Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Integration of the Cloud Native Renew Plugin into the Technical Realization of the Mushu Architecture

vorgelegt von

Marvin Taube

Fakultät MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich: Algorithmen, Randomisierung, Theorie

Studiengang: Bachelor Informatik

Matrikelnummer: 7158355

Erstgutachter: Dr. Daniel Moldt

Zweitgutachter: Michael Haustermann

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Integration of the Cloud Native Renew Plugin into the Technical Realization of the Mushu Architecture

presented by

Marvin Taube

Faculty MIN-Fakultät

Department Informatik

Group: Algorithmen, Randomisierung, Theorie

Course: Bachelor Informatik

Student Id: 7158355

Supervisor: Dr. Daniel Moldt

Co-Supervisor: Michael Haustermann

Abstract. Die Integration von neuen Elementen in bestehende Architekturen ist ein wiederