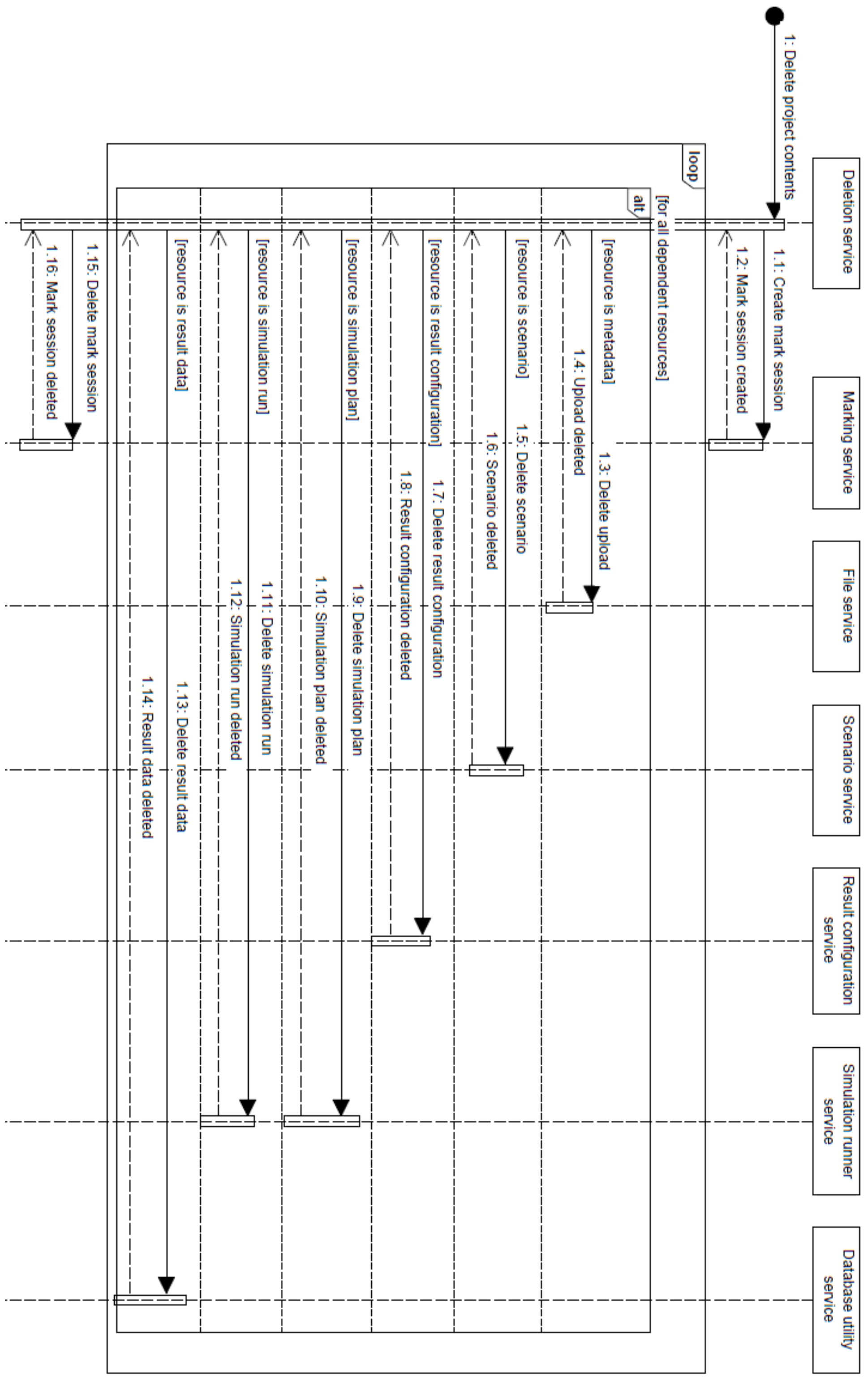

Deletion service
Marking service
File service
Scenario service
Result configuration service
Simulation runner service
Database utility service
1: Delete project contents
loop
[for all dependent resources]
alt
1.1: Create mark session
1.2: Mark session created
[resource is metadata]
1.3: Delete upload
1.4: Upload deleted
[resource is scenario]
1.5: Delete scenario
1.6: Scenario deleted
[resource is result configuration]
1.7: Delete result configuration
1.8: Result configuration deleted
[resource is simulation plan]
1.9: Delete simulation plan
1.10: Simulation plan deleted
[resource is simulation run]
1.11: Delete simulation run
1.12: Simulation run deleted
[resource is result data]
1.13: Delete result data
1.14: Result data deleted
1.15: Delete mark session
1.16: Mark session deleted

Figure 21: Sequence diagram representing the workflow of the deletion service

The deletion service also makes use of a hosted service. When the deletion service starts, the hosted service requests all mark session that should be deleted. Afterwards, it starts a long running job for deleting every one of them. At any point in time, any of the services could be unavailable. The deletion process stops only if all dependent resources are deleted.

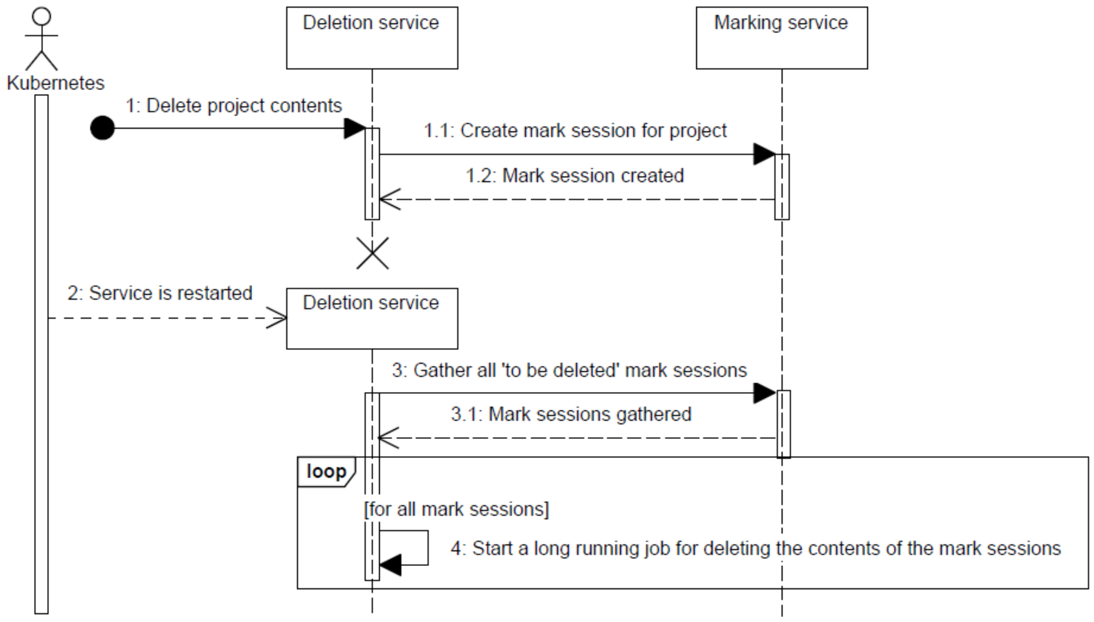

Kubernetes
Deletion service
Marking service
1: Delete project contents
1.1: Create mark session for project
1.2: Mark session created
2: Service is restarted
Deletion service
3: Gather all 'to be deleted' mark sessions
3.1: Mark sessions gathered
loop
[for all mark sessions]
4: Start a long running job for deleting the contents of the mark sessions

Figure 22: Sequence diagram representing a stopped deletion process and its recovery after a sudden change in the deletion service life cycle

Figure 22 illustrates a scenario in which the deletion service is killed right before it starts deleting resources from a mark session. Upon service restart, it requests all mark sessions that must be deleted and starts a long running background job for each of them.

4.6.3 Optimizations

To further improve the performance of the service, all delete requests are done in parallel (refer to **Figure 21**), rather than waiting for the previous one to finish and performing the next.

4.6.4 Scalability

The deletion service is stateless, nothing is persisted to a database. This makes the service highly scalable. An exception are the long running background jobs, but all related logic is handled internally by the library. The jobs are automatically restarted together with the service if they have not finished execution.

4.6.5 Overview

The main responsibility of the deletion service is to delete all dependent resources belonging to a mark session. It incorporates hosted services and long running jobs to recover from sudden changes in the service life cycle.

The deletion service exposes a REST API, which all other services use to communicate. REST is chosen over gRPC, because of its wider usage within the MARS system.

Figure 23 represents the main functionality of the deletion service exposed by a REST API.

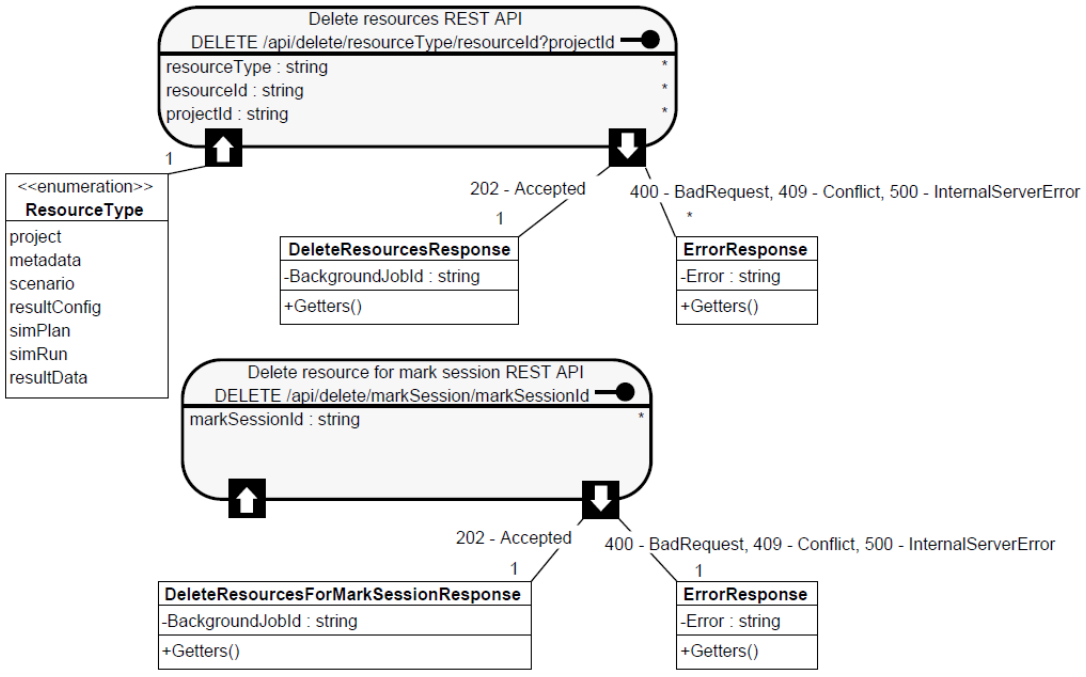

Delete resources REST API
DELETE /api/delete/resourceType/resourceId?projectId
resourceType : string
resourceId : string
projectId : string
1
<<enumeration>>
ResourceType
project
metadata
scenario
resultConfig
simPlan
simRun
resultData
202 - Accepted
1
400 - BadRequest, 409 - Conflict, 500 - InternalServerError
*
DeleteResourcesResponse
-BackgroundJobId : string
+Getters()
ErrorResponse
-Error : string
+Getters()
Delete resource for mark session REST API
DELETE /api/delete/markSession/markSessionId
markSessionId : string
202 - Accepted
1
400 - BadRequest, 409 - Conflict, 500 - InternalServerError
1
DeleteResourcesForMarkSessionResponse
-BackgroundJobId : string
+Getters()
ErrorResponse
-Error : string
+Getters()

Figure 23: Class diagram representing the DELETE methods of the REST API associated with the deletion service

Table 4 presents a summary of the status codes, together with the possible reason to recieve the specified status code from the deletion service.

Status code	Reasoning
202 – Accepted	The desired operation is being processed, immediate feedback cannot be given.
400 – Bad Request	A required parameter is not specified. A wrong value is given for a required parameter.
409 – Conflict	A mark session could not be created for the specified resource.
500 – Internal Server Error	A service, used internally by the deletion service, is currently unavailable. An error has occurred within the deletion service.

Table 4: Status codes together with their reasoning, produced by the deletion service

4.7 Design patterns

This section outlines common design patterns used for both, marking and deletion, services.

4.7.1 Hosted services

A hosted service provides a mechanism for running background tasks within the lifetime scope of an ASP.NET Core application. The hosted service is started on application start and stopped on application shut down [21].

Hosted services are used to resolve unfinished tasks due to a sudden change in the service's life cycle (refer to **Section 4.5.2** and **Section 4.6.2**).

4.7.2 Long running background jobs

A long running background job is a process that is executed in the background. Since the ASP.NET Core framework provides limited functionality for background processing, an external library is used called Hangfire.

Hangfire is an open-source software, used to perform background processing in ASP.NET Core applications. Some of its key advantages are simplicity, reliability and persistence. The library is quite easy to set up and use. Background jobs are regular static or instance method, no base classes or interfaces are required. The background jobs are created in a persistent storage. Old records are removed automatically. Furthermore, Hangfire takes care of re-trying a background job if the application is terminated [22].

Long running background jobs are used for the unmarking process, by the marking service, and for the deletion, by the deletion service. Both processes rely on several external services, which can be unavailable at any point in time, because of the distributed nature of the MARS system. Both processes are designed to be atomic and they must run until completion. If there is an error encountered, the processes are restarted within a short period of time. It is of high importance for the unmarking process to finish, because it imposes restrictions on the usability of resources. As for the deletion process, the deletion must fully complete or it should not start at all.

Since the long running background jobs are not deterministic, additional functionality is needed to monitor their progress. Hangfire already takes care of persisting the jobs together with their state. However, external services need to access this information in order to adjust their functionality accordingly. Hangfire has multiple states for each job; nevertheless, since the jobs are running until completion, a simplified state model is introduced in **Figure 24**.

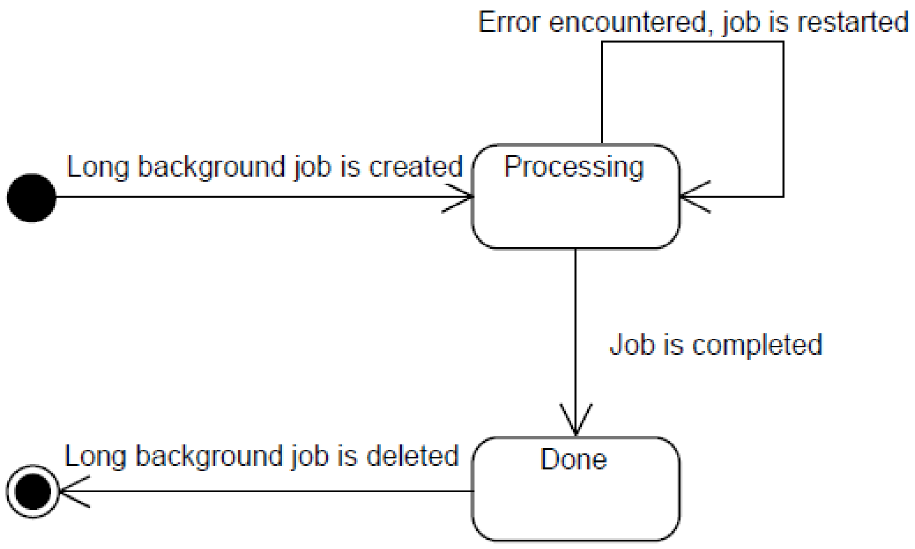

Error encountered, job is restarted
Long background job is created
Processing
Job is completed
Long background job is deleted
Done

Figure 24: State diagram representing the states of a long running background job and the state transitions

Both services expose an API endpoint, which enables external services to get the state information about any background job.

4.7.3 Dependency injection

Dependency injection is a design pattern that allows instances of objects to be passed to other objects, which require them at runtime. Instead of expecting a concrete implementation of a dependency, all objects can expect an interface, which abstracts all implementation details. This leads to higher level on modularity and maintainability in the applications. In addition, mocking of dependencies for testing purposes becomes easier [21].

ASP.Net Core implements dependency injection as a first-class citizen in its infrastructure and has an Inversion of Control (IoC) container built into its core. All required dependencies and their instances must be registered in the IoC container. The IoC container cakes care of passing the correct instance of an object requested during runtime [21].

5 Implementation

5.1 Infrastructure changes

The infrastructure changes required by **Section 4.2.1** and **Section 4.5.1** have been implemented into the MARS Cloud. An emphasis has been given to the requirement that all services must obey the marks of the resources. In addition, the verification has been made that the newly added changes did not break old working functionality.

5.2 Marking service

Appendix A shows the implementation of the core feature of the marking service, namely the marking process (refer to **Figure 12** for an overview). The process finds all resources belonging to a project and marks them. The resources are converted into requests, which are executed in parallel. Afterwards, all successful results are included into the mark session and persisted into MongoDB.

Refer to the mars-marking-svc project, part of the mars-marking-svc solution, for all specific implementation details regarding the marking service.

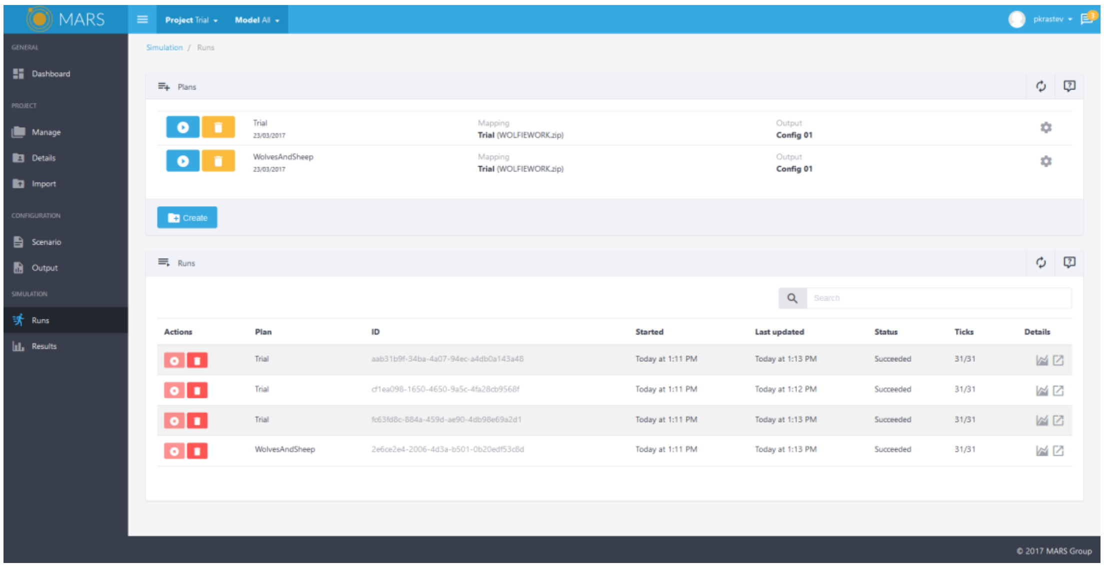
5.3 Deletion service

The implementation of the core feature of the deletion service, namely the deletion process (refer to **Figure 21** for an overview), is presented in **Appendix B**. The deletion process takes place after a mark session has been created by the marking service. The mark sessions contains all marked dependent resources. The process is started as a long running background job, which does not end until it is completed. The delete requests for all resources, part of the mark session, are done in parallel to yield better performance. After all resources are deleted, the mark session is also removed.

All specific implementation details for the service can be found in the mars-deletion-svc project, part of the mars-deletion-svc solution.

5.4 Integration into the MARS system

The deletion service has been integrated into the MARS Web UI and MARS API Gateway as required by **Section 3.1.3**.



	Trial 23/03/2017	Mapping **Trial** (WOLFIEWORK.zip)	Output **Config 01**
	WolvesAndSheep 23/03/2017	Mapping **Trial** (WOLFIEWORK.zip)	Output **Config 01**

Actions	Plan	ID	Started	Last updated	Status	Ticks	Details
	Trial	aab31b9f-34ba-4a07-94ec-a4db0a143a48	Today at 1:11 PM	Today at 1:13 PM	Succeeded	31/31	
	Trial	cf1ea098-1650-4650-9a5c-4fa28cb9568f	Today at 1:11 PM	Today at 1:12 PM	Succeeded	31/31	
	Trial	fc63fd8c-884a-459d-ae90-4db98e69a2d1	Today at 1:11 PM	Today at 1:13 PM	Succeeded	31/31	
	WolvesAndSheep	2e6ce2e4-2006-4d3a-b501-0b20edf53c8d	Today at 1:11 PM	Today at 1:13 PM	Succeeded	31/31	

Figure 25: Screenshot of the MARS Web UI illustrating the simulation plans and simulation runs view

Figure 25 illustrates the view with the simulation plans and simulation runs, part of the MARS Web UI. The deletion service is accessible through visual elements in the form of buttons. Such elements have been added to all other resources. In addition, the status of the deletion process is presented to the user in the form of notifications.

6 Testing

6.1 Unit tests

The marking and deletion services have a UnitTests project, part of each service's solution, which contains all unit tests. All methods for both services have at least one unit test. The unit tests are conducted with the help of the xUnit tool [23]. Testing a single unit of work can become complicated if it has external dependencies. To deal with this issue, a mocking framework is used, Moq [24], which abstracts the external dependencies. This approach is very successful, because all dependencies are supplied through dependency injection.

Appendix C shows a unit test, which tests the method for marking uploads. Internally the method retrieves the metadata for an upload from the metadata service, then checks if the upload is already marked. If not, the method performs a mark request, again to the metadata service. If an error is encountered or the upload is already marked, an exception is thrown. This method relies on several API calls to the metadata service. However, the metadata service is not accessible into the UnitTests project, it is available only in the MARS Cloud. To resolve this issue, the call and response to the metadata service are mocked. The mocked response returns an already marked upload. This test verifies if an exception is thrown when an already marked resource is encountered. This requirement is of high importance, because it removes the possibility that two services can manipulate the same resources at the same time.

6.2 Testing environment resembling the MARS Cloud

To properly test, the marking and deletion services, a testing environment that resembles the MARS Cloud must be made. Both services heavily rely on functionality and data from external ones. Further testing strategies, such as integration tests, would not be possible without such an environment.

Docker Compose is a tool for defining, launching and managing services. A service is defined as one or more replicas of a Docker container. Docker compose enables developers to describe full environments and service component interactions [25]. Docker compose is used to start an environment, similar to the MARS Cloud, with all services from **Figure 4**. However, the marking and deletion services need data to function. Two extra services are introduced to the testing environment that seed data into the MongoDB database. A major difficulty was encountered when trying to seed data into the PostgresSQL database. The complexity and time limitation led to not including the project service, which relies on PostgresSQL, into the tests ahead.

6.3 Integration tests

Integration testing ensures that the components of an application work as expected when they are all put together. Unlike unit testing, integration testing involves testing all implementations of individual components together, which may include infrastructure concerns [21].

The marking and deletion services have an IntegrationTests project, part of each service's solution, which contains all integration tests. Inside each project, the testing environment is specified within the **docker-compose.yaml** file. All integration tests can be run locally using the **run-tests-locally.sh** script. Upon starting the script, the testing environment is build, together with all services. Data is seeded into the MongoDB database and the integration tests within the project are executed. The integration tests are also conducted with the help of the xUnit tool [23]. However, the mocking framework has been excluded, since all services, part of the MARS Cloud, are accessible in the testing environment.

Appendix D presents an integration test, which tests the creation of a mark session. Internally the method finds all resources based on a root resource and marks them. This functionality is very similar to the workflow described in **Figure 12** and the implementation included in **Appendix A**. When the mark session is created, the verification is made whether the root resource is actually marked. The purpose of a mark session is to ensure that all resources are gathered and marked.

6.4 End-to-end tests & results

End-to-end tests for both services have been conducted manually. The complexity of the MARS system, including the creation of an automated test for the MARS Web UI, led to performing tests as an end user inside the MARS beta deployment. This is possible, because the deletion service is integrated into the MARS Web UI. Since the deletion service relies heavily on the marking service to perform its functionalities, starting a deletion process from the MARS Web UI involves a complete end-to-end test for both services.

Prior to all tests, resources have been uploaded or created to a project. Since the result data, produced by simulation runs, are largest in volume, they are the most widely used resource for all test cases. Each end-to-end test is conducted by deleting the contents of a project. **Table 5** summarizes the number of resources belonging to a project together with the duration of the marking and the deletion processes.

Service	Process	Number of resources	Duration in seconds
Marking	Marking	10	1.16
Deletion	Deletion	10	0.25
Marking	Marking	25	1.17
Deletion	Deletion	25	1.05
Marking	Marking	50	1.41
Deletion	Deletion	50	2.22

Table 5: Performance metrics for the marking and deletion services

The data for the performance metrics of both services has been taken from the each service’s log output. The marking and deletion services have a logging strategy that logs all requests and background jobs together with the duration of the process.

7 Conclusion

The work presented in this document provides a distribution deletion solution for the MARS simulation framework. MARS is a distributed system that had adopted the microservice architecture pattern. The functionality of the whole system is split into multiple independent microservices that communicate via a network interface. The current infrastructure of the MARS system has been examined in-depth for the purpose of providing the optimum solution, in the form of a microservice.

The main challenge encountered when designing the deletion microservice was concurrency. Multiple services and users can have access to the same resources at the same time. This imposes risks of race conditions when performing deletion. To resolve this issue the concept of marking resources has been introduced. Prior to deleting any resources, they are marked as ‘to be deleted’. This ensures that if all services obey the marks, the concurrent access to the resources can be restricted. For the purpose of modularity and reusability, the marking logic is placed into a separate microservice, called the marking service. This permits other microservices to easily reuse functionality from the marking service that include avoiding issues related to concurrent requests for resources together with avoiding possible race conditions within the system.

Finally, for the purpose of usability, the deletion service has been integrated into the MARS Web UI. This allows users to delete any resources they have uploaded or created, which in return gives them the possibility to manage their cloud storage in a more efficient manner.

7.1 Future work

The marking service allocates resources to other services using the ‘first come, first serve’ policy. In most use cases, this approach is sufficient. However, is it worth to mention that in certain scenarios, this policy is unfair and certain services might not get a chance to execute some requested functionality (refer to **Figure 16**). A solution that utilizes a distributed queue is proposed. If a service is denied resources, it will be placed on the queue. The next allocation of resources by the marking service will be coordinated with respect to the services within the queue. However, there is no guarantee that any service within the queue would request resources once again, since most services are triggered upon user interaction. The queue must be able to remove services that do not request resources once again.

Furthermore, the MARS Web UI could be improved. In the current version, only the root resource is removed immediately upon successful deletion. All other dependent resources remain and the user is forced to hard reload the page.

References

[1] George Coulouris, Jean Dolimore, Tim Kindberg, Gordon Blair, "Distributed Systems, Concepts and design", 5th ed., Addison-Wesley, 2012, ISBN: 978-0-13-214301-1.

[2] Christian Hüning, Mitja Adebahr, Thomas Thiel-Clemen, Jan Dalski, Ulfia Lenfers, Lukas Grundmann, "Modeling & Simulation as a Service with the Massive Multi-Agent System MARS," in *Agent-Directed Simulation Symposium*, Pasadena, California, 2016, ISBN: 978-1-5108-2315-0.

[3] Andrew Tanenbaum, Maarten van Steen, "Distributed Systems", 3rd ed., Maarten van Steen, 2017, ISBN: 978-90-815406-2-9.

[4] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina, "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering,* pp. 195-216, 2017, ISBN: 978-3-319-67424-7.

[5] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report," *Communications in Computer and Information Science,* vol. 567, pp. 201-215, 2015, ISBN: 978-3-319-33313-7.

[6] Paolo Di Francesco, Ivano Malavolta, Patricia Lago, "Migrating towards Microservice Architectures: an Industrial Survey," in *IEEE International Conference on Software Architecture*, Seattle, USA, 2018.

[7] Mojtaba Shahin, "Microservices: Architecting for DevOps and Continuous Deployment," in *IEEE International Conference on Software Architecture*, Seattle, USA, 2018.

[8] Chris Richardson, Floyd Smith, "Microservices, From Design to Deployment", NGINX Inc., 2016.

[9] Sam Newman, "Building Microservices: Designing Fine-Grained Systems", O'Reilly Media, 2014, ISBN: 978-1-491-95035-7.

[10] Christian Hüning, "Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS", Hamburg: HAW Hamburg, 2016.

[11] Daniel Glake, Julius Weyl, Carolin Dohmen, Christian Hüning, Thomas Clemen, "Modeling through model transformation with MARS 2.0," in *Agent-Directed Simulation Symposium* , Virginia Beach, Virginia, 2017.

[12] Kristina Chodorow, "MongoDB, The Definitive Guide", 2nd ed., O'Reilly Media, Inc., 2013, ISBN: 978-1-449-34468-9.

[13] Regine Obe, Leo Hsu, "PostgreSQL: Up and Running", 2nd ed., O'Reilly Media, Inc., 2015, ISBN: 978-1-449-37319-1.

[14] Sanjay Patni, "Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS", Apress Media LLC, 2017, ISBN: 978-1-4842-2665-0.

[15] Google Inc., "About gRPC," [Online]. Available: https://grpc.io/about/. [Accessed 22 07 2018].

[16] Google Inc., "gRPC Concepts," [Online]. Available: https://grpc.io/docs/guides/concepts.html#service-definition. [Accessed 22 07 2018].

[17] James Turnbull, "The Docker Book", James Turnbull, 2014, ISBN: 978-0-9888202-0-3.

[18] Jonathan Baier, "Getting Started with Kubernetes", Pack Publishing Ltd., 2015, ISBN: 978-1-78439-403-5.

[19] Prannoy Mulmi, "Design and Implementation of an Archive Microservice solution for the Multi-Agent Research and Simulation Distributed System", Hamburg: HAW Hamburg, 2018.

[20] Robert C. Martin, "Clean Code, A Handbook for Agile Software Craftsmanship", Pearson Education, Inc., 2009, ISBN: 978-0-13-235088-4.

[21] Fanie Reynders, "Modern API Design with ASP.NET Core 2, Building Cross-platofrm Back-End Systems", Apress, 2018, ISBN: 978-1-4842-3518-8.

[22] Sergey Odinokov, "Hangure Overview," Hangfire, [Online]. Available: https://www.hangfire.io/. [Accessed 22 07 2018].

[23] "About xUnit.net," .NET Foundation, [Online]. Available: https://xunit.github.io/. [Accessed 22 07 2018].

[24] "Moq," GitHub Inc., [Online]. Available: https://github.com/moq/moq4. [Accessed 22 07 2018].

[25] Jeff Nickoloff, "Docker in Action", Manning Publications Co., 2016, ISBN: 978-1633430235.

Appendix A

The following code snippet has been taken from **DependentResourceHandler.cs** class, which can be found in the path: **../mars-marking-svc/mars-marking-svc/DependentResouce**.

```
private async Task MarkResourcesForProjectMarkSession(
    MarkSessionModel markSessionModel
)
{
    var projectId = markSessionModel.ProjectId;

    markSessionModel.SourceDependency = await _projectClient.MarkProject(projectId);
    await _markSessionRepository.Update(markSessionModel);

    var metadataForProject = await _metadataClient.GetMetadataForProject(projectId);
    await MarkResourcesThenUpdateMarkSession(metadataForProject, projectId, markSessionModel);

    var scenariosForProject = await _scenarioClient.GetScenariosForProject(projectId);
    await MarkResourcesThenUpdateMarkSession(scenariosForProject, projectId, markSessionModel);

    var resultConfigsForMetadata = new List<ResultConfigModel>();
    foreach (var metadataModel in metadataForProject)
    {
        resultConfigsForMetadata.AddRange(
            await _resultConfigClient.GetResultConfigsForMetadata(metadataModel.DataId)
        );
    }
    await MarkResourcesThenUpdateMarkSession(resultConfigsForMetadata, projectId,
markSessionModel);

    var simPlansForProject = await _simPlanClient.GetSimPlansForProject(projectId);
    await MarkResourcesThenUpdateMarkSession(simPlansForProject, projectId, markSessionModel);

    var simRunsForProject = await _simRunClient.GetSimRunsForProject(projectId);
    await MarkResourcesThenUpdateMarkSession(simRunsForProject, projectId, markSessionModel);

    await MarkResultDataThenUpdateMarkSession(simRunsForProject, markSessionModel);
}
```

Appendix B

The following code snippet has been taken from **MarkSessionHandler.cs** class, which can be found in the path: **../mars-deletion-service/mars-deletion-service/MarkSession**.

```
public async Task StartDeletionProcess(
    string markSessionId
)
{
    var isMarkSessionDeleted = false;
    var taskExecutionDelayInSeconds = 1;
    var restartCount = 0;
    var stopwatch = new Stopwatch();

    while (!isMarkSessionDeleted)
    {
        try
        {
            _loggerService.LogBackgroundJobInfoEvent(
                $"Deletion job for mark session with id: {markSessionId} will start in
{taskExecutionDelayInSeconds} second/s, restart count: {restartCount}"
            );
            await Task.Delay(TimeSpan.FromSeconds(taskExecutionDelayInSeconds));
            stopwatch.Start();

            var markSessionModel = await
_markingServiceClient.GetMarkSessionById(markSessionId);
            await
_dependantResourceHandler.DeleteDependantResourcesForMarkSession(markSessionModel);
            await _markingServiceClient.DeleteEmptyMarkingSession(markSessionId);

            stopwatch.Stop();
            isMarkSessionDeleted = true;
        }
        catch (MarkSessionDoesNotExistException)
        {
            stopwatch.Stop();
            isMarkSessionDeleted = true;
        }
        catch (Exception e)
        {
            stopwatch.Stop();
            _loggerService.LogBackgroundJobErrorEvent(stopwatch.Elapsed.TotalSeconds, e);
            taskExecutionDelayInSeconds = taskExecutionDelayInSeconds * 2 %
MaxDelayForJobInSeconds;
            restartCount++;
        }
    }

    _loggerService.LogBackgroundJobInfoEvent(
        stopwatch.Elapsed.TotalSeconds,
        $"Deletion job for mark session with id: {markSessionId} completed!"
    );
}
```

Appendix C

The following code snippet has been taken from **MetadataClientTests.cs** class, which can be found in the path: **../mars-marking-svc/UnitTests/ResourceTypes/Metadata.**

```
[Fact]
public async void MarkMetadata_ToBeDeletedMetadataModel_ThrowsException()
{
    // Arrange
    var httpResponseMessage = new HttpResponseMessage
    {
        StatusCode = HttpStatusCode.OK,
        Content = new StringContent(MetadataModelDataMocks.MockToBeDeletedMetadataModelJson)
    };
    var httpService = new Mock<IHttpService>();
    httpService
        .Setup(m => m.GetAsync(It.IsAny<string>()))
        .ReturnsAsync(httpResponseMessage);
    var metadataClient = new MetadataClient(httpService.Object);
    Exception exception = null;

    try
    {
        // Act
        await metadataClient.MarkMetadata(It.IsAny<string>());
    }
    catch (ResourceAlreadyMarkedException e)
    {
        exception = e;
    }

    // Assert
    Assert.NotNull(exception);
}
```

Appendix D

The following code snippet has been taken from **MarkingServiceClientTests.cs** class, which can be found in the path: **../mars-deletion-service/IntegrationTests/MarkingService**.

```
[Fact]
public async void CreateMarkSession_NotMarkedResources_ReturnsMarkSessionModel()
{
    // Arrange
    var resourceType = ResourceTypeEnum.Metadata;
    var resourceId = "45db3205-83be-42a1-af14-6a03df9d9536";
    var projectId = "73fcb3bf-bc8b-4c8b-801f-8a90d92bf9c2";
    var markSessionType = MarkingServiceClient.MarkSessionTypeToBeDeleted;
    var httpService = new HttpService(new HttpClient());
    var markingServiceClient = new MarkingServiceClient(httpService);

    // Act
    var result = await markingServiceClient.CreateMarkSession(
        resourceType,
        resourceId,
        projectId,
        markSessionType
    );

    // Assert
    // Verify that the mark session is created
    Assert.NotNull(result);

    var metadata = await ResourceTypeHelper.RetrieveMetadata(resourceId);

    // Verify that the metadata is marked
    Assert.Equal(MetadataModel.ToBeDeletedState, metadata.State);
}
```

Declaration

I declare within the meaning of section 25(4) of the Ex-animation and Study Regulations of the International De-gree Course Information Engineeting that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, July 23rd 2018

City, Date

Signature: Petar Krastev