Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Prannoy Mulmi

Design and Implementation of an Archive Microservice solution for 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

Prannoy Mulmi

Design and Implementation of an Archive Microservice solution for the Multi-Agent Research and Simulation Distributed System

Bachelor Thesis based on the 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 Hamburg University of Applied Sciences

Supervising examiner : Prof. Dr. Henning Dierks
Second Examiner : Prof. Dr. Thomas Clemen

Day of delivery 21. Juli 2018

Prannoy Mulmi

Title of the Bachelor Thesis

Design and Implementation of an Archive Microservice solution for the Multi-Agent Research and Simulation Distributed System

Keywords

Distributed System, Microservice, Two-Phase commit protocol, Archive, Decentralized data, Multi-Agent Research and Simulation (MARS)

Abstract

This thesis introduces the design and implementation of an Archive Microservice in the Multi-Agent Research and Simulation (MARS) framework. Due to the distributed architecture of the system, the process of the archive and restore is complex to implement and maintain as there multiple possibilities of failure. This thesis uses different strategies to tackle the issues present in the system and provide an Archive service.

Prannoy Mulmi

Titel der Arbeit

Konzeption und Implementierung einer Archiv Microservice Lösung für die Multi-Agent Research and Simulation Verteiltes System

Stichworte

Verteilte Systeme, Microservice, Zwei-Phasen-Commit-Protokolle, Archiv, Dezentrale Daten, Multi-Agent Research and Simulation (MARS)

Kurzzusammenfassung

Diese Arbeit zeigt den Entwurf und die Implementierung eines Archiv Microservice im Multi-Agent Research and Simulation Research und Simulation (MARS). Aufgrund der verteilten Architektur des Systems ist der Prozess des Archivierens und Wiederherstellen komplex zu implementieren und zu pflegen, da es mehrere Möglichkeiten eines Ausfalls gibt. Diese Thesis verwendet verschiedene Strategien, um die im System vorhandenen Probleme anzugehen und einen Archivierungsdienst bereitzustellen.

Acknowledgment

I would like to express gratitude towards my supervising mentors Prof. Dr. Henning Dierks and Prof. Dr. Thomas Clemen for their expertise, guidance and encouragement which played an important role in the success of this thesis.

Also, I thank Julius Weyl and Lennart Karsten from the MARS team who were always ready to help out during difficult situations by providing constructive ideas. Also, I am grateful to the MARS team for being very cooperative and giving me opportunities to get a hands on experience with this state-of-the-art technologies. During this period I learned a lot about these technologies therefore gaining valuable experience for the future.

Lastly, I would like to thank my family without their support I would not have came to Germany and pursued my higher education.

Contents

List of Tables

List of Figures

Abbreviations

API	Application Programming Interface
HTTP	Hyper Text Transfer Protocol
UI	User Interface
MARS	Multi-Agent Research and Simulation
NAS	Network Attached Storage
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
CI	Continuous Integration
OOD	Object Oriented Design
AOP	Aspect Oriented Programming
MAMS	Multi Agent Management System
JSON	JavaScript Object Notation

1 Introduction

Nowadays consumers and businesses are increasingly aware of the value of archives, as the archived data has also proven to be valuable asset in the field of scientific research where an expert could archive their data for future reference. Having said that, the MARS[1] framework offers an easy interactive platform for a domain expert i.e. ecologist, biologist, chemist to simulate complex real life scenarios (e.g. reproduction of bacteria in a body) based on the concept of Agent-based modeling and simulation (MAMS[2]) [2].

1.1 Motivation

The inspiration for this work comes from the fact that an archive adds excellent value to the existing system by allowing the experts to store their results and projects and use them in future. Additionally, the MARS system requires a large number of computational resources and an Archive service would provide an opportunity for the MARS system to improve their performance because this service would move some part of data from the primary storage (Ceph cluster [34]) to the secondary storage (NAS synology [1]).

However, the underlying structure of the MARS system is also quite challenging to archive a given simulation with its resources. The system design is based on a Microservice architecture [25] having a decentralized data [28] access structure. The MARS Resources have a hierarchal structure (Section 2.1.1) which has to be followed so a successful simulation can occur. As per the design of MARS, these resources are stored separately in different storages/databases which should be accessed only by a certain API. Section 1.3 describes those difficulties.

[1]MARS: Multi Agent Research and Simulation
[2]MAMS: Multi Agent Management System

1.2 Goals

The primary goal of this thesis is to design and implement an Archive service which would deal with the decentralized structure of the MARS framework and provide an easy interface for users to archive and restore the projects and simulations back into the system.

At the time of writing this thesis, the project resources are stored in a Ceph distributed file system [34]. It provides more efficiency, reliability, and scalability by separating data and metadata using a pseudo-random data distribution function [34, p. 307] to store data in a distributed system [33]. It is financially more expensive to possess such a system in larger volumes although it is efficient and scalable. In contrast to its continuous data production, the primary storage volume available to the system is very limited. Additionally, a high level of correlation has been observed between the operating cost of a software system and its data volume. Therefore, the Archive service would move the requested data from the Ceph storage to the slower NAS[3] Synology [1]. This cloud storage is owned by the MARS for backup and archive purposes.

1.3 Problem Statement

MARS is a complex Distributed System which brings upon different levels of complexities for this thesis. The problems being dealt within the thesis are as follows:

1. **Data distribution:** In contrast to a monolithic application [31, p. 94] where there is only one database for the whole system, every microservice in the MARS System owns a separate database which should be accessed only by itself in order to be scaled independently [32, p. 27]. As a result, the archive service is coupled with the other services to get and post data into their respective databases creating more risk to failures.

2. **Data consistency and coherence:** By design of the MARS system it is necessary for the Archive service to have a distributed transaction. Therefore, it is extremely difficult to maintain a strong data consistency and coherence as the change of data in one database is unknown to the other service since an ACID [13, p. 290] transaction between the databases does not exist. Thus, the archiving process can unintentionally be led to a false state in case of failure during communication process.

3. **Understanding MARS resource hierarchy:** The MARS Resource Hierarchy is the order in which the assets in the system have to be created. Understanding this complex

[3]NAS: Network Attached Storage

hierarchy(Section 2.1.1) is vital since restoring or archiving data may lead to corrupted states.

4. **Additional changes to MARS services:** Also, for the Archive service to operate as intended, additional functionalities have to be added into the other services. The Archive service is only allowed to communicate via an API gate i.e. HTTP or RPC as accessing another service's database directly is considered an Anti-pattern [32, p. 27]. It imposes another challenge in understanding the structure and algorithms of services which are drafted in various technologies.

1.4 Thesis Overview

Here is a brief description of the thesis, providing a short overview on what each chapter contains.

Chapter 2: "Background:" this chapter describes the design of the MARS system, the hierarchal structure of the services and the technologies used to run it.

Chapter 3: "Requirement Analysis:" this chapter describes the functional and non-functional objectives of the Archive service.

Chapter 4: "Planning and Software Design:" this chapter explains, the methodologies for data storage, the archive process, and the software design.

Chapter 5: "Implementation:" this chapter is where the details of the implementation decision are explained.

Chapter 6: "Testing:" this chapter explains in detail some test cases and result validation that were carried out to give more credibility to the designed system.

Chapter 7: "Conclusion:" this chapter presents the outcomes and some suggestions for improvements which can be applied in the future.

Appendix, provides a brief description of how the archive service can be built and deployed locally for future development.

2 Background

This chapter introduces the MARS project with the intention to provide a descriptive insight of the architecture, technologies, and the structure of the system. These details play a crucial part in understanding this project. The systems architecture and technologies used to orchestrate the architecture including the relationship between the services are explained.

2.1 Multi-Agent Research and Simulation

The MARS is a simulation framework developed in HAW Hamburg as a part of a student research project. The project can be classified as a Distributed System [33] designed to carry out simulations of a given model [15]. A model describes a digital prototype of a physical agents i.e. wolves, sheep, grass which can be simulated to predict a real world scenario. A simple model would be the Wolves and Sheep; using this prototype one can simulate the interaction between the agents. As a result one can analyze the population change between them. As a second example, an Agent-Based Traffic model has been developed, which simulates realistic traffic situations in a city scale with great accuracy using the MARS framework [35].

2.1.1 MARS Resource Hierarchy

To leverage the MARS framework, specific steps have to be carried out in a chronological manner. It has to follow a specific sequence, which is the MARS Resource Hierarchy.

1. **Create a project**: A project can be defined as a collection of all the resources and the simulation results. The resources include models, scenario description, result configurations, simulation plans, simulation runs, simulation results, and different layers required by the model. Different layers are available for use of the models from a generic basic layers to additional layers. At the time of writing, the available layers to the MARS framework are as follows [12, p. 8] :

    - The Geographic Information System (GIS) Layer [11, p. 1] provides geospatial data to the agents.

- The Time series Layer enables the agents to get a data point relative to a time point (e.g. weather data of Hamburg over a day) [26].
- The Obstacle Layer provides the definition of the geographical/spatial boundaries (e.g. fish agents having a watershed boundaries).
- The Potential Field Layer provides the agents to find and follow a defined potential. This can be used to depict an agents spatial boundaries.

2. **Upload models and its corresponding layers**: The model upload is the first step required for a simulation to take place. The model contains information of the behaviour of the agents for a simulation run. The input files containing initialization data of the layers i.e. GIS, Time series, Obstacle, Potential field are also uploaded in this step.
3. **Create a scenario**: A scenario of a project can be described as the initialization of the model. In the process of creating a scenario, attributes like number of agents i.e. wolves, sheep are specified. The initialization data files like the GIS, Time series etc., are assigned to the scenario when required. The global parameters such as start date and end date of a simulation is also specified.
4. **Configure result configuration**: The result configuration represents the settings of a desired simulation result. In this step the desired parameters i.e. agent properties are selected. As a result, only the enabled properties are stored in the database which could be used for further analysis.
5. **Create simulation plan and run**: The simulation plan is a complete description of the simulation which includes, scenario and result configuration. For the execution of a simulation one must run the simulation plan, which creates a simulation run. A simulation run contains all the metadata i.e. simulation id, simulation job status. Using the simulation run, one can analyze the simulation results.

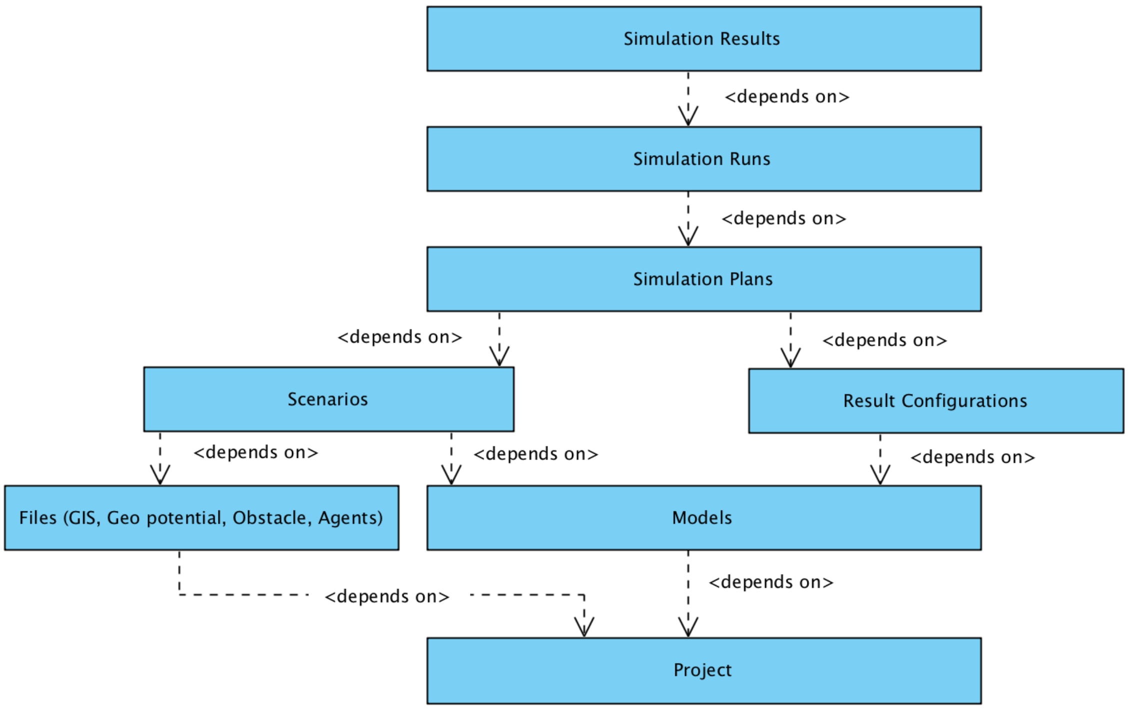


Figure 2.1: MARS Resource UML dependency graph

Figure 2.1 shows the dependencies between MARS resources. It can be observed that the order of existence of the resources has to be from the **project** to the **simulation** results (bottom to top) when adding a new simulation. Failure to follow this hierarchy will result in failure to successfully run a simulation.

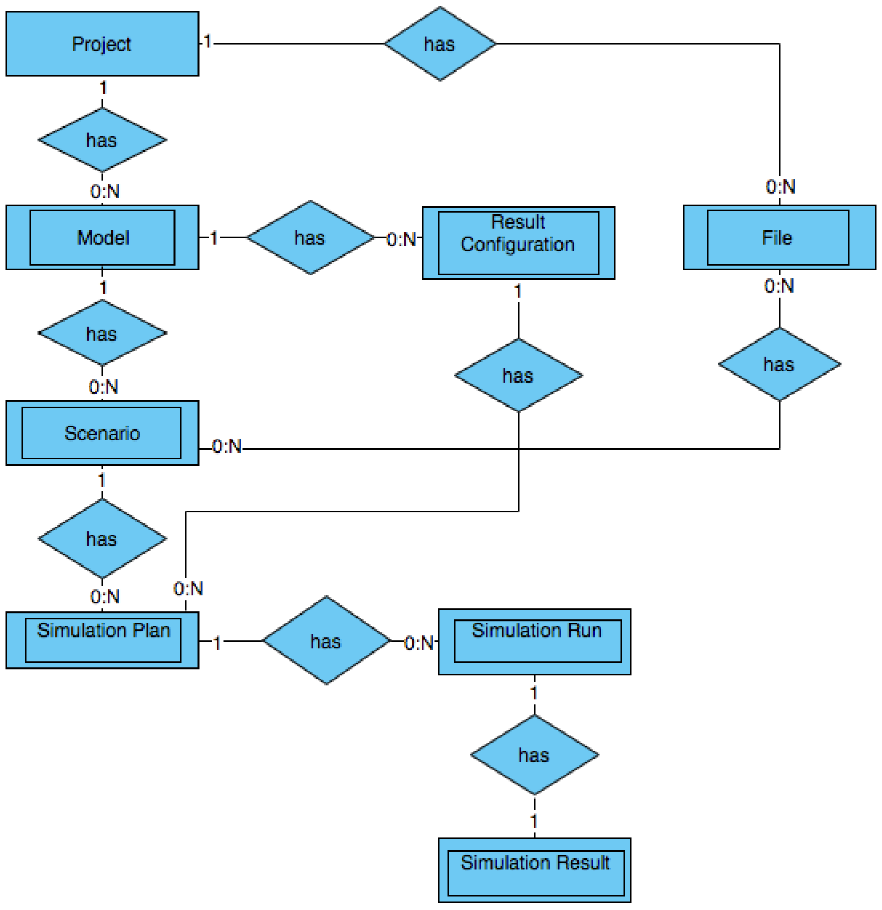


Figure 2.2: Chen notation Entity Relation Diagram for MARS resources

Figure 2.2 shows the data flow of the MARS Resources. From this figure it is obvious how the resources are dependent upon each other. This follows the hierarchal structure seen in Figure 2.1, where the project data is at the top and no other entity can exist without it. A pattern for the cardinality of the entities can be observed. The lower level entity can only have a reference to one parent entity, whereas the parent can have multiple children. An exception to this pattern is between Simulation run and Simulation results. A Simulation run will not have multiple results because it represents a job which will produce the output i.e. Simulation Results. It is also to be mentioned that every entity except the project is identified as a weak entity because they do not cease exist without its parent entity.

Furthermore, the different data flows mentioned in Figure 2.2 are handled by various services in the MARS framework. Table 2.1 gives an overview of the elementary services which are responsible for creating and running simulations. For simplicity reasons, only the services which have direct dependency with Archive service is mentioned.

Service Name	**Description**
Project Service	Handles project resources.
File Service	Handles the import and export of different file resources i.e. models, GIS[1], Time series.
Metadata Service	Manages all the metadata resources.
Result Config Service	The Result configurations handles which properties of a model will be stored in the database for a simulation.
Scenario Service	The Scenario services handles the mapping from the model constructor type to the imported files.
Sim Runner Service	Handles Simulation plans and Simulation run.
Database Utility Service	Handles all the Simulation results and is responsible to backup the project data.
Marking Service	Handles the Marking of the resources, so that when the resource is marked by one service it cannot be altered.
Deletion Service	Handles Deletion of the resources.

Table 2.1: MARS Resource Hierarchy Elementary Services Overview

2.2 Distributed Systems

The MARS cloud is architectured as a distributed system. Thus, it is of great importance to understand this architecture, so that one can anticipate the operations of the MARS framework and the technical challenges that could occur due to its complex structure.

A distributed system can be defined as a number of autonomous computing elements which appear to a user as a single coherent system [33, p. 2]. This definition implies that the system is split into meaningful domains which behave independently from each other and that the system supports resource sharing in order to appear as single coherent application to the user. For the system to appear as a single coherent system the individual units must establish some kind of collaboration with each other usually done by exchanging messages over a network. This enables the system to share different resources that are physically separated, as an element in the system can be controlled by passing messages.

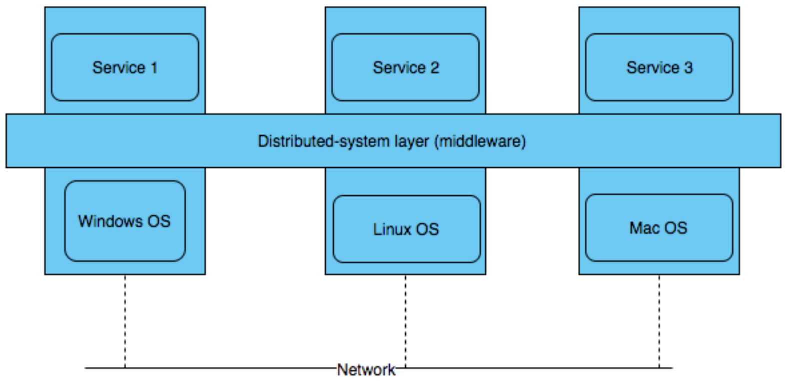


Figure 2.3: A distributed system extended over multiple devices with same application interface [33, p. 5]

Distributed Systems can be utilized to realize a complex application dispersed across multiple machines which communicates via a network protocols (e.g. HTTP [23], GRPC [9]). The components interact with each other to achieve a common goal. It also provides more reliability compared to a non-distributed system because there is no single point of failure when a system is designed properly.

Figure 2.3 shows an example of an application being distributed amongst different computers. It can also be seen that the different parts of the application are allowed to communicate via a common middleware whose main responsibility is to efficiently manage resources across the distributed applications. This kind of system makes most sense for deployment which require high performance computing power as this system can allow an application to share different resources (e.g CPU power, memory and storage) located in another machine.

2.2.1 Advantages of a Distributed System

Reliability and Availability

One of the main reasons for building a distributed system is to make the application free from single failure events. Since the application is generally spread across different nodes connected via a network, failure of a single node will not crash the system completely. This makes a distributed system more available, reliable, and independent to a user as the availability of the application is not hindered completely.

Scalability

Scalability is an important step in software development process, as the requirements for an application tends to changes by time and also requires more resources (e.g. more processing power, more data volume). In contrast to a single system, where the computer has to be replaced completely by a really high end device, there is a possibility to just expand the system but adding another device in the network. Since the application in a distributed system are able to communicate via network it is easier to scale and add more resources and also scale down if required.

2.2.2 Challenges

Data Coherency

The data is said to be coherent when it is uniform across an entire network. In other words, data is coherent when all the resources between a server and a client is synchronized. Since a distributed system is susceptible to network failures, a network partition [8, p. 59] cannot be avoided. In presence of a network partition the change in data will not be synchronized to all of its client leading to have inconsistent data. Furthermore, the complexity to maintain this coherency increases drastically when the system has more clients trying to gain access to the data. This phenomena must be taken into consideration while designing an application for this kind of system.

Network Issues

Generally in a distributed system, different applications communicate via network protocols i.e. HTTP, GRPC. It is to be noted that communication via a network is not always reliable. This is because managing a distributed network is rather complex. Also due to external reasons the communication can break leading to loss of messages which disables some parts of the application. This phenomena is also not seen in a single system.

Error Handling

Errors are imminent in every application and to continue working normally again it has to detect and recover from them. Detecting errors in a distributed system requires a different approach since the application is spread across multiple systems. It is not enough that each service ensures its own correctness because the system is interconnected via a network connection. Due to the fact that the network is also involved in a distributed system there are additional error detection methods that have to be implemented. This brings up more complications in comparison to a single system where an error is contained within a single system.

2.3 Microservices

Microservice is a specialized implementation of Service Oriented Architecture(SOA) [4, chapter 3]. A service in a SOA is a functional unit that performs a specific business action (e.g. user authentication) accessed typically via a network that encapsulates its state and the operations performed on the data. Figure 2.4 illustrates an example of Service Oriented Architecture (SOA) in a distributed system where different services call each others interface to perform a certain action. Although microservices are built using the SOA paradigm, they have their differences. In a microservice, a service can be deployed and operated independently because the services are designed to be more fine grained with a single purpose, unlike SOA. Also they are lightweight and Domain Driven [5] that makes the application simple to understand, develop, and test. The smaller set of services can be developed autonomously by different teams and be deployed quickly as they are usually lightweight in nature. This architecture promises to bring loose coupling by separating a big application into smaller logical units.

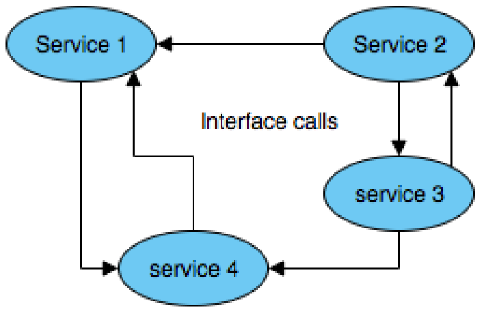


Figure 2.4: An service oriented architecture in a distributed system [33, p. 62]

Figure 2.5 shows an example of how a service is domain bounded and there is a flexibility of choosing different technologies for the isolated services [25]. Each service is encapsulated with their own life cycles, which communicates with each other using protocols (e.g. HTTP [23], websockets [24], GRPC [9]).

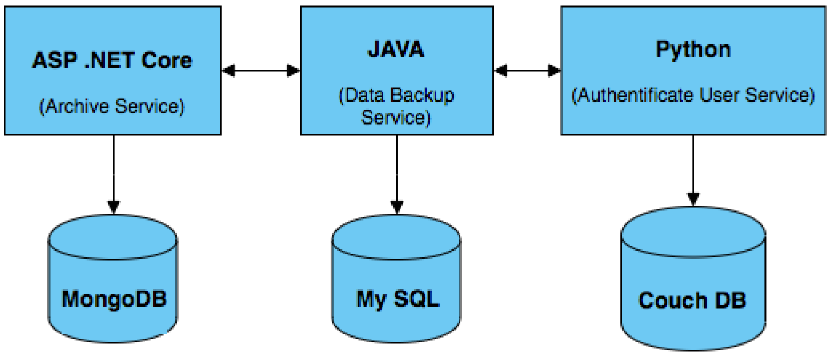


Figure 2.5: Illustration of services as fine grained independent entities

A single monolithic application [31, p. 94] is built as a large unit where all the logic lies within a single system. This is considered the most natural way to develop a server-side application. It is seen that as the application scales in size, it gets harder to keep up with the changes as an entire system scaling is required. This is where microservices can be beneficial, as only the required bounded module can be scaled up as needed. There are different factors to be considered before going for a microservice architecture as improper planning could lead to an unstable system.

Advantages	**Disadvantages**
The services can be developed with different languages	A mature team must be present to maintain large number of services
A strong modular boundaries is present which reinforces a modular structure.	All the services must manage data consistency amongst the services which is harder to manage in a large distributed system.
Independent deployment is easier since the services are autonomous.	Harder to program since remote calls must be made.

Table 2.2: Advantages and disadvantages of microservices [21]

2.3.1 Data Sovereignty in Microservice

It is an important guideline for a microservice architecture to own its domain data and logic [32, p. 29]. Decentralized data would assist a microservice to become solely independent and help them evolve separately. The approach of each microservice owning its own database is also know as **Polyglot Persistence** [22]. Applying this pattern would imply that the data belonging to one service is available to the others only via the API of the microservice.

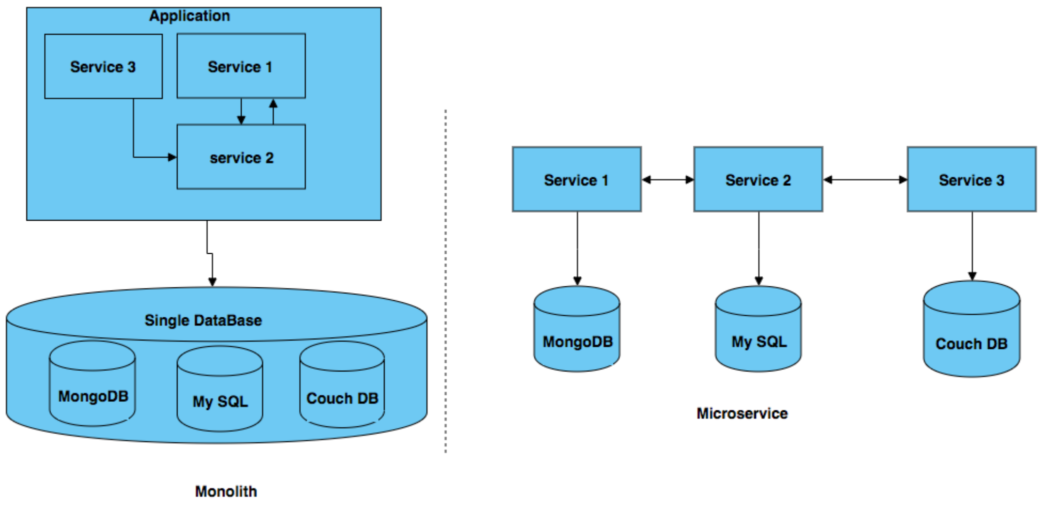


Figure 2.6: Data management approach in a Monolithic application vs microservice [21]

In Figure 2.6, it can be observed how a Monolithic application owns only a single database for the whole application. Meaning, the application has a centralized database which are shared amongst the services. Whereas, in microservices each service owns a single database or few services share databases which is easier to manage. Having said that data sovereignty is very beneficial, it also brings various difficulties i.e. coordinations between services, which is very challenging to tackle and creates data coherency issues in the system.

2.4 Archive

Archiving in computer science is an act of storing single or a collection of data with its meta-data for long-term retention. The data being archived is not needed currently in the active system. Generally these data are valuable for an organization or an individual which is not to be discarded but are seldomly needed. This brings forward the need of relocating the data into a cheaper storage i.e. archiving. Traditionally, these kinds of data were stored in magnetic tapes but nowadays due to availability of cheaper NAS[2] primary storage, these storage means are being preferred. The advantages of using NAS i.e. Synology [1] are listed below:

1. **Easy File Sharing** It is easier for many users to access the archived data because the storage is connected to the cloud. Whoever has access to the network can get the data compared to magnetic tapes where one needs to physically posses the tape in order to get the required data.
2. **Easy Usability** The system is easy to manage since it provides an easy installment procedure, and also a graphical interface for file access.

Often an archive can be confused with a backup of a system. The key differences for archive and backup are mentioned in Table 2.3.

Archive	**Backup**
It is unused but a desired copy of the data useful for future use.	It is a copy of the current active data store used to recover from data corruption.
The data is relocated from the current storage system onto a less expensive storage.	The data is just a copy of the working copy and may or may not be stored in the same storage as the active system.
The duration for keeping an archive is longer since it would in most cases not change frequently.	The duration of the backup would be less compared to archiving since it would be updated frequently (e.g. daily, weekly, monthly) to have the newest working copy.

Table 2.3: Differences between archive and backup

[2]NAS: Network Attachted Storage

3 Requirement Analysis

The main requirement of this thesis is to design, and implement an Archive service i.e. back-end web service for the MARS framework. The service's role is to archive the MARS resources mentioned in Subsection 2.1.1 from the Ceph cluster [34] to the Network Attached Storage (NAS) Synology drive [1].

This service targets any user who desires to archive the MARS resources needed for a simulation including the existing simulation results, which would be analyzed in the future. The Archive service would expose its API, calling it, one can archive and restore the resources. The exposed API is also integrated in the MARS graphical interface (MARS Teaching User Interface). The MARS Teaching API acts as proxy between the users and the MARS back-end Microservices, as it provides some level of abstraction by hiding unnecessary endpoints for the user in the graphical interface.

3.1 Functional Requirements

This section describes the functional requirements for the Archive service. The functional aspects which carves the Archive service are mentioned below.

3.1.1 Archive Project Resources

The designed system must be able to archive the MARS resources from the active system (Ceph cluster at the time of writing) into the Synology [1]. The application must be able to archive a project which does not have all the resources mentioned in Table 3.1 (e.g. no simulation runs have been triggered). This must be supported since it could be the case that the user wants to archive only some of the resources.

Resource Name	**Description**
Metadata	This resource stores the metadata (e.g. file id, file name). It gives the system the information about existing files in the system.
Files	This resource correspond to the models (e.g. wolves and sheep model) and input files (e.g. GIS, Time series) which describe a simulation.
Scenarios	This resource defines the parameters for the model which would be simulated (e.g. simulation run time, number of agents).
Result Configurations	This resource defines which parameters of a model and layers are going to be stored in the database that will be used for visualization and result analysis.
Simulation Plans	This resource contains the scenario and the result configuration which can be executed. The simulation plan could be configured to have different scenario and result configuration to produce different kind of output.
Simulation Runs	This resource contains the metadata for a simulation results i.e. simulation id, simulation status.
Simulation Results	This resource is the output and contains the results for a single simulation run.

Table 3.1: MARS resources which are to be archived

Assurance of correct data being persisted

MARS being a distributed system, data coherency (Subsection 2.2.2) is one of the big issue which this thesis faces. As a consequence, wrong or unwanted data could be archived. Therefore, the Archive service must ensure that while an archive is running the data is not altered.

3.1.2 Retrieve Project Resources

The designed software must support the retrieval of the archived projects from the Synology into the active system. The system must be able to restore the project given that, the services support the data format which is archived in the Synology.

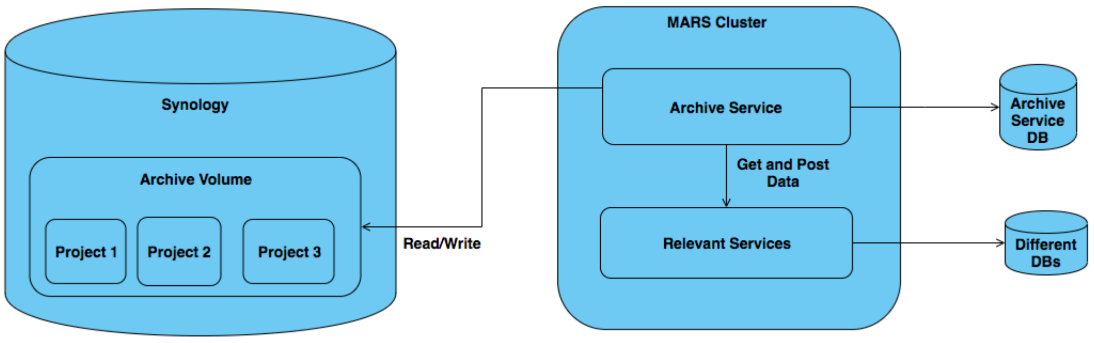


Figure 3.1: Archive service's communication structure

Figure 3.1 illustrates that the Archive service must only use the volume assigned for archiving and nothing more. This requirement must be fulfilled to comply with the MARS development standard. It also has to be made sure that the retrieved resources are usable (e.g. the restored simulation plans should be able to run a simulation again).

3.1.3 Archive and Retrieve Process Status

The archive and retrieve processes are long running tasks. The designed software must run these processes in the background to avoid long waiting time for another requests. Given this, an API endpoint must be made available which gives the current status of the archive or retrieve job. Using the status a user can determine whether the job is being executed or is finished.

3.1.4 Download Archived Data as a Compressed File

It is of great importance for a domain Expert i.e. ecologist who are not technical experts to have a graphical interface. In this interface, it must be possible to navigate to the project of interest and easily download the project as a zip file. There could be cases where the MARS system is out of order and the data is required. Then it can be accessible by anyone with basic knowledge of the system.

3.1.5 Fault-Tolerant Design

Firstly, the Archive service has to communicate with many services in the system, leading the rate of failure being higher in comparison to a system which does not depend on other services. A breakdown of one service would cause the whole process of archive/retrieve to stop unexpectedly. Secondly, it is also possible that a running Archive service can be terminated due to some unexpected reason. Therefore, fault tolerance mechanism has to be included in the archive system, so that it has a chance of recovery.

3.2 Non Functional Requirements

The requirements specified in this section present us the technical/non-functional aspects of the Archive service. A tabular description (Table 3.2) is presented below. The detailed description depicts the benchmarks of how the system should be designed to meet the needs for a better sustainable prototype. The result from this work delivered must comply with the following technical requirements.

Requirement	**Description**
Build and deployment	The service should be deployable in the MARS kubernetus [10] cluster using the Gitlab[1] pipeline as seen in Figure 6.1 which is valid at the time of writing this Thesis. In addition, the build stages for the pipeline also has to be written.
Extensibility	The system must be made extensible so that future requirements can be easily added.
Robustness	The system must be able to cope with different kinds of errors during execution.
Logging	The service must provide logging information.
Usability	The system should be integrated in the MARS UI so that it is easily usable by all end users.
Make a Swagger API interface	The Archive service should have a Swagger [30] interface available so that other developers can use the service with ease.
Follow Microservice patterns	The service should follow data sovereignty pattern for Microservices mentioned in (Subsection 2.3.1).
Responsiveness	The API should give some kind of feedback to the user never the less if request cannot be made an error message should be returned instead of no result.

Table 3.2: Technical requirements for the Archive service

4 Planning and Software Design

The chapter discusses the decisions, design patterns and, architectures utilized for this thesis, following the requirements mentioned in Chapter 3.

4.1 API Design

The Archive service must expose its web API, so that an interaction between its clients and the application can occur. The Representational State Transfer (REST) [6, Chapter. 5] architectural approach is chosen to design its API. This approach is a common approach to build a distributed system as it is technology independent. Therefore using this architecture, the system can later support any kind of system, providing a broader layer of flexibility to the Archive service. Also, the standardized aspect of a RESTful service enables a software to create reusable elements [3]. A combination of HTTP with REST to do a CRUD (Create, Read, Update, Delete) operation is preferred because HTTP is widely supported by most clients and programming languages (e.g. web browsers). The CRUD over HTTP consists of few uniform noun based interactions that can be executed by the client [3, p. 13]. The Table 4.2 describes the API endpoint for archive, retrieve, and job status with a brief description. HTTP CRUD operations which are going to be implemented are described in Table 4.1.

HTTP Verb	**Description**	**Application**
POST	Creates a new resources and dependent resources.	The POST request will be used to archive and retrieve the projects because new resources are being created for these requests.
GET	Reads the resource.	The GET request will be used to check the status of the archive and retrieve process.
PUT	Updates the resource.	The PUT request will be used to update the status of the archive and retrieve process.
DELETE	Deletes the resource.	The DELETE request will be used to delete a running archive or retrieve process.

Table 4.1: CRUD interaction over HTTP in Archive service

API Endpoint	**Description**
archive/archiveProject/projectId	Archives a project given an id. This is a HTTP POST method.
retrieve/retrieveProject/projectId	Restores a project given an id. This is a HTTP POST method.
job/status/projectId	Gets the status of the archive or retrieve process, given an project id. This is a HTTP GET method.
job/status/jobId	Gets the status of the archive or retrieve process, given a job id. This is a HTTP GET method.
delete/project/projectId	Deletes the archived project from the synology drive, given a job id. This is a HTTP DELETE method.

Table 4.2: API Endpoints description for Archive service

The API end points are designed considering the fact that more functionality could be added to the archive service without big changes needed in the client. For example, the endpoint **"archive/archiveProject/projectId"** is designed thinking an archive could be also extended for other resources besides this project. When the Archive service would like to support, archiving of only the simulation results, the endpoint for it would be **"archive/archiveSimulationResult/SimulationId"**. Hence, it would make it more flexible for the client to add the functionality without much effort. Also, to avoid multiple API calls and increase the performance of the server, the get status (Table 4.2) endpoint combines vital information needed for the client in one request (See Listing 4.1).

```
{
    "status": "PROCESSING",
    "projectId": "70C961b7-89bf-4bd5-bf61-31b6a17a15d9",
    "error": "NO ERROR",
    "lastUpdate": "2018-06-17T10:05:50.216Z",
    "archiveName": "NONE",
    "markSessionId": "AK5961b7-89bf-4bd5-bf61-31b67a15d88",
    "jobId": "855961b7-89bf-4bd5-bf61-31b6a17a15d3",
    "currentProcess": "Archive"
}
```

Listing 4.1: Sucessful GET request for a archive status

4.2 Archive Process Design

4.2.1 Preconditions Required for an Archive

Certain preconditions have to be met before an archive process could start to ensure the correctness of the data which are mentioned below:

1. **Mark resources:** The MARS framework is a multi user application meaning a project can be accessed by many users at the same time. As multiple users can modify the data simultaneously, it could be possible that someone changes a resource during an archive and the Archive service has no way to detect this modification. This would lead to inconsistent data being archived. Therefore, to avoid this situation the resources must be marked before the start of an archive. The marking would ensure that no other process except the Archive service would have access to modify the marked contents during the archive process. The marking process would be handled using the Marking service.

2. **Get Metadata for the project:** The metadata contains all necessary information about the different resources. The scenarios, files and the result configuration depends on these metadata to retrieve their respective data. If the metadata cannot be obtained the archive process cannot continue.

3. **Get Simulation runs:** The simulation run contains the simulation id which is required to archive the correct simulation. This dependency with simulation run makes it necessary that they are obtained before archiving the simulation results.

4.2.2 Decision for Not Archiving the Project

The resources depicted in table (Table 3.1) must be archived following the MARS resource hierarchy (Figure 2.1). Following the hierarchy, it is arguable why the project is not being archived, despite it being present. The project lies on the top of the hierarchy meaning no other resources are usable without its existence. Section 3.1.2 mentions the requirement that the Archive service must restore all the archived resources back into the system. Therefore, during a restore, if a specific archived project is not available, then other resources cannot be brought back into the system as all of them are dependent to the project. To make the restore process possible, the project data is not archived which now acts as a point of reference to bring back the children resources. If the project was to be archived then additionally a mechanism is needed that ensures the users referenced in the archive also exist in the Mars system (active system). It could be possible that during a retrieve the users in an archived project may not exist anymore as they were removed causing the process to fail. Therefore, this decision also reduces the complexity of the archive and restore as this mechanism can be avoided.

4.2.3 Data Format

A suitable file format for archiving must be chosen because it determines how the data access will be realized and whether it meets the criteria mentioned in Section 3.1. The different types of data archived are the metadatas for files, scenarios, result configurations, simulation plans, simulation run including the input files, models, and the simulation results. The data format that is being discussed in this section mainly focuses on the metadata and they are received originally as a JSON document. The metadata is very important because they give the system vital information that will be used during retrieve process.

HDF5

HDF5 is a file format for storing and managing data which has support for various data types designed for efficient I/O, compression, portability and, supports big data [7]. It has been successfully applied to different scientific projects which involve simulation (Efficient for simulation data [29, p. 11]). This file can also be defined as an abstract data container which includes building blocks for data organization. This file system can hold a variety of heterogeneous objects like images, graphs, documents, tables etc. It also has support for a n-dimensional table [7, p. 2]. The HDF5 format has two primary objects which define the data storage structure:

1. **Groups:** They are responsible to organize the data objects in the HDF5 file format. A Group can be compared to a directory in Windows or Unix system [7]. Figure 4.1 shows an example of a Group (e.g. project1) in a HDF5 file. Using the API of the HDF5 libraries a Dataset (e.g. scenario Metadata) can be accessed via the path name (e.g. /Root/project1/scenario Metadata).

2. **Datasets:** A Dataset can be defined as a multidimensional array of data. This object contains the raw/actual data (e.g. simulation results). These data are stored in a n-dimensional array format, where one can specify the different data types for the raw data i.e. integer, float, character, variable length strings [7]. Figure 4.2 shows an example of a Dataset in a HDF5 file which is stored in an array.

Figure 4.1 also shows how the Groups and Datasets can be used to archive the MARS projects. Every new project to be archived will be added as a new Group as depicted in the figure and the datasets are the resources i.e. scenario, file, simulation results.

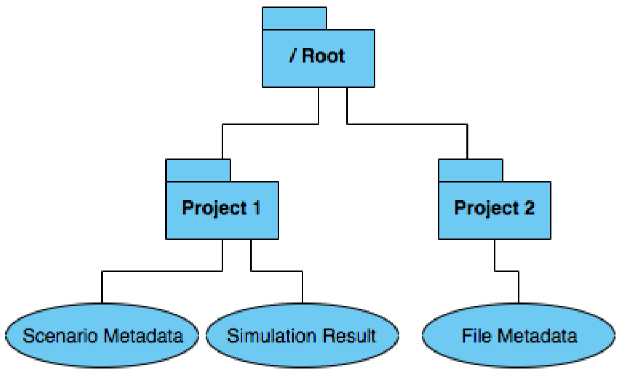
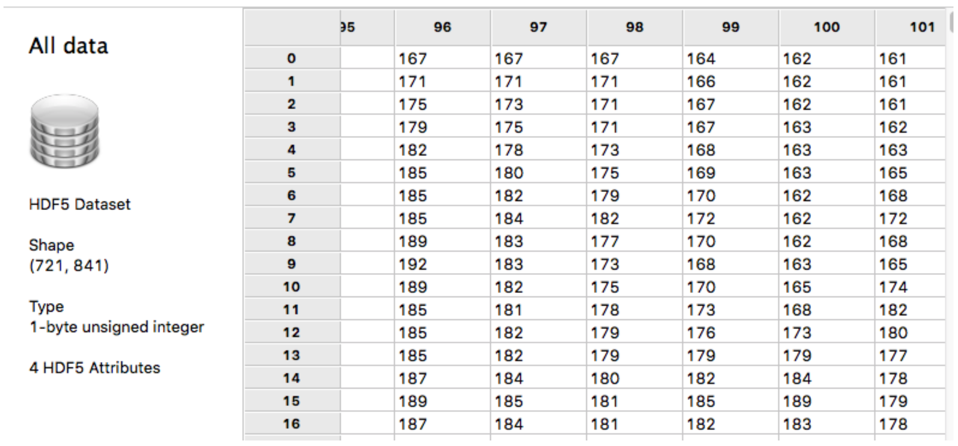


Figure 4.1: HDF5 Groups and Datasets [7]



All data

HDF5 Dataset

Shape
(721, 841)

Type
1-byte unsigned integer

4 HDF5 Attributes

	95	96	97	98	99	100	101
0		167	167	167	164	162	161
1		171	171	171	166	162	161
2		175	173	171	167	162	161
3		179	175	171	167	163	162
4		182	178	173	168	163	163
5		185	180	175	169	163	165
6		185	182	179	170	162	168
7		185	184	182	172	162	172
8		189	183	177	170	162	168
9		192	183	173	168	163	165
10		189	182	175	170	165	174
11		185	181	178	173	168	182
12		185	182	179	176	173	180
13		185	182	179	179	179	177
14		187	184	180	182	184	178
15		189	185	181	185	189	179
16		187	184	181	182	183	178

Figure 4.2: Example of a HDF5 dataset

JSON

JSON(JavaScript Object Notation) is a data format which is very easy for humans and machines to read and write. This format is completely language independent which follows the conventions used in different programming languages i.e. C#, Java, Python and more. The REST API implemented in the MARS framework supports this format with ease making this a very suitable candidate. This format is widely accepted and even MongoDB supports it without any problem. Also, MongoDB seems to be a good candidate but the requirement (Section 3.1.4) states that the archived data should be easily accessible to a non expert and using MongoDB requires some amount of technical expertise.

Conclusion

To store the data in HDF5 all the attributes of the metadata in JSON must be parsed in a n-dimensional array structure which can be understood by the HDF5. It is important to understand that MARS supports different kind of models which makes it impossible to predict the structure of a resource. To elaborate, the Wolves and Sheep [15] model has a different metadata structure than the KNP [15] model as the Agents and Layers involved are different. Due to this reason, the resources must be parsed every time into a n-dimensional array by writing each field. Also, at the time of restore this has to be converted back to a JSON as the MARS system does not understand the HDF5 structure. In addition, to avoid parsing the data, a one dimensional array of size 1 was created (type variable string) and all the unparsed JSON data was stored in this array. The main intention of this experiment was to see how the HDF5 files would react as it is easy to get the string from a 1d array (e.g. exprimentArray[0]). This also did not bring any positive result, instead it created so much overhead to the file as a JSON file with the size of 2 KB had a file size of around 0.5MB for the HDF5 variant. This made the HDF5 file system very inconvenient to use. Despite the HDF5 file providing different benefits i.e. fast I/O, portability, support for big data and, compression on a dataset it does not seem suitable in the MARS system due to the amount of complexity that needs to be dealt with for the archive and restore process.

Considering the above mentioned factors the file format for the metadata is planned to be JSON instead of the HDF5 file. Different advantages such as easy handling, no extra parsing required for the MARS system and, easy conversion to different types of file format i.e. CSV used often in the MARS for analysis, makes it more suitable for this purpose. To aid the performance, the meta files for each kind of resources are going to be stored separately (e.g. scenario.json, resultConfiguration.json). Splitting the JSON files like this would aid in faster serialization and deserialization as only the required resource can be loaded into the memory. As the input files and the simulation results do not need to be read they are just chosen to be zipped.

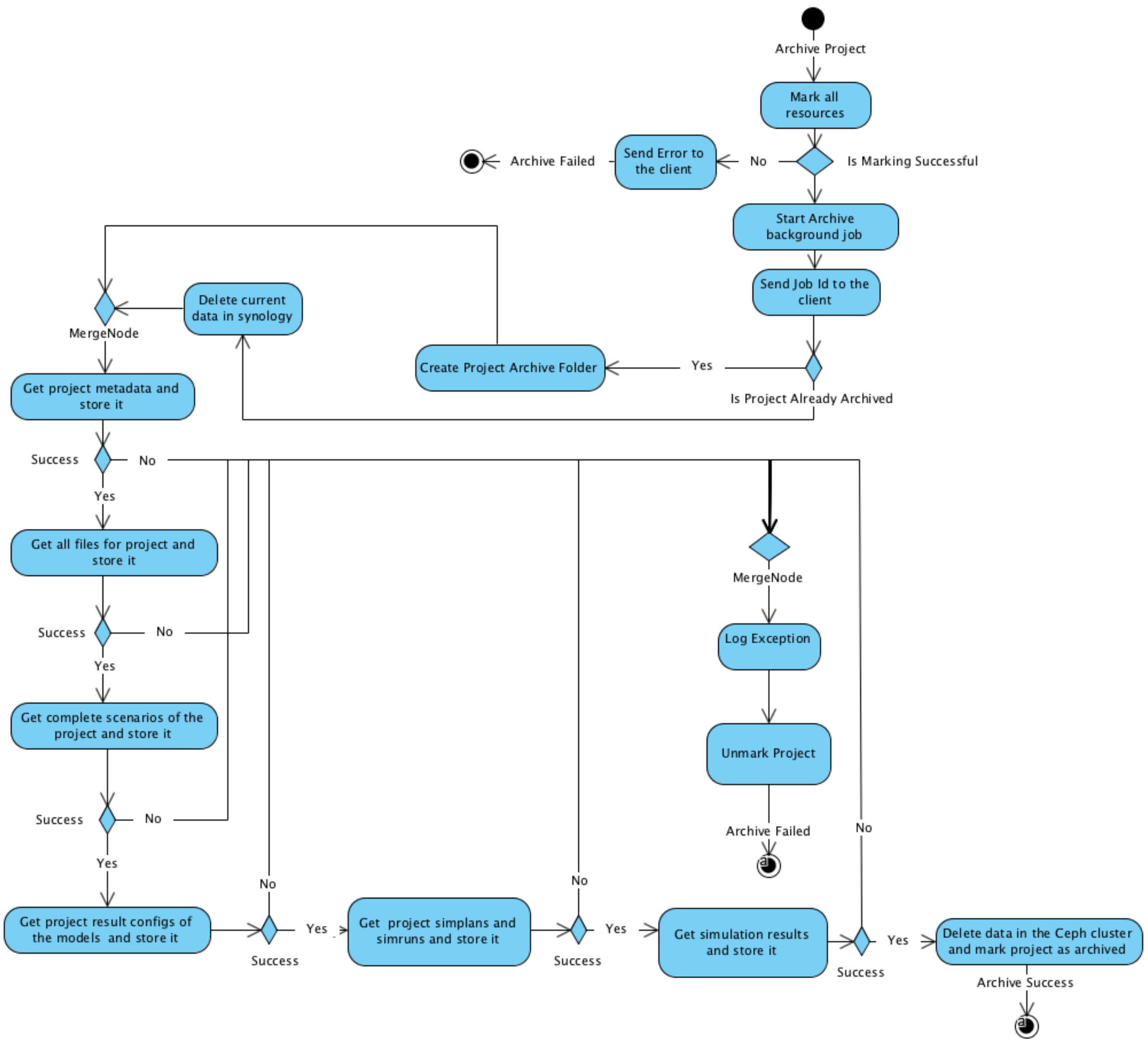

Archive Project
Mark all resources
Is Marking Successful
No
Send Error to the client
Archive Failed
Start Archive background job
Send Job Id to the client
Is Project Already Archived
Yes
Create Project Archive Folder
Delete current data in synology
MergeNode
Get project metadata and store it
Success
No
Yes
Get all files for project and store it
Get complete scenarios of the project and store it
Get project result configs of the models and store it
Get project simplans and simruns and store it
Get simulation results and store it
Delete data in the Ceph cluster and mark project as archived
Archive Success
Log Exception
Unmark Project
Archive Failed

Figure 4.3: Activity Diagram of MARS project Archive process

Figure 4.3 illustrates an activity diagram of archiving an entire project in the MARS system. As mentioned in Section 4.2.1, marking the resources for the project is the first requirement needed to start the process. If the project resources are marked successfully then the archive would be initialized as a background job and the job id would be sent to the client. In case the marking would fail, the error message will be sent to the client and the archive process will halt. The archiving process is visioned to be a background job due to the fact that a single process could take a long period of time and would block the server for additional requests. Following the successful job creation, the process checks if an archive already exists. After the archive folder creation, the metadata, files, scenarios, result configurations, simulation plans, simulation runs, and the simulation results are retrieved respectively. After a successful archive process the resources are deleted from the Ceph cluster. Lastly, in case of some failures during archiving the exception will be logged which can be later analyzed for maintenance.

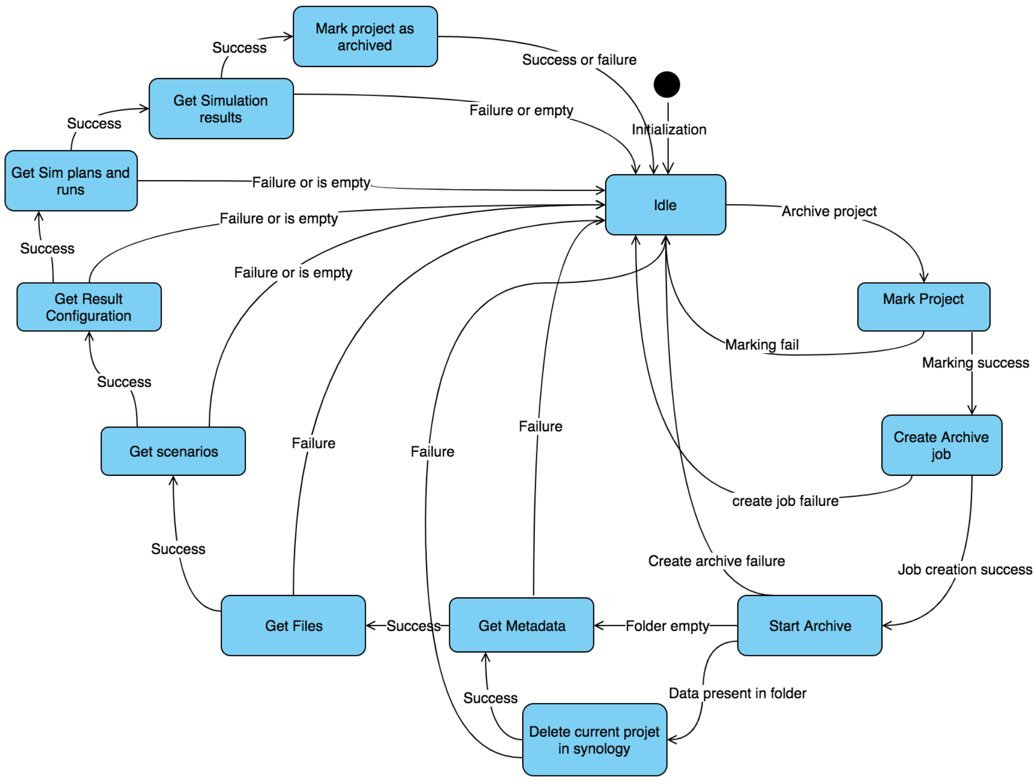


Figure 4.4: State Diagram of MARS project Archive process considering empty states

Figure 4.4 illustrates the transitions that can occur during the archive process. The idle state is when no archive process is being executed. Additionally, the state digram also considers how the state would change if one of the resources is empty. In the case of an empty resource (e.g. no scenarios available for the project) the archive process stops gracefully by logging the error and transitions to the idle state.

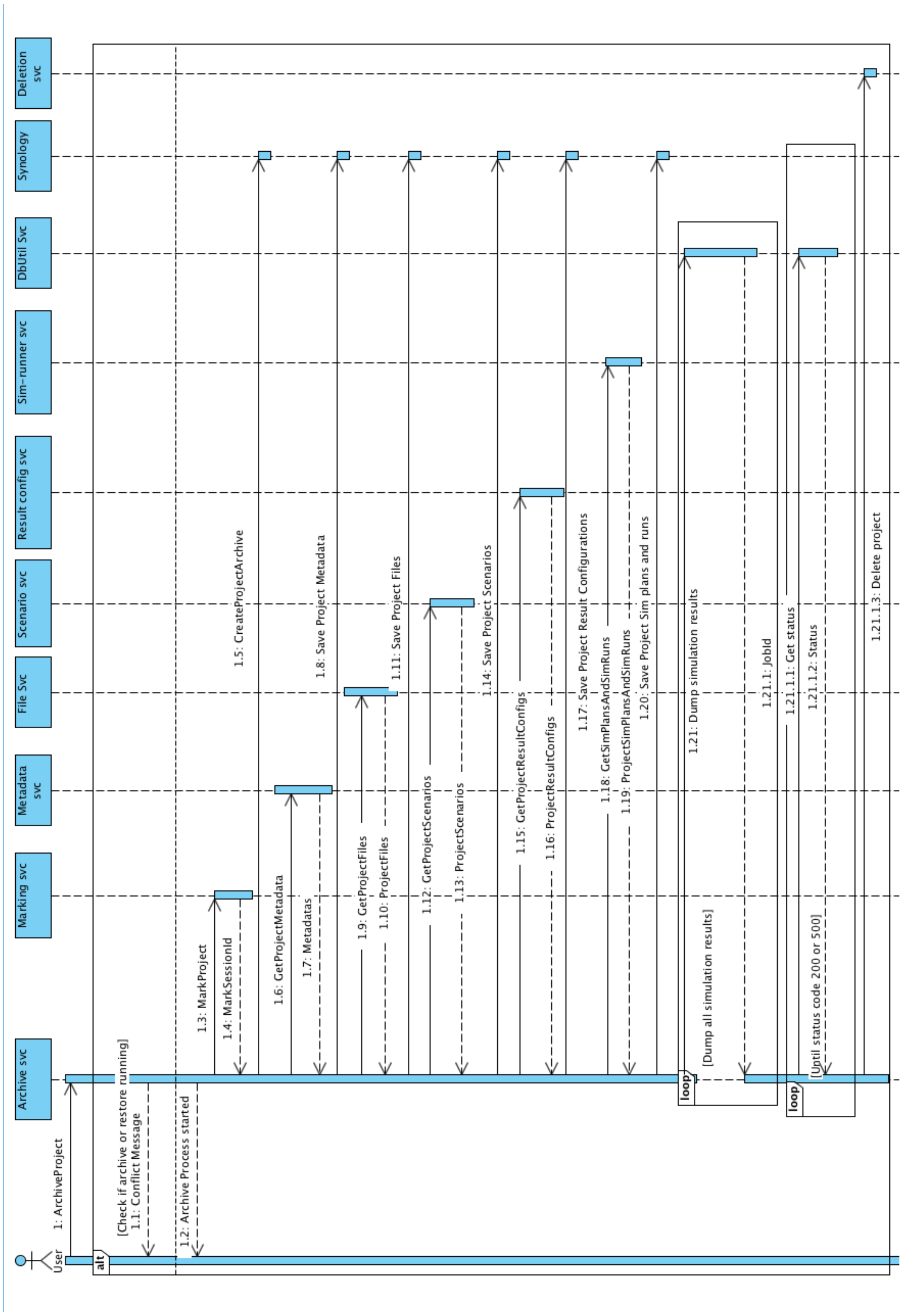

User
Archive svc
Marking svc
Metadata svc
File Svc
Scenario svc
Result config svc
Sim-runner svc
DbUtil Svc
Synology
Deletion svc
alt
1: ArchiveProject
[Check if archive or restore running]
1.1: Conflict Message
1.2: Archive Process started
1.3: MarkProject
1.4: MarkSessionId
1.5: CreateProjectArchive
1.6: GetProjectMetadata
1.7: Metadatas
1.8: Save Project Metadata
1.9: GetProjectFiles
1.10: ProjectFiles
1.11: Save Project Files
1.12: GetProjectScenarios
1.13: ProjectScenarios
1.14: Save Project Scenarios
1.15: GetProjectResultConfigs
1.16: ProjectResultConfigs
1.17: Save Project Result Configurations
1.18: GetSimPlansAndSimRuns
1.19: ProjectSimPlansAndSimRuns
1.20: Save Project Sim plans and runs
loop
[Dump all simulation results]
1.21: Dump simulation results
1.21.1: JobId
loop
[Until status code 200 or 500]
1.21.1.1: Get status
1.21.1.2: Status
1.21.1.3: Delete project

Figure 4.5: Sequence Diagram for the Archive process

Figure 4.5 illustrates the Sequence diagram for a complete archive process. The first step after an archive request would be to check whether an archive or retrieve process for the current project is under progress. In case the process is in progress, the archive request would be denied to the user with a conflict message. If no process with the project is running then an archive job (a separate thread) would be created and a message to the client with start of archive process would be sent. Following the job creation, the project will be marked so that during archive no changes to the project resources could be made. If this step fails the archive process would stop by logging the error. After the marking step completes, the archive process receives a mark session id and the dependent resources which would allow the process to make changes to the resources. Using the dependent resources the process retrieves metadata, files, scenarios, result configurations, simulation plans and, simulation runs respectively and persists them in Synology. Lastly, the simulation result dump action will be called which will archive the result data. The process waits until all the result data is archived successfully. After a successful archive, a request to delete the project data would be made so that the system memory can be freed.

A major issue to be discussed is when the archive process fails which requires a rollback by unmarking the project. As mentioned earlier the project is marked as "TO_BE_ARCHIVED" so that no other processes can modify the contents during the archive process. This is a great strategy if everything goes as planned but often this is not the case, and it is mandatory that an unmarking of the project is done otherwise the project would be unusable.

It also happens that since the marking service is dependent upon many services, it has a very high rate of failure as well. This brings upon the problem how would the archive service behave if the unmarking of the project fails. It seems very natural to just repeat the process until the unmarking request would succeed, since it is absolutely necessary to unmark a project. If this happens only with a single process, it does not make a huge difference. However when thinking of the bigger picture if this occurs with 100 different processes at the same time, it would use valuable processing resources as it may be stuck in a deadlock condition until a outside interruption is made. To avoid this, a fixed number of retries to unmark the project with certain time interval for the request will be made.

Although, this solves the issue of using up resources but the problem that the project is unusable is still there. Except a manual unmarking, no other solution can be seen, so it is decided that the archive process would also persist the marking session id which can be used to call the unmarking endpoint. With the use of this id a manual trigger is possible as soon as the error is fixed. The marking session id can be easily retrieved also from the GUI as it will be included in the status request of the archive job.

4.3 Retrieve Process Design

This section describes the design and behaviour for restoring the archived project from the Synology back to the system so that it can be used again using the MARS UI.

4.3.1 Decision of File Upload via File-svc

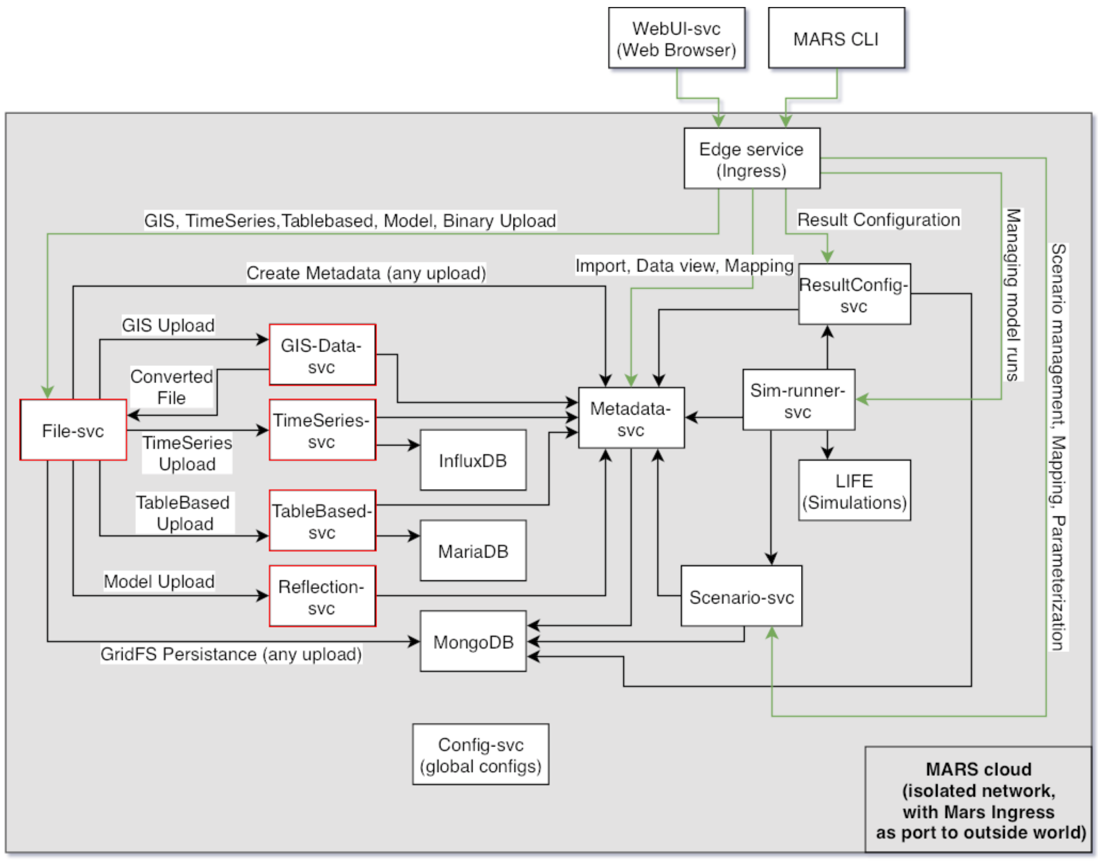


Figure 4.6: File Upload in MARS Cloud [20]

Figure 4.6 illustrates how a file upload in the MARS cloud is done at the time of writing this thesis. The MARS cloud is a complex Distributed System consists of many Microservices and databases at its disposal. It is a point of interest how the project would be restored back as there are different possibilities for it. However, it is a requirement for the Archive service to call the corresponding service to access, add, or modify the resources (Chapter 3). As mentioned in earlier chapters, the MARS system has different types of files (e.g. models, timeseries, GIS) which are managed by their own service. The files can be uploaded in two different methods.

1. **Upload files via File-svc** The File-svc is a service which accepts all kind of inputs i.e. GIS, models, timeseries. It communicates to the concerning service by checking the fle type. This is the only method which is possible in the UI.
2. **Upload the respective file via its service** This method requires the Archive service to communicate with each service of the file type. The archive service can communicate to any service provided an endpoint (interface). Therefore, it is also possible to upload the different kinds of files using the corresponding service instead of the file service.

If a file type model is to be uploaded, an API call to the reflection service has to be made. Similarly, a GIS file needs the GIS Data service.

The File-svc can be seen as an abstraction layer for uploading different types of files. This layer reduces the number of direct dependencies to the Archive service because it does not call the other services directly. Choosing the File-svc also provides an additional advantage if a new file type is added in the future. In this case, the archive service does not need to modify any code to upload the new file type. Given the reasons to have cohesion and easy maintenance, file uploads via File-svc deemed to be a better choice.

4.3.2 Retrieve as an Atomic Action

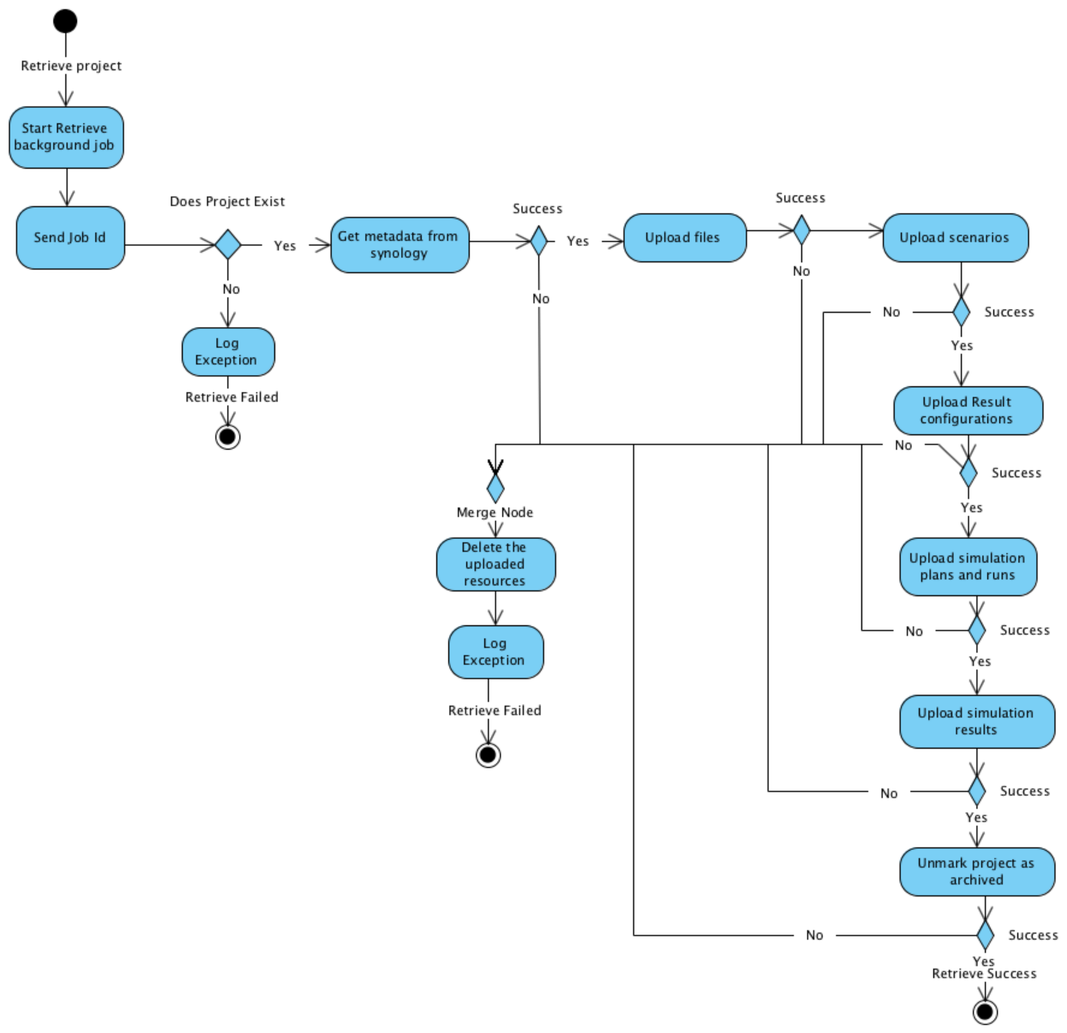


Figure 4.7: Activity Diagram for retrieving a project

Figure 4.7 depicts the activity diagram for restoring a project. The retrieving process is also a background job due to the same reason as for the archive i.e. long running times. The first step after creating the retrieve job is to get the metadata from the Synology and then upload all the files. All the files have to be finished uploading and processed, otherwise the sequential steps would not have the references of the files. After all the uploads are complete, the scenarios get the reference to the file id so that it could be uploaded. Following the scenarios, the result configurations are also uploaded for the corresponding models. As the simulation plan is dependent upon the scenario and the result configuration, this is the next resource which will be uploaded. Lastly, the simulation runs and the simulation results would be uploaded respectively.

In case an error occurs, a Two-phase commit protocol [28] is adopted. This strategy is taken into consideration to bring atomicity on decentralized data as it tries to roll back if the distributed transaction fails. Due to chances of failure, an incomplete data restore process could occur. In the MARS system, one cannot work with having incomplete data since the resources are dependent upon each other. Having an atomic transaction for the retrieve process would be a simple mechanism to overcome this issue. In case of any failure during retrieval, the partially restored resources would be deleted to make the retrieve process as an atomic action.

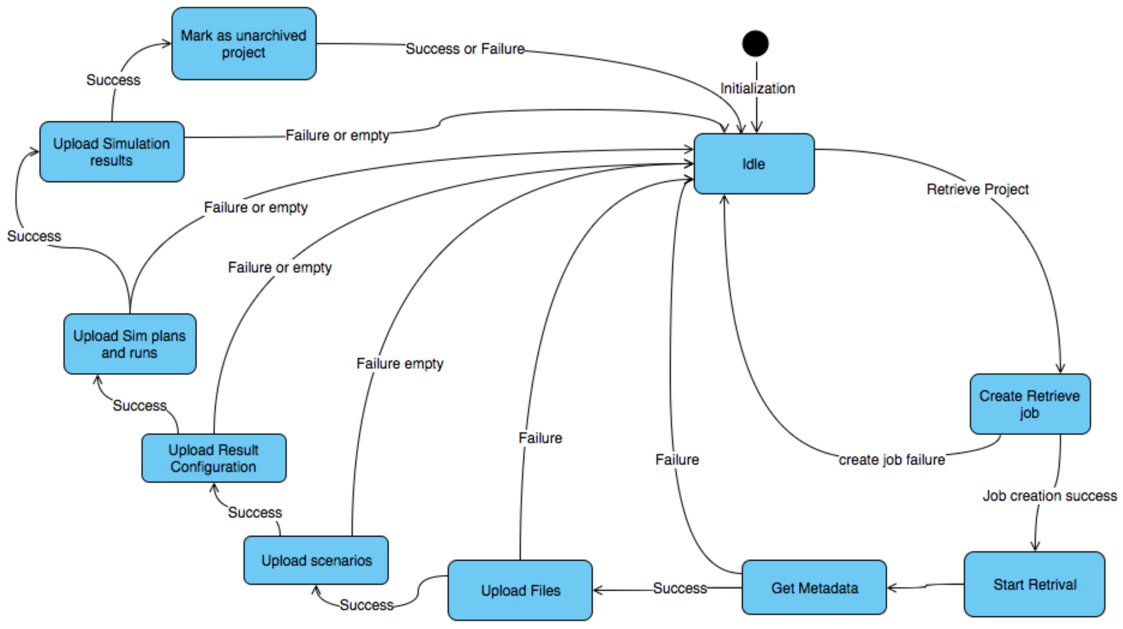


Figure 4.8: State Diagram of MARS project retrieval process considering empty states

Figure 4.8 illustrates the transitions that can occur in the retrieval process. The state digram has a very similar procedure as the archive (Figure 4.4), as both execute their actions in the same order. Also, marking of the resources is not done before the start process in contrast to the archive, because data coherency issues are not present as the archived data is stored in a centralized storage i.e. Synology which is accessed only by the Archive service.

Figure 4.9 illustrates the sequence diagram for the retrieve process. The starting step is similar to the archive process where a check is made if a process for the project is running or not. If a process is found running then the restore process will be denied. After a successful job creation the file metadatas are fetched from the archives. Using them the corresponding files are uploaded one after the other. In this process different type of input files such as GIS, Timeseries, and models can be uploaded. Section 4.3.1 mentions that the files are uploaded via the File-svc (In the UI) which determines the input file type and forwards it for processing which requires additional time. It is mandatory that all the uploaded files have a "FINISHED" status which can be acknowledged by making another request using the data id received when uploading the file. Using these ids the restoring file process waits until all the files have a "FINISHED" status. The request will be done in a designated time interval to avoid many network calls. In the case of a status "FAILED", request timeouts, unknown status, and server internal error the whole restore process will halt. This is necessary because the failed files cannot be used by the children resources.

The next step would be to get the scenario metadata. Restoring this data back to the system is not quite simple because it needs some additional work to be done. The problem arises from the fact that the archived data have attributes like the resource id which is changed as a new resource is uploaded. For more clarification, Listing 4.2 presents an example of the archived metadata of a file. This resource is needed so that the restore process can determine the different attributes (e.g. title, project id) while uploading a new resource. During a new file upload its data id would change, as a new id is assigned by the File-svc (See Listing 4.3). This is a big problem because the other resources such as a scenario cannot be uploaded until it knows the new data id that was assigned to the model it depends on. Listing 4.4 shows the archived scenario which has a reference to the data id from the archived file metadata. This is only one example as there are many attributes that must be taken care of. In order to solve this, a map using the old attribute as the key and the new id as the value will be made (See Listing 4.5). This way, while uploading the scenario resource it gets the new data id by using the old id from the map and replacing it during an upload.

Following a scenario upload, the other resources i.e. result configuration, simulation plan, simulation run will also use the same mapping strategy to replace the attributes required for restoring. Lastly, the simulation results will be restored from the archives. The restore process also waits until all the simulation runs are finished. Using a job id retrieved from the Database Utility service the status of the simulation restore can be known. Similar precaution for the file uploads are taken into consideration which prevents an infinite running of this process.

```
{
    "DataId":"7cae6055-d7fd-418e-9ba0-bdc2980ffb4c",
    "Title":"KNPGIS.zip",
    "Description":null,
    "ProjectId":"c5deed87-dd03-45c3-a0c4-fdf9f1a307a0",
    "UserId":"af7e045f-edf4-4df5-a9c8-6327186e6ddb",
    "Privacy":"PROJECT_PRIVATE",
    "State":"TO_BE_DELETED"
}
```

Listing 4.2: Snippet of archived MARS metadata resource

```
{
    "DataId":"27765261-8a65-45ab-bdeb-db8b5b7f8f43",
    "Title":"KNPGIS.zip",
    "Description":null,
    "ProjectId":"c5deed87-dd03-45c3-a0c4-fdf9f1a307a0",
    "UserId":"af7e045f-edf4-4df5-a9c8-6327186e6ddb",
    "Privacy":"PROJECT_PRIVATE",
    "State":"TO_BE_DELETED"
}
```

Listing 4.3: Snippet of the uploaded MARS metadata resource

```
{
    "MetaDataId":"7cae6055-d7fd-418e-9ba0-bdc2980ffb4c",
    "Description":"No description available.",
    "ClearName":"gis_vector_percipitation.zip",
    "AllowedTypes":["SHAPEFILE","GEOJSON"],
    "ParameterMapping":[]
}
```

Listing 4.4: Snippet of the archived MARS scenario resource

```
{
    "7cae6055-d7fd-418e-9ba0-bdc2980ffb4c":"27765261-8a65-45
      ab-bdeb-db8b5b7f8f4"
}
```

Listing 4.5: The mapped key value attributes that the scenario metadata would need

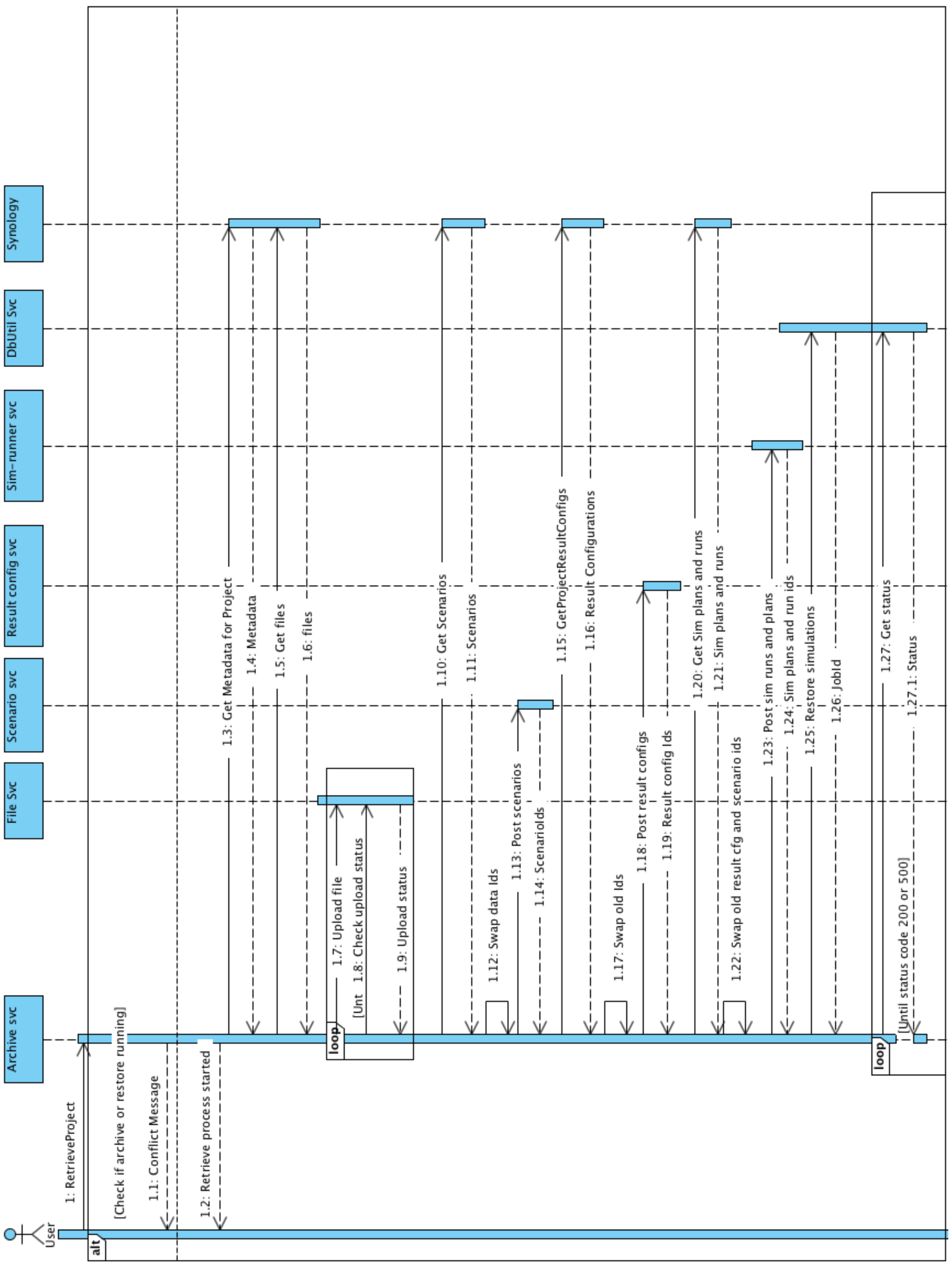


Figure 4.9: Sequence Diagram for the restore process

4.4 Status Retrieve Design

This section describes the design and behaviour for querying the status of a archive or retrieve process.

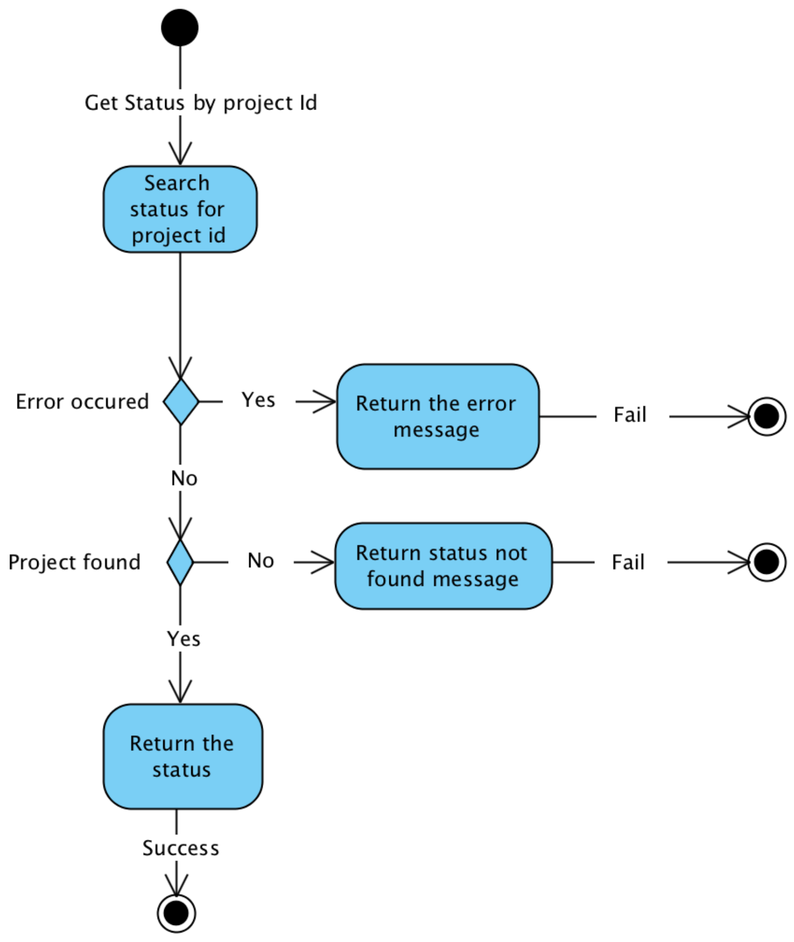


Figure 4.10: Activity Diagram of status acknowledgement process

Figure 4.10 illustrates the activity diagram for the status acknowledgement of a project (Listing 4.1). Firstly, the status for the project id is searched and if any error is caught (e.g. unable to connect to database) a message is returned to the client. If the status is not found a message stating the "project not found" is sent. Lastly, if the status is found it will be sent to the client.

4.5 Fault-Tolerance Design

This section describes the design of the archive service in case of different kinds of failures. The system being part of a large Distributed system, faces different difficult situation which must be handled for a stable application. Table 4.3 lists the possible errors which could occur with a brief description i.e. network issues, failure of a dependent service, sudden termination of the archive service.

Errors	**Description**
Network glitches	The MARS framework is built upon a complex distributed architecture and the communication between the services happen via a network (e.g HTTP, RPC). It is a possibility that the connection cannot be established for a small period of time due to network problems. This would lead the archive service to fail even though all the services are functioning.
Failure of a dependent service	There is a possibility that a service which the archive service is dependent upon goes down temporarily due to an unexpected failure or is in maintenance. The failure of the dependent service to reply would also generate an error in the Archive service.
Sudden failure of the Archive service	Like all the other services the Archive service is also prone to getting an unexpected restart. This restart would cause the running job to stop and the progress to be lost.

Table 4.3: Possible errors which could occur in the Archive service

Figure 4.11 illustrates the activity diagram which describes how the Archive service will be designed so that a recovery from errors mentioned in Table 4.3 could be handled. The main strategy for the failure mitigation is to re-run the process again from the beginning once an error occurs. The number of restarts and wait time can be configured by a programmer. This is designed in such a way so that the service avoids a deadlock situation of the resource being restarted infinitely. Additionally, a cumulative wait time for the restart can also be added so that there is some gap between the process restart.

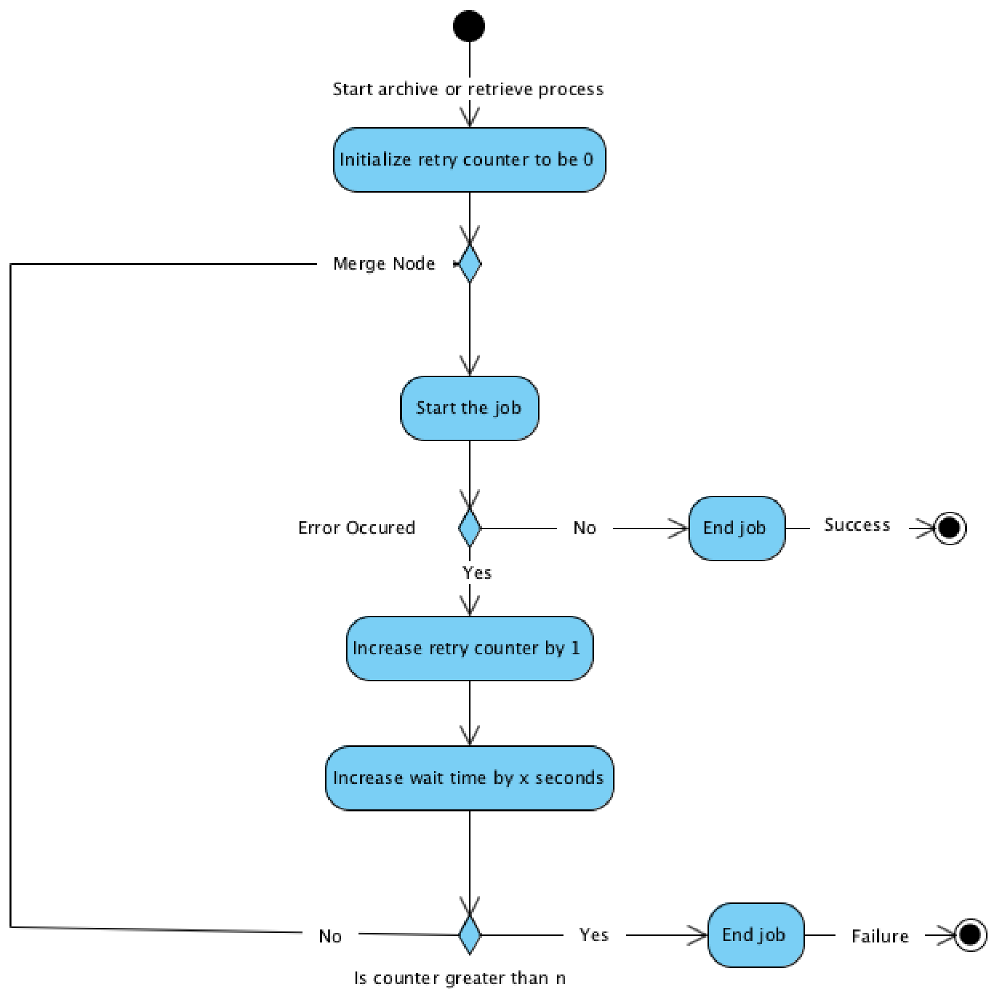

Start archive or retrieve process
Initialize retry counter to be 0
Merge Node
Start the job
Error Occured
No
End job
Success
Yes
Increase retry counter by 1
Increase wait time by x seconds
No
Yes
End job
Failure
Is counter greater than n

Figure 4.11: Activity Diagram for failure mitigation for the Archive service

5 Implementation

This chapter describes in depth about the programming languages, Object Oriented Design patterns, libraries and tools used to achieve the intended system design mentioned in Chapter 4.

5.1 Archive Process Implementation

This section gives an overview for the architecture of the archive process that would be responsible to move the project data to the Synology.

Figure 5.2 illustrates the class diagram for the archive process. This diagram attempts to depict only the top level classes which performs actions (e.g. archiving files, archiving simulation result). The archive process is is a complex task, thus involves many operations and communications. The operations include HTTP GET request to an external service, storing of received, data in to the Synology and forwarding the received data to the next component which requires it. This involves more classes than the number depicted and cannot be illustrated in a single diagram. This diagram is shown to point out the order of complexity that the archive process undergoes and how it is being implemented. Also, to make the modules of the archive service more reusable the components are separated into several classes like ArchiveMetadata, ArchiveScenarios etc. This separation of classes would later allow one to extend the archive process with less effort. It could be the case that in future a new requirement that involves the Archive service to only archive the input files, is needed. In this case as the components are already separated one can just use the interface of ArchiveFile for a quick implementation. In addition, different design patterns have been used in the Archive service like the Repository pattern [19] to help make the software more coherent.

Repository Pattern Implementation

Many components require access to the Synology storage to archive their respective data, that presents a problem of having data persistence logic duplication in many components. To solve this the repository pattern will be implemented where there would be an abstraction layer i.e. repository which provides the query interface to the component. This abstraction layer would be injected to the required components and they can just call the method to carry out persistence actions (CRUD). In addition, this also decouples the component from the type of storage being used i.e. Synology, so it would not matter for the component if the type of storage is changed from Synology to something else since it just needs the interface for persistence. Figure 5.1 illustrates how the repository acts as an abstraction layer for the client aiding the system to be more cohesive.

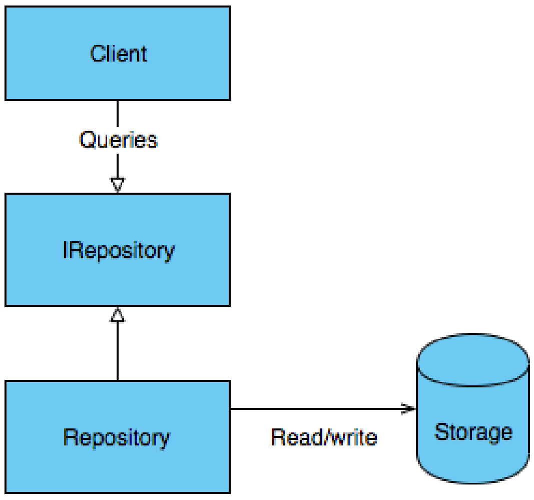
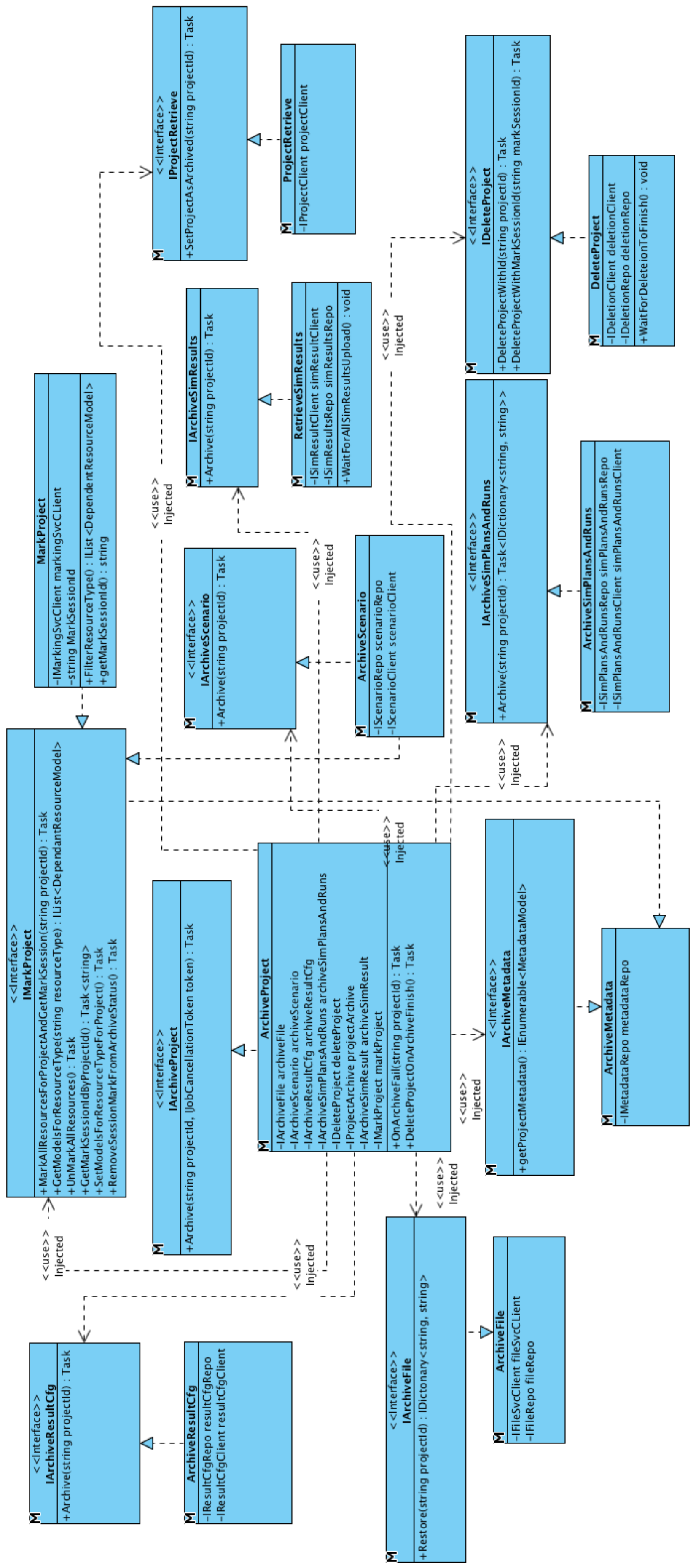


Figure 5.1: Repository Pattern overview



<<Interface>>
IProjectRetrieve
+SetProjectAsArchived(string projectId) : Task
ProjectRetrieve
-IProjectClient projectClient
<<Interface>>
IDeleteProject
+DeleteProjectWithId(string projectId) : Task
+DeleteProjectWithMarkSessionId(string markSessionId) : Task
DeleteProject
-IDeletionClient deletionClient
-IDeletionRepo deletionRepo
+WaitForDeleteionToFinish() : void
MarkProject
-IMarkingSvcClient markingSvcClient
-string MarkSessionId
+FilterResourceType() : IList<DependentResourceModel>
+getMarkSessionId() : string
IArchiveSimResults
+Archive(string projectId) : Task
RetrieveSimResults
-ISimResultClient simResultClient
-ISimResultsRepo simResultsRepo
+WaitForAllSimResultsUpload() : void
<<Interface>>
IArchiveScenario
+Archive(string projectId) : Task
ArchiveScenario
-IScenarioRepo scenarioRepo
-IScenarioClient scenarioClient
<<Interface>>
IArchiveSimPlansAndRuns
+Archive(string projectId) : Task<IDictionary<string, string>>
ArchiveSimPlansAndRuns
-ISimPlansAndRunsRepo simPlansAndRunsRepo
-ISimPlansAndRunsClient simPlansAndRunsClient
<<Interface>>
IMarkProject
+MarkAllResourcesForProjectAndGetMarkSession(string projectId) : Task
+GetModelsForResourceType(string resourceType) : IList<DependantResourceModel>
+UnMarkAllResources() : Task
+GetMarkSessionIdByProjectId() : Task<string>
+SetModelsForResourceTypeForProject() : Task
+RemoveSessionMarkFromArchiveStatus() : Task
<<Interface>>
IArchiveProject
+Archive(string projectId, IJobCancellationToken token) : Task
ArchiveProject
-IArchiveFile archiveFile
-IArchiveScenario archiveScenario
-IArchiveResultCfg archiveResultCfg
-IArchiveSimPlansAndRuns archiveSimPlansAndRuns
-IDeleteProject deleteProject
-IProjectArchive projectArchive
-IArchiveSimResult archiveSimResult
-IMarkProject markProject
+OnArchiveFail(string projectId) : Task
+DeleteProjectOnArchiveFinish() : Task
<<Interface>>
IArchiveMetadata
+getProjectMetadata() : IEnumerable<MetadataModel>
ArchiveMetadata
-IMetadataRepo metadataRepo
<<Interface>>
IArchiveResultCfg
+Archive(string projectId) : Task
ArchiveResultCfg
-IResultCfgRepo resultCfgRepo
-IResultCfgClient resultCfgClient
<<Interface>>
IArchiveFile
+Restore(string projectId) : IDictionary<string, string>
ArchiveFile
-IFileSvcClient fileSvcClient
-IFileRepo fileRepo
<<use>>
Injected

Figure 5.2: Class Diagram for the Archive process (Top level)

5.2 Retrieve Process Implementation

This section gives an overview for the architecture of the retrieve process which would restore the data back to the active system from the Synology, so that it can be used for running more simulations and analyze the results.

Figure 5.3 illustrates the class diagram for the retrieve process. The structure for the order of retrieve is very similar to the archive process since it has to follow the MARS resource hierarchy (Section 2.1.1). All the dependencies for the retrieve project class are injected using the Dependency Injection Container of ASP .NET framework. The restoring is done by getting the data from the Synology and then posting it back to the system using the respective service.

5.2.1 Addition of Functionalities in Other Services

The restore needs to call a lot of endpoints to upload the respective resource. After analyzing the MARS cloud and their available endpoints, it was seen that some functionalities are not present in the current system which are necessary for restoring the complete system. These functionalities are added to the services for a successful restore. Table 5.1 describes the functionalities that have been added in the required services.

Service	**Functionality**	**Description**
Project service	Add archived and is being archived mark in the project	The archived mark is necessary because it would provide a user information if the project being queried, is already archived, or is in the process of archiving. This has to be implemented using a GRPC communication, since this is the protocol in the project service, compared to the other services.
Scenario service	Return scenario id with the full scenarios	It is the case, when a full scenario is requested the scenario id is not returned. The id is required while retrieval because it needs to map the old scenario id with the new one. If the mapping fails the simulation plans cannot be created since they are dependent upon scenarios.

Sim runner service	Upload a simulation run without running a simulation	The Sim-runner service executes the simulation run producing an output (simulation result) when it is created. This is not desired by the Archive service because an simulation result is already available that needs to be restored. Therefore, an added functionality which just uploads the simulation runs without running a simulation is implemented.
Database utility service	Dump and restore for result database	The simulation results are normally big and it takes a longer and it is faster to perform a dump for this. The dump and restore functionality must be added, so that the archive service can restore and archive the simulation results.
Database utility service	Make the dump and restore a background job	It is important that the dumping and restoring process is implemented as a background task because this process is time consuming in comparison to the other resources i.e. scenario.

Table 5.1: Functionality implemented to the other services for retrieve process

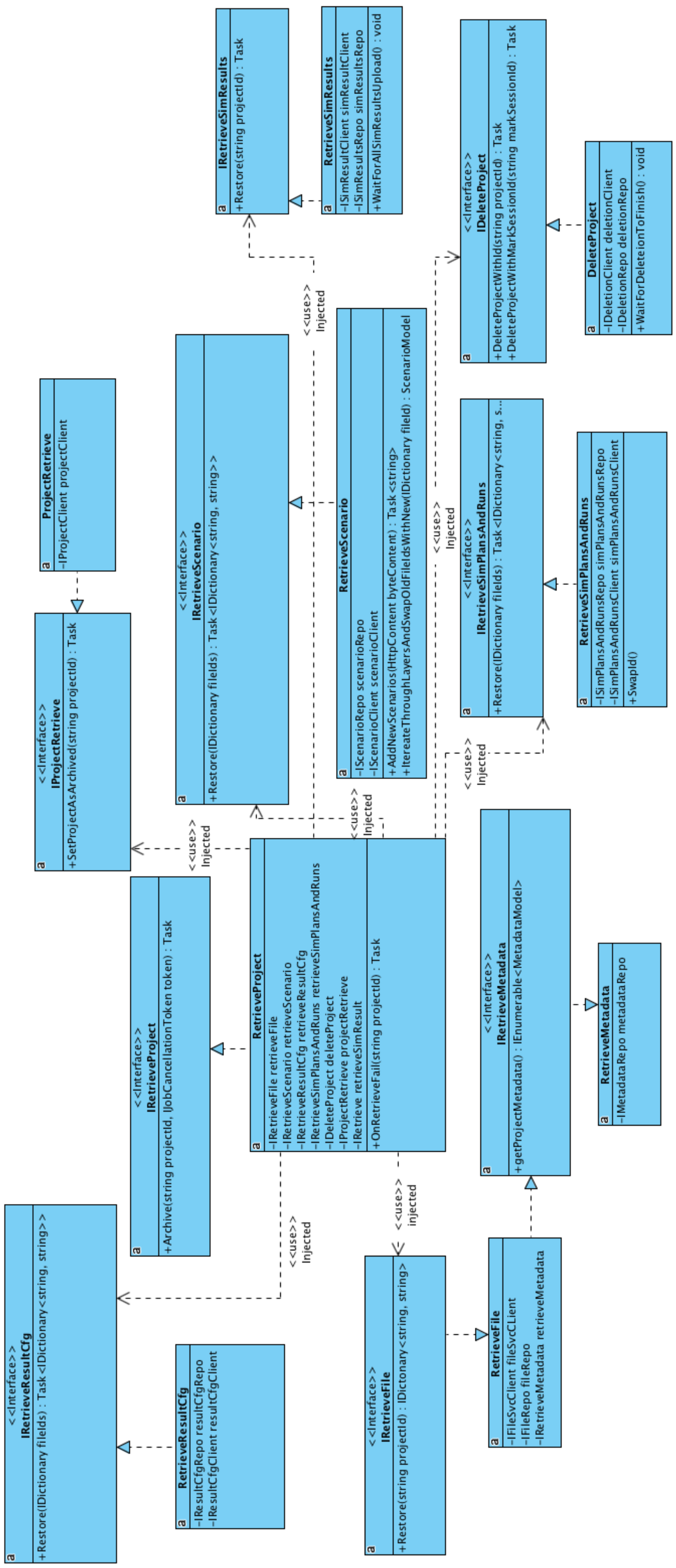

IRetrieveSimResults
+Restore(string projectId) : Task
RetrieveSimResults
-ISimResultClient simResultClient
-ISimResultsRepo simResultsRepo
+WaitForAllSimResultsUpload() : void
<<Interface>>
IDeleteProject
+DeleteProjectWithId(string projectId) : Task
+DeleteProjectWithMarkSessionId(string markSessionId) : Task
DeleteProject
-IDeletionClient deletionClient
-IDeletionRepo deletionRepo
+WaitForDeleteionToFinish() : void
ProjectRetrieve
-IProjectClient projectClient
<<Interface>>
IProjectRetrieve
+SetProjectAsArchived(string projectId) : Task
<<Interface>>
IRetrieveScenario
+Restore(IDictionary fileIds) : Task<IDictionary<string, string>>
RetrieveScenario
-IScenarioRepo scenarioRepo
-IScenarioClient scenarioClient
+AddNewScenarios(HttpContent byteContent) : Task<string>
+IterateThroughLayersAndSwapOldFileIdsWithNew(IDictionary fileId) : ScenarioModel
<<Interface>>
IRetrieveSimPlansAndRuns
+Restore(IDictionary fileIds) : Task<IDictionary<string, s...
RetrieveSimPlansAndRuns
-ISimPlansAndRunsRepo simPlansAndRunsRepo
-ISimPlansAndRunsClient simPlansAndRunsClient
+SwapId()
<<use>>
Injected
<<Interface>>
IRetrieveProject
+Archive(string projectId, IJobCancellationToken token) : Task
RetrieveProject
-IRetrieveFile retrieveFile
-IRetrieveScenario retrieveScenario
-IRetrieveResultCfg retrieveResultCfg
-IRetrieveSimPlansAndRuns retrieveSimPlansAndRuns
-IDeleteProject deleteProject
-IProjectRetrieve projectRetrieve
-IRetrieve retrieveSimResult
+OnRetrieveFail(string projectId) : Task
<<Interface>>
IRetrieveMetadata
+getProjectMetadata() : IEnumerable<MetadataModel>
RetrieveMetadata
-IMetadataRepo metadataRepo
<<Interface>>
IRetrieveResultCfg
+Restore(IDictionary fileIds) : Task<IDictionary<string, string>>
RetrieveResultCfg
-IResultCfgRepo resultCfgRepo
-IResultCfgClient resultCfgClient
<<Interface>>
IRetrieveFile
+Restore(string projectId) : IDictonary<string, string>
RetrieveFile
-IFileSvcClient fileSvcClient
-IFileRepo fileRepo
-IRetrieveMetadata retrieveMetadata
a

Figure 5.3: Class Diagram for the Restore process (Top level)

5.3 UI Integration

The MARS system is already equipped with a User Interface where one can create a project, manage the different resources, run the simulation, and analyze the results. The different back end services (See 2.1) are integrated in a Graphical User Interface so that a domain expert i.e. ecologists who are not computer experts could utilize the powerful tools that the framework offers. The aspect that this framework is intended to be used by users with less technical expertise makes it an absolute necessity that the functions which the Archive service delivers be part of the graphical system.

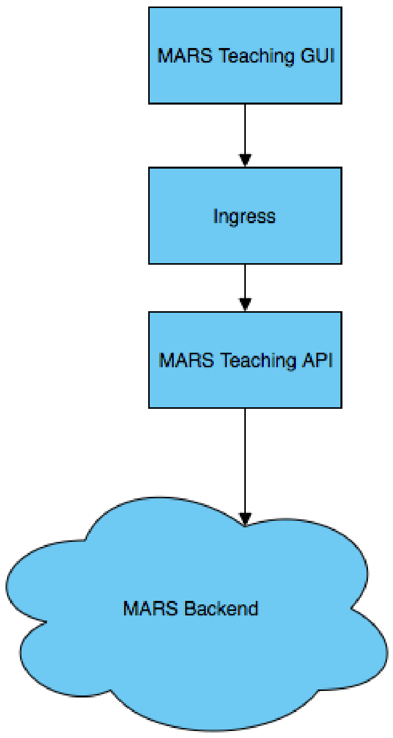


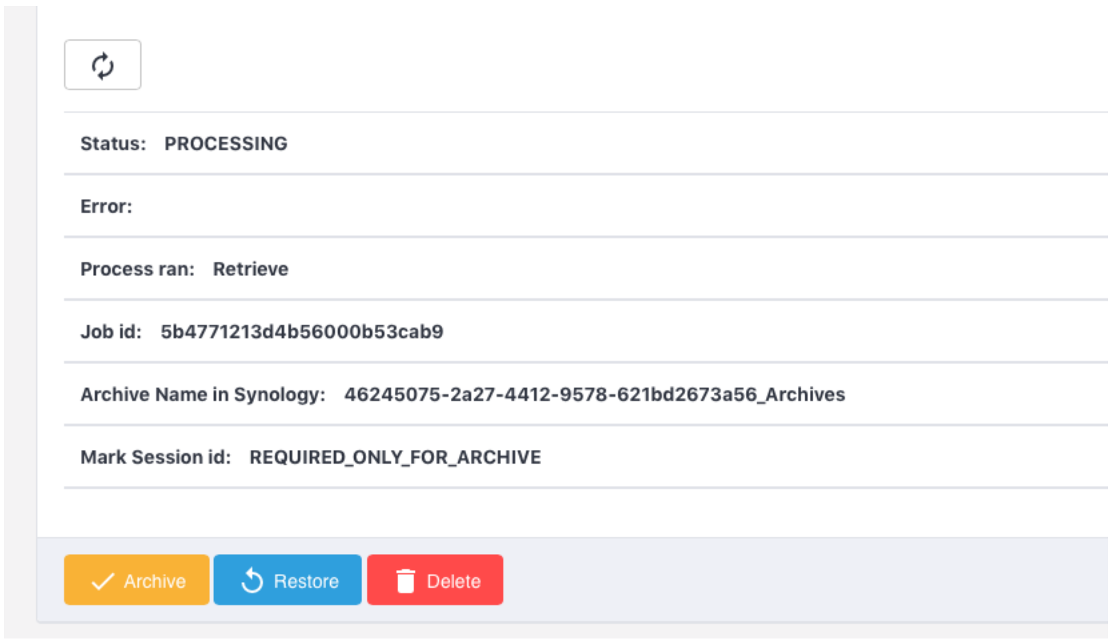
Figure 5.4: MARS cloud overview with the UI

Figure 5.4 illustrates a high level communication between UI and the back end services. It is to be noted that the backend services are running in an self contained environment which is not available to the outside ports and other clients. Therefore, to make the services available the Ingress[1] is present. Ingress contains a set of rules which allow inbound connections

[1] https://kubernetes.io/docs/concepts/services-networking/ingress/

to reach the services that are inside the cluster (encapsulated from outside). In the MARS system, the Ingress creates a reverse proxy to the teaching API. The teaching API is the bridge between the different backend services and the GUI. The purpose of making the requests via the Teaching API is to have a more secure system as it authenticates whether the client which is making the requests is the right owner or is an intruder. This check is done as the teaching API provides an authentication token i.e. Bearer token to the GUI and this token has to be included in the header of every request making it more secure as only the clients having the tokens can make a successful request. After a successful request to the Teaching API, it forwards the request to the concerning web service.

The first step needed for a complete integration of the Archive service in the UI is to determine which endpoints are going to be exposed to the outside ports i.e web browser. The endpoints decided to be exposed for the thesis are archive project, retrieve project, and get status. These endpoints are also programmatically added in the Teaching API written in GO Lang. Lastly, after the endpoints are made available to the UI components, the functions are implemented in the Teaching GUI service using the Angular 4 framework.


Status: PROCESSING

Error:

Process ran: Retrieve

Job id: 5b4771213d4b56000b53cab9

Archive Name in Synology: 46245075-2a27-4412-9578-621bd2673a56_Archives

Mark Session id: REQUIRED_ONLY_FOR_ARCHIVE

Figure 5.5: Archive service UI controls in the MARS Teaching UI

Figure 5.5 illustrates the integrated Archive service in the MARS Teaching GUI where a retrieve job is being executed. The archive process status can be seen in the same area where the retrieve information is being shown, when an archive is running.

5.4 Fault-Tolerance and Maintenance Implementation

The MARS system is very susceptible to different kinds of errors due to the architectural factors mentioned in the previous chapters. Therefore, a fault tolerant system is needed which would try to recover in case of any transient error. For the Archive service an open source framework named Hangfire is used for implementing the design in Section 4.5. This framework is used because it provides the functionality mentioned in the design section including many additional features i.e. reliability, easy maintainability and simplicity. It can be called reliable because when a background process is created, this framework would persist the input parameters, retry times, method name etc., in a database. This is also known as the Forget and Fire feature [27]. It would ensure that the jobs triggered once will always be executed and retried. As an example, if the Archive service restarts during a running process, in the general case the job cannot be recovered. However, this framework will check if there are any incomplete jobs and will start them automatically, fetching the stored job information from the database. This Forget and Fire feature ensures that the triggered processes would always run compared to the native threading offered by ASP .NET Core. Listing 5.1 shows the simplicity of how a job retry can be configured in case of an exception. One of the biggest disadvantages of this framework is that it is tightly coupled with a database. If there is a connection issue present between the Archive service and the database, the whole service will be facing a downtime. However, the benefits this framework provides outweighs its disadvantages.

```
private IServiceCollection BuildServices(IServiceCollection
   services)
{
    // hangfire job filter
    GlobalJobFilters.Filters.Add(new AutomaticRetryAttribute
        { Attempts = 4 });
}
```

Listing 5.1: Hangfire retry attempt configuration

AOP Logging

Logging is very important part of the service as it gives the maintainers key information of what has happened to the system. This makes logging important in each part of the system which leads to copying of the logging logic to the different components in the archive service violating the SOLID [17] principles. To solve this issue, the interceptor pattern is used. Using this pattern a decorator would be added in the class which wants to include the common logging logic. Listing 5.3 shows the implementation of the decorator "Intercept("logger")" which is injected in the class. The classes decorated with attribute would have a logging logic triggered. Listing 5.3 shows the main logic for the logging. In this logic, a success message with all the method arguments, time elapsed would be printed in case of a successful run and will log the exception in case of failure. This aids in the maintenance of the system as new feature are added only the decorator must be added to log the start, success, or exception of the method.

```
[Intercept("logger")]
public class ArchiveFile: IArchiveFile
{
    public ArchiveFile(IFileClient fileClient)
    {
        _fileClient = fileClient;
    }
}
```

Listing 5.2: Interceptor decorator example

```
private async Task InterceptAsync(IInvocation invocation)
{
//Before method execution
var stopwatch = Stopwatch.StartNew();

try
{
    //Calling the actual method, but execution has not been
       finished yet
    invocation.Proceed();

    //Wait task execution and modify return value
    if (invocation.Method.ReturnType == typeof(Task))
    {
        try
        {
            await InternalAsyncHelper.
               AwaitTaskWithPostActionAndFinally(
                (Task) invocation.ReturnValue,
                exception =>
                {
                    if (exception == null)
                    {
                        Console.WriteLine("Success: Class
                           Name: {0} Method: {1}  Duration:
                           {2:0.000} ms, Arugments: {3}",
                           GetClassName(invocation),
                            invocation.
                               MethodInvocationTarget.Name,
                               stopwatch.Elapsed.
                               TotalMilliseconds,
                               GetMethodArguments(invocation
                               ));
                    }
                    else
                    {
                        Console.WriteLine(exception);
                        throw exception;
                    }

                });
        }
        catch (Exception)
        {
            Console.WriteLine("Exception occured in async
               call");
        }

    }
}
```

Listing 5.3: Interceptor logger logic implementation

6 Testing

This chapter presents the methodology used for testing the correct functionality of the application. The validation consists a set of automatic test i.e. unit test, and integration test, including a manual system test to ensure the proper functionality of the service. The verification is done on both the functional aspects of the back end logic and also on the GUI controls for the Archive service.

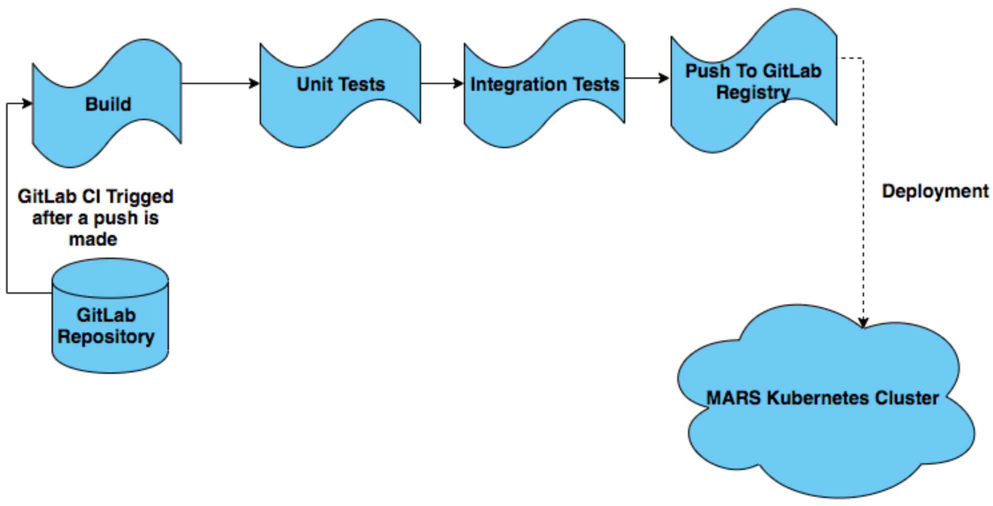


Figure 6.1: MARS Continuous Integration Pipeline build

Figure 6.1 presents the Continuous Integration system which is being followed by the MARS developer community. This pipeline plays an important role for the maintenance of the service because the automatic tests are executed here. The CI[1] pipeline would be triggered as soon as a new commit is being pushed to the remote GitLab[2] repository. This would then build the docker image of the service with the new changes. The next step would be to run the unit tests written for the service, which is a mandatory step. Lastly, if the pipeline passes,

[1]CI: Continuous Integration
[2]https://about.gitlab.com

the docker image will be pushed to the GitLab registry[3]. If this is successful, then the image can be used in one of the MARS Kubernetes cluster i.e. MARS beta, MARS production environments.

6.1 Unit Testing

These tests are designed to verify the individual components of the software which validates that each unit of the software performs as intended. A White Box Testing methodology has been applied for the unit testing. This method is used as the internal structure/requirements are known before hand. In this process, the specific inputs and outputs are predetermined and the tests are ran to verify if the expected output is produced by that input.

Figure 6.1 above depicts unit testing as an mandatory step for deployment of the Archive service. Running this test with every build process will boost the confidence in changing and maintaining the code for the Archive service. The ASP .NET platform offers a lot of unit-testing frameworks (e.g. XUnit, Nunit, MSTest/Visual Studio). Nunit is choosen as the preferred the testing framework for this service as it has a good reputation for fast testing and it is already integrated the IDE i.e. Rider (JetBrains) used for this work.

Making stubs, and mocks/fake objects are one of the important steps to do a proper unit testing. This statement can be backed up saying that unit tests are used to analyze only one module/method. To check only one module the other dependent objects have to be faked. The Archive service is designed following the SOLID [17] principle, the D stands for **Dependency Inversion Principle** where classes depend upon abstractions i.e. Interfaces not concrete implementation. The application of this principle has allowed an easy method to fake objects as the depending classes are just abstractions and the concrete implementations are injected by the ASP .NET Dependency Injection framework. A framework name as Moq [4] is used which can be used to fake objects i.e.Interfaces and return the desired output from them.

```
[Fact]
public async Task Archive_CorrectDataId_NoExceptions()
{
    // Arrange
    var fileRepoMock = new Mock<IFileRepo>();
    var fileClientMock = new Mock<IFileClient>();
    // Injecting the mock interfaces
```

[3]https://about.gitlab.com/2016/05/23/gitlab-container-registry/
[4]https://github.com/Moq/moq4/wiki/Quickstart

```
    var archiveFile = new ArchiveFile(fileRepoMock.Object, fileClientMock.Object);
    var testFileName = "test.zip";
    var testDataId = "4a97bd47-7713-4f4d-89c4-aacf04ae5d20";

    // Act
    // Using the Moq framework to return desired output to the method Archive
    fileRepoMock.Setup(m => m.AddFileForProject(It.IsAny<string>(), It.IsAny<byte[]>())).Returns(Task.CompletedTask);
    fileClientMock.Setup(m => m.GetFileAsync(It.IsAny<string>())).ReturnsAsync(new byte[]{1});

    Exception ex = null;
    try
    {
        await archiveFile.Archive(testFileName, testDataId);
    }
    catch (Exception e)
    {
        ex = e;
    }

    // Assert
    Assert.Null(ex);
}
```

Listing 6.1: Example of faking objects using Moq framework

Listing 6.1 illustrates an example of an unit test which is done on the Archive method for a file. The Moq framework injects the dependencies and in the setup method one can return the desired output from the methods. This unit tests verifies if all the dependent methods work as expected. Following the functional requirements mentioned in Section 3.1 the unit tests are designed to test if the core logic functions as expected which includes error handling as well.

6.1.1 Archive and Retrieve process test

The unit test designed for these processes are aimed to test the components responsible for each type of resource (See Table 3.1) and its dependent module. The class overview can be seen from Figure 5.2 and 5.3. Although, the figure does not show all the classes that the archive process is using but when one sees the attributes that a class is using it will give an good idea of the remaining classes under use.

Test if an background job is enqueued

For this test, after a successful API request for archiving a project, the module should create a background job using Hangfire. This test would check if the background tasks were created or not. As it is a requirement for the archive process to run a background job, so it must be ensured that a background job is created after a successful request. Listing 6.2 shows how the test was written to verify the creation of the background job.

```
[Fact]
public async Task
   ArchiveProject_SucessfulRun_ShouldBeEnqueued()
{
    // Arrange
    var client = new Mock<IBackgroundJobClient>();
    var iArchiveRepoMock = new Mock<IArchiveStatusLoggerRepo
       >();
    var mockMarkProject = new Mock<IMarkProject>();
    var mockStatusHandler = new Mock<IBackgroundHandler>();
    var controller = new ArchiveApiController(client.Object,
        iArchiveRepoMock.Object, mockMarkProject.Object,
       mockStatusHandler.Object);
    var projectId = Guid.NewGuid().ToString();
    var model = new ArchiveRetrieveStatusModel(){Status = "
       FINISHED"};

    // Act
    iArchiveRepoMock.Setup(mo => mo.AddInitialStatus(model))
       .Returns(Task.CompletedTask);
    mockMarkProject.Setup(mark => mark.
       MarkAllResourcesForProjectAndGetMarkSession(It.IsAny<
       string>()))
```

```
        .Returns(Task.FromResult(Guid.NewGuid().ToString()));
    iArchiveRepoMock.Setup(mo => mo.GetStatusForId(projectId)).ThrowsAsync(new ResourceNotFoundException(It.IsAny<string>()));

    // Assert
    await controller.ArchiveProject(projectId);
    client.Verify(x => x.Create(
        It.Is<Job>(job => job.Method.Name == "Archive" && (string) job.Args[0] == projectId && job.Args[1] == JobCancellationToken.Null),
        It.IsAny<EnqueuedState>()));

}
```

Listing 6.2: Hangfire Job creation test

Return Conflict Message in Case a Process for the Project is Running

This test checks, the denial of an archive or retrieve job creation when it is in "PROCESSING" state. It is of high priority that this check for conflict works as expected otherwise there could be cases of unwanted data loss.

Exception Catch

For this test, different checks are made in many modules whether the expected exception is caught in case of an error. It also includes as well a check for the general exception class. This test will ensure that some method will not just consume an exception when an unexpected error occurs. This is important because if the exceptions are not caught, it would be very hard to fix bugs if some are introduced in the near future.

```
[Fact]
public async Task
   MarkAllResourceAndGetSessionId_MarkingSvcClientReturnEmpty_ThrowsMar
   ()
{
    // Arrange
    var markingSvcClientMock = new Mock<IMarkingSvcClient>()
       ;
    var iArchiveRepoMock = new Mock<IArchiveStatusLoggerRepo
       >();
    var markProject = new MarkProject(markingSvcClientMock.
       Object, iArchiveRepoMock.Object);

    // Act
    markingSvcClientMock.Setup(mock => mock.
       MarkAndGetProjectResources(It.IsAny<string>())).
       ReturnsAsync(string.Empty);

    // Assert
    await Assert.ThrowsAnyAsync<MarkingFailedException>(
       async () =>
        await markProject.
           MarkAllResourcesForProjectAndGetMarkSession(It.
           IsAny<string>()));
}
```

Listing 6.3: Exception catch test example

Expected Value Checks

For this test, different modules are tested with a mock inputs (e.g. data id, mock models) and then the method under test is executed with an expected value. These tests are designed to ensure that the methods deliver correct results when a correct input from other modules are received. These tests would aid to discover a faulty implementation according to the requirements mentioned for this work.

Null or Empty Checks

For this test, the different modules are tested against the most famous Null pointer exception in the different modules and how it is handled. This is an important test case as the archive/retrieve process should still run and terminate accordingly if some resources are null. As an example, during a normal archive run if the result received from the scenario service is empty i.e. no scenarios in the project, the archive process should not fail due to a null pointer exception, rather finish the archive process as there are no children resources (See Figure 2.1). Listing 6.4 shows an example that, if the scenario client returns a null the process does not break but instead the method for archiving scenarios is not triggered.

```
[Fact]
public async Task Archive_NoScenarios_ScenariosNotAdded()
{
    // Arrange
    var scenarioRepoMock = new Mock<IScenarioRepo>();
    var scenarioClientMock = new Mock<IScenarioClient>();
    var markProjectMock = new Mock<IMarkProject>();
    var archiveScenario = new ArchiveScenario(
       scenarioRepoMock.Object, scenarioClientMock.Object,
       markProjectMock.Object);

    // Act
    markProjectMock.Setup(m => m.GetModelsForResourceType(It
       .IsAny<string>()))
        .Returns(DependantResourceMock.
           GetFilteResourceModels("scenario"));
    scenarioClientMock.Setup(m => m.GetScenariosById(It.
       IsAny<string>()))
        .ReturnsAsync(null);
    await archiveScenario.Archive(It.IsAny<string>());

    // Assert
    scenarioRepoMock.Verify(m => m.AddProjectScenarios(It.
       IsAny<string>()), Times.Never);
}
```

Listing 6.4: No scenarios added if resource empty

File Read Tests

For this test, the module for reading files from the Synology is tested. To do so, dummy files were added (mocking the existence of Synology without a network connection) in the build for the unit tests. This test reads the file and then compares it with the expected result. This test is designed to check if the file reading algorithm works as expected. Unfortunately, the module to write the data could not be verified because when the tests are built they are packed into a .dll file where a write operation is not allowed.

6.1.2 Test Coverage

A total of 174 unit test cases were made on the different modules existing in the Archive service at the time of writing this thesis. The total time for running all the test cases measured on average is 1m50s. For a proper calculation of the Test coverage by available features in the service, Table 6.1 gives an short overview of the type of modules and the different features available.

Modules	**Features**
Http communication client	File service client, Metadata service client, Scenario service client, Result configuration client, simulation plans client, simulation runs client and simulation results client.
API controller	Archive, Retrieve and background jobs.
Archive	Files, Metadata, Scenario, Result configuration, simulation plans, simulation runs and simulation results.
Retrieve	Files, Metadata, Scenario, Result configuration, simulation plans, simulation runs and simulation results.
File repositories	Read and Write.
Utilities	AOP loggers and HTTP helpers.

Table 6.1: Modules available for unit testing

Out of the 28 different features of the given modules, the unit tests were made for 26 of them. This gives a testing coverage of about 92%. Some Test cases could not be carried out due to the complexity of making the tests i.e. AOP[5] loggers, HTTP helpers, File Writers.

[5]AOP: Aspect Oriented Programming

6.2 Integration Test

The main aim of this test is to verify the communication between the different services that the Archive service is dependent upon. This test is designed in such a way that it calls the endpoints which are used in order to create a simulation run from adding files to running the simulation. As a result of checking these endpoints, it gives an additional benefit of detecting errors introduced from other services in the area of resource creation. As an example, assuming a service (excluding the Project, User, Marking, and Deletion service) made some changes which has some bug with resource creation and then if the Archive service integration test are executed, it would try to recreate the mock resource and would result in a fail.

6.2.1 Challenges

The Integration Tests are very beneficial to have a more stable system but realizing this test posed really enormous challenges which took considerably long time with some constraints following. Due to the Microservice architecture of the MARS framework the services are deployed as an independent entity which have their own databases. It was an enormous task to figure out how to combine all these independent services and have them running in a testing environment. The solution to this issue was to create a multi-container Docker application using Docker-compose. Here, the images of the services required are loaded and the MongoDB database for each service are seeded with mock data. It is also very important for the order for the services and the seeding to loaded in a specific order since the services are dependent upon each other.

```yaml
archive_svc_tests:
    image: nexus.informatik.haw-hamburg.de/microsoft/dotnet:
      2.0.0-sdk
    volumes:
      - ../:/mars-archive-svc
    entrypoint:
      - sh
    command:
      - ./mars-archive-svc/IntegrationTests/run_tests.sh
    links:
      - mongodb
      - metadata-svc
      - scenario-svc
      - file-svc
```

```
        - reflection-svc
        - resultcfg-svc
        - sim-runner-svc
        - sim-runner
        - mongo-seed
        - result-mongodb
        - result-mongo-seed
        - database-utility-svc
        - marking-svc
    depends_on:
        - mongodb
        - metadata-svc
        - scenario-svc
        - file-svc
        - reflection-svc
        - resultcfg-svc
        - sim-runner-svc
        - sim-runner
        - mongo-seed
        - result-mongodb
        - result-mongo-seed
        - database-utility-svc
        - marking-svc
```

Listing 6.5: Docker compose configuration snippet for Archive service Integration Test

Listing 6.5 shows a snippet of the docker compose file for running the Integration test. The depends_on attribute for the Archive service has many services in it. This means the archive_svc_test is waiting for the other services to load and the MongoDB to be seeded with mock data. Unfortunately, since the Project service does not use MongoDB but instead Postgres Sql as a database, for some unknown reason the synchronization of the seeding of the data did not succeed. Therefore the tests could not be executed. The Marking service and Deletion service endpoints that the Archive service uses have a dependency with the Project service which resulted as a unsuccessful test for them. Although, it is a point of interest in the future to investigate and to figure out the reason for this and complete the whole test.

6.2.2 Correctness of received data

For this test, the GET endpoints of the services were tested. A dummy model for the data which the Archive service expects is compared to the result form the GET endpoints. This check would aid to verify whether the services return a data model which the service expects. If any changes to the other services related to the data model is made, this test would detect it.

6.2.3 Correctness for uploading data

For this test, the POST and PUT endpoints were tested. It is designed to verify during the restoring process if the expected data model is still compatible with the services or if there is some error introduced after a new change. The process is conducted by uploading the data models using the mentioned endpoints and in return expecting a success status.

6.2.4 Correctness of response

This test is designed to check the interface of the API provided by the other services. This check would aid to validate if the API of the service returns a correct status as described by their swagger interface. As an example, while posting a scenario in the scenario service if the name and the data id of the model is already existing, an conflict status code will be returned by the request. Therefore, for this example an existing model would be uploaded and the test would verify if an conflict status is returned. Likewise, the other responses are also verified by this test.

6.2.5 Integration with the Database

This test is designed to test if a proper integration between the MongoDB and Archive service is maintained. The Archive service stores vital metadata which gives the client information about the status of the job and many other data that it needs. This check verifies the correctness of the read and write operations done with the database.

6.2.6 Test coverage

A total of 81 tests were made which would aid in verifying the integration of the Archive service with the dependent services and the database. The total time for running the test measured is on average 7m38s at the time of writing. Table 2.1 mentions the services that the Archive service has dependency towards. 6 out of 9 services which are successfully tested. Among the total services which were supposed to be tested only 67% were carried out. It is also a point of interest in the future to complete these requirements which would boost the reliability of the Archive service by investigating the issue.

6.3 System Test

This test verifies the requirements mentioned in Section 3.1 by performing manual GUI tests. System test are manual tests which ensures the correct behavior of the system.

6.3.1 Successful archive process start

For this test, it is verified that the archive process can be started from the MARS Teaching GUI with the precondition of no other process for this project is running. One Wolves and Sheep model is uploaded with other resources i.e. scenario, result configuration, simulation plan, simulation run, simulation result. As this test was to verify the successful execution of the process, only one of each resources were uploaded and checked.

6.3.2 Archive with a More Complex Model

For this test, the Kruger National Park (KNP) model is used instead of the Wolves and Sheep because the KNP is more complex since it requires more layers i.e. GIS, Timeseries, Geopotential layers. The successful archive of this model with all its resources i.e. scenario, result configuration, simulation plan, simulation run, simulation results were verified.

6.3.3 Successful Data Archive in Synology

For this test, the Synology drive where the archives were supposed to be uploaded were verified. The successful archive process of the Wolves and Sheep model was correctly stored in the archive folder inside the drive.

6.3.4 Successful Retrieve Process Start

For this test, it is verified if the retrieve process could be successfully started from the MARS Teaching GUI when no process for the project is running. Both the KNP and Wolves and Sheep models with its resources were restored back to the system with all its resources.

6.3.5 Correctness of the Restored Project

For this test, it is verified if the retrieved data are the same as for the archive. Also, a check is done whether the restored files can be used in the active system to produce further results (e.g. a simulation was ran from the restored simulation plan, a new scenario is created from the model).

6.3.6 Fault-Tolerance Test

For this test, a intentional error is created to verify if the implemented strategy for fault tolerance is executed. A successful verification for this strategy was checked. Also a check was done by removing the server from the cluster while a job was being processed. This checked and verified that the Archive service would restart the job in case of sudden failure after the server has be revived. This is the case only if the defined number of retires are not exceeded.

6.4 Performance Test

This test verifies the performance metrics of the Archive service by analyzing different numbers of files, file sizes and archive strategies.

6.4.1 Archive Performance

For this test, an archive process is executed and repeated 4 times to get an average general performance overview. Figure 6.2 illustrates a bar diagram for running an archive process (zipped simulation results) with 7 files, 2 scenarios, 2 result configurations, 2 simulation plans, 12 simulation runs, and results. The results seen in the figure shows that it took about 34.9s on average to run this process. It is also to be noted that the simulation results are zipped making their total size about 86 Mb. Figure 6.3 shows the archive process of the same project without compressing the simulation results. It can be seen that the file sizes for the uncompressed process is significantly higher in contrast to the compressed process with file size on avg about 683 Mb and 43s of processing time.

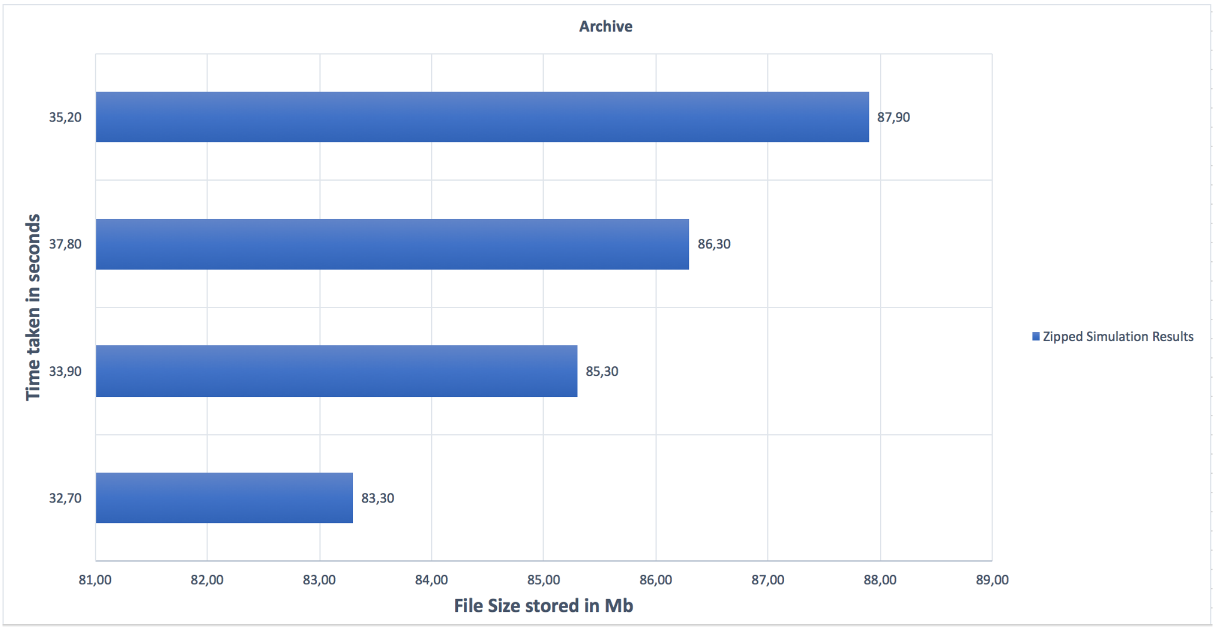


Figure 6.2: Overall performance of the archive process (compressed simulation results)

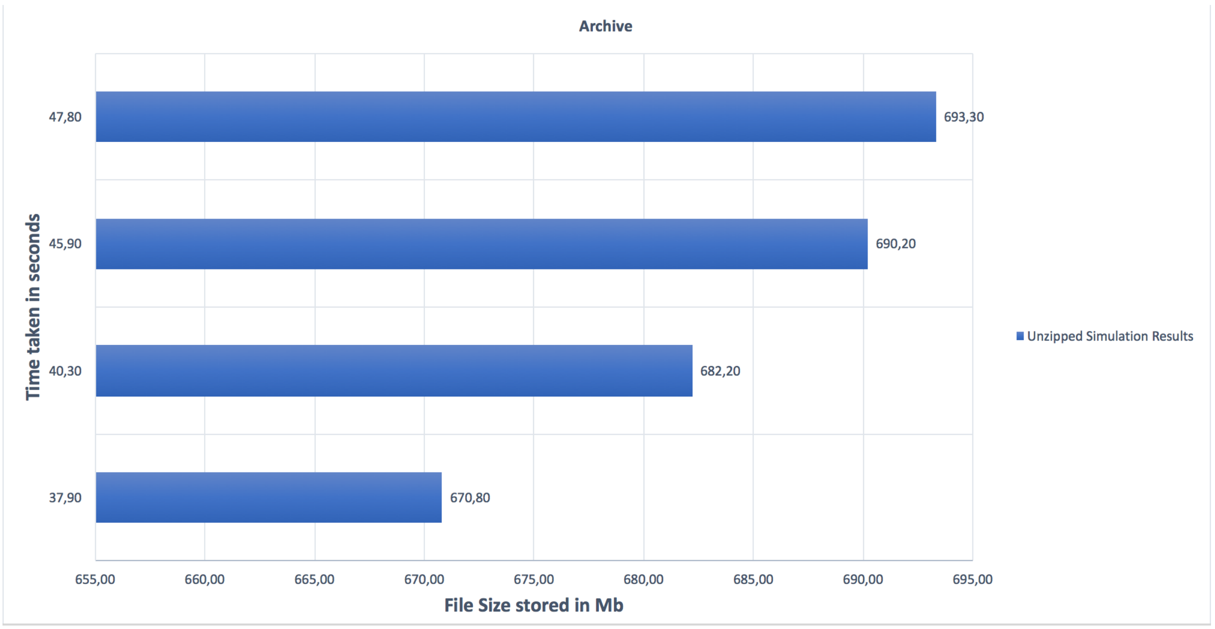


Figure 6.3: Overall performance of the archive process (uncompressed simulation results)

Unfortunately simulation results with larger file sizes could not be tested as no such simulation results was available at the moment. However, with this metrics it can be said that for smaller file sizes, it is a better to compress the simulation data while archiving it since there is a significant volume save and the time taken for the archive does not differ by much. It is also seen that the I/O is a bottleneck because the larger file (uncompressed) needs more time to archive than the smaller (compressed) file.

6.4.2 Retrieve Performance

For this test, the same project that was archived is being restored to analyze the performance of the retrieve process. Figure 6.4 illustrates the results of the retrieve process with the compressed simulation results which takes 4.3 mins on average. Figure 6.5 illustrates the results of the retrieve process using the uncompressed simulation results which took on average of 4 mins.

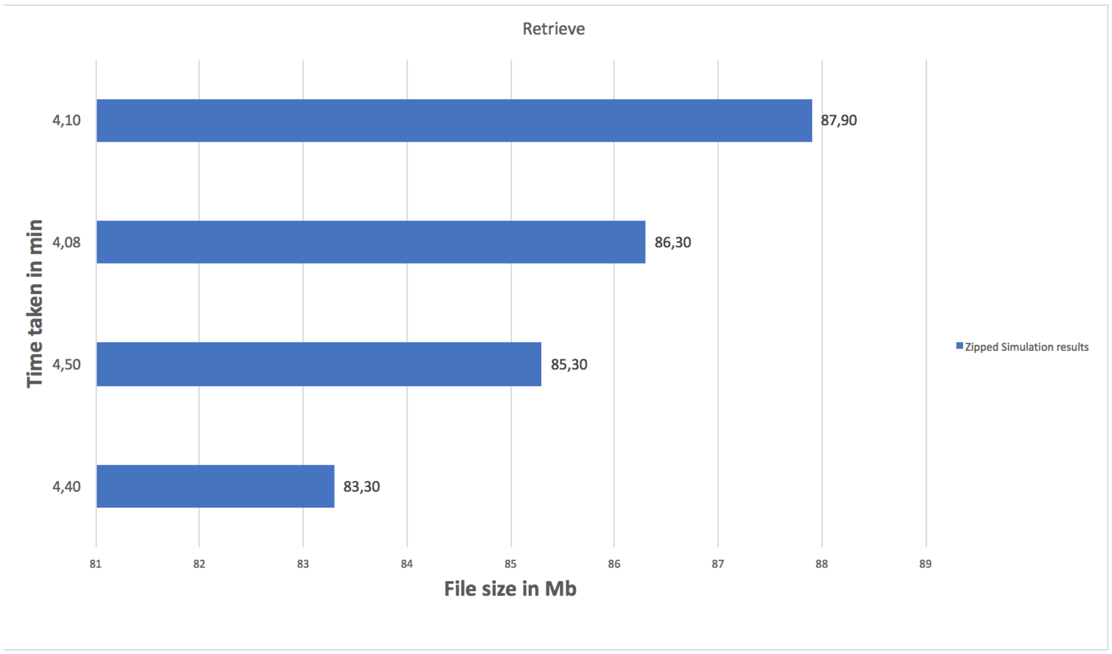


Figure 6.4: Overall performance of the retrieve process (compressed simulation results)

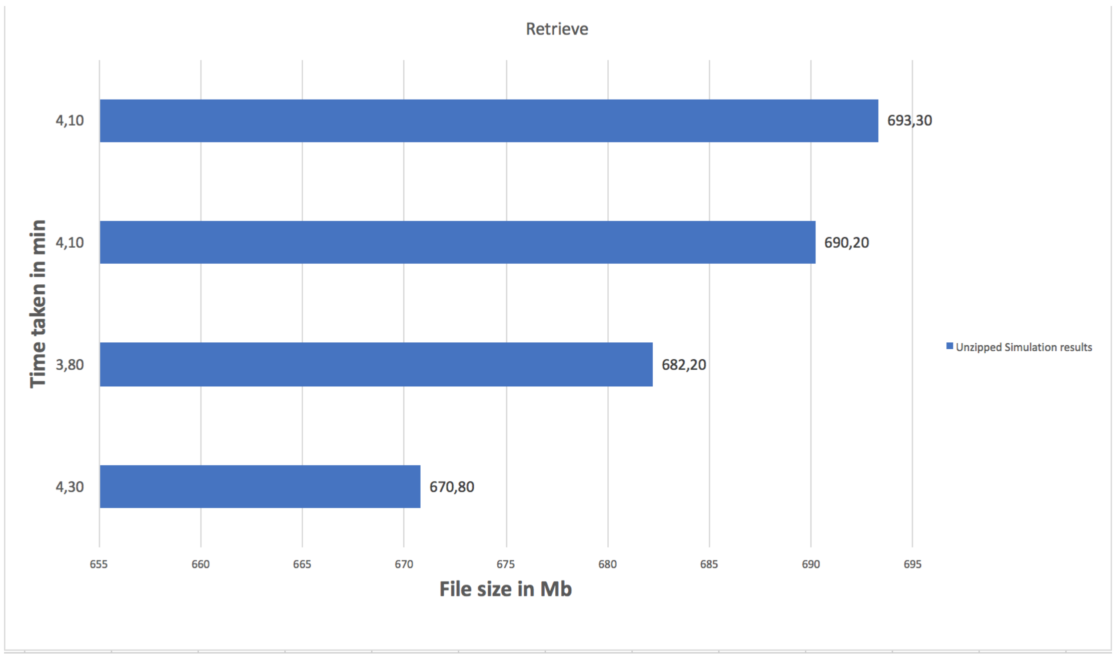


Figure 6.5: Overall performance of the retrieve process (uncompressed simulation results)

Also, to verify the complexity of the processes, the number of simulations were doubled i.e. 24 simulation and the performance was recorded and an average of 55 sec for archive and 8.2 mins for retrieve was observed. As a result, the processing time was doubled as expected.

7 Conclusion

This work presents the design and implementation of the Archive service within the MARS framework. The ideas shared in Chapter 4 concentrates on aspects of archiving and restoring the MARS resources stored in a distributed storage, managing the synchronization – control of the MARS Resources hierarchy mentioned in Section 2.1.1 and failure recovery using the Two Phase Commit strategy on decentralized data [28]. Also, a brief overview of the challenges and introduction to MARS system.

All the requirements mentioned in Chapter 3 were tested thoroughly using different testing strategies (See Chapter 6). The test done verify the correct functioning of the Archive service which can be used via the MARS teaching UI to archive and restore a project.

Finally, in the process of developing this system a deep understanding of Distributed System, Microservice together with the state-of-the-art technological stacks such as Kubernetes [10], Docker, etc., has been established. Together with this knowledge, an Archive service for the MARS framework has been implemented and integrated which aids in improving the performance of the MARS system by moving inactive data into the Synology storage.

7.1 Further Work

It is a point of interest that the Archive service would be further enhanced. Firstly, it is seen that the current UI design of the archive process is not very convenient for the users as the archived and unarchived projects are grouped together which may produce some confusion. One proposal could be to design a separate section where the archived projects could be shown. To aid this process the implementation of the mark showing if the project is archived or not is already included, which is changed via the Archive service.

Secondly, improving the performance of the archive and retrieve would also be a factor to be considered next. The archive and retrieve process are implemented as an atomic operation. Therefore, in case of failure the process is repeated from the beginning, causing big performance loss for larger projects. More research and effort have to be put forward to obtain a solution for this issue.

It can be noticed from the sequence diagram that the resources are being persisted right after they are successfully retrieved rather than a bulk operation (e.g. sequence number 1.8 in Figure 4.5). This is implemented considering a possibility that in the near future the archive service would archive the project as a snapshot. In case of archive failure the process could be resumed from the snapshot in contrast to the current atomic implementation of the process.

Lastly, to strengthen the stability when deploying this service the test i.e. Unit, Integration tests must increase the test coverage so that more potential errors could be avoided.

References

[1] DiskStation Manager | Synology Inc. [Online] https://www.synology.com/en-global/dsm.

[2] Q. A. Chaudhry. An introduction to agent-based modeling modeling natural, social, and engineered complex systems with NetLogo: a review. *Complex Adaptive Systems Modeling*, 2016. Springer Berlin Heidelberg, ISSN: 9780262328111.

[3] T. Eizinger and M. D. Bernhard Löwenstein Lukasz Juszczyk Vienna. API Design in Distributed Systems: A Comparison between GraphQL and REST, 2017. University of Applied Sciences Technikum Wien.

[4] T. Erl, P. Merson, and R. Stoffers. *Service-Oriented Architecture*. Prentice Hall, 2016. ISBN: 978-0-13-385858-7.

[5] E. Evans. *Domain Driven Design Quickly*. Addison-Wesley, 2006. ISBN: 978-0-321-12521-7.

[6] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. University of California, Irvine.

[7] F. Format, D. Model, T. P. Software, and P. Model. High Level Introduction to HDF5 Introduction to HDF5. pages 1–25, 2016.

[8] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Computing Surveys*, 33:51, 2002. ISSN: 01635700.

[9] Google. [Online] https://grpc.io.

[10] Google. Production-Grade Container Orchestration - Kubernetes. [Online] https://kubernetes.io/.

[11] B.-C. GRAEME F. *Geographic Information Systems For Geoscientists*. PERGAMON, 2016. ISBN: 0 080 418 678.

[12] L. Grundmann. Globale Sensitivitats und Unsicherheitsanalyse mit MARS. page 114, 2018. HAW Hamburg.

[13] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983. ISSN: 0360-0300.

[14] M. Hauck. NoSQL Databases Explained. [Online] https://www.mongodb.com/nosql-explained, 2014.

[15] HAW Hamburg. - MARS Group. [Online] https://mars-group.org/.

[16] H. Holzmann, V. Goel, and A. Anand. ArchiveSpark: Efficient Web Archive Access, Extraction and Derivation. *Proceedings of the 16th ACM/IEEE-CS on Joint Conference on Digital Libraries*, pages 83–92, 2016. ISBN: 978-1-4503-4229-2.

[17] H.-J. Hotop. Software Engineering Contents (main subjects) HAW Hamburg. 2015.

[18] iso. Warc file format. [Online] http://archive-access.sourceforge.net/warc/warc_file_format-0.16.html. IIPC Framework Working Group.

[19] P. Lalanda. Shared repository pattern. In *Proc. of the 5th Pattern Languages of Programs Conference (PLoP 1998)*, page 10. Thomson-CSF Corporate Research Laboratory, 1998.

[20] H. H. MARS. Mars cloud.

[21] F. Martin. Microservices Guide. [Online] https://martinfowler.com/microservices/.

[22] Martin Fowler. PolyglotPersistence. [Online] https://martinfowler.com/bliki/PolyglotPersistence.html, 2011.

[23] Mozilla. [Online] https://developer.mozilla.org/en-US/docs/Web/HTTP.

[24] Mozilla. [Online] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[25] S. Newman. *Building Microservices*. Oreilly, 2015. ISBN: 978-1-491-95035-7.

[26] U. North Carolina State University. Definition and examples of time series. [Online] https://www.stat.ncsu.edu/people/martin/courses/st782/Notes/Definition.

[27] S. Odinokov. Hangfire. [Online] https://www.hangfire.io.

[28] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, 1983. ISSN: 07342071.

[29] D. Savic, F. Potorti, F. Furfari, M. Pustiek, J. Beter, and S. Tomazic. A tool for packaging and exchanging simulation results. *Recent Advances in Modeling and Simulation Tools for Communication Networks and Services*, pages 443–462, 2007. ISBN: 978-0-387-73907-6.

[30] SmartBear Software. Swagger. [Online] https://swagger.io/.

[31] R. Stephens. *Beginning Software Engineering*. John Wilez and Sons Inc., 2015. ISBN: 9781119209515.

[32] C. D. L. Torre, B. Wagner, and M. Rousos. .NET Microservices: Architecture for Containerized .NET Apllications Edition 1.0. 2017.

[33] M. Van Steen and A. S. Tanenbaum. *Distributed Systems*. Number 3. Maarten van Steen, 2017. ISBN: 978-90-815406-2-9.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. pages 307–320, 2006.

[35] J. Weyl, D. Glake, and T. Clemen. Agent-based Traffic Simulation at City Scale with MARS. *Proceedings of the 2018 Spring Simulation Multiconference*, page 9, 2018. In: Proceedings of the 2018 Spring Simulation Multiconference, Baltimore, Maryland, USA : Society for Computer Simulation International, 2018 (ADS 18) - accepted.

8 Appendix

This appendix shows one how to build the archive service locally.

Build Archive service

- Clone the mars-archive-svc from the MARS gitlab repository.
- Install Kubernetes command line tool using the guide https://kubernetes.io/docs/tasks/tools/install-kubectl/
- Install Docker.
- Login to docker and nexus see MARS Confluence documentation.
- Make sure to check the Kubernetes deployment file to check the image name i.e. it should be with the dev tag (See Listing 8.2)
- Make changes and run the bash script called start.sh inside the archive folder

```
#!/usr/bin/env bash
GITLAB_REGISTRY="docker-hub.informatik.haw-hamburg.de"
PROJECT="mars/mars-archive-svc"
SERVICE_NAME="archive-svc"

rm -rf out

dotnet publish -o out

cd ..

docker build -t ${GITLAB_REGISTRY}/${PROJECT}/${
   SERVICE_NAME}:dev .
docker push ${GITLAB_REGISTRY}/${PROJECT}/${SERVICE_NAME
   }:dev

kubectl -n mars-mars-beta delete pod -l service=${
   SERVICE_NAME} --force
```

Listing 8.1: Archive service local build script

```
spec:
      serviceAccount: mars-group-serviceaccount
      containers:
      - image: docker-hub.informatik.haw-hamburg.de/mars/
         mars-archive-svc/archive-svc:dev
```

Listing 8.2: Archive service deployment file snippet

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering 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 21, 2018	
City, Date	sign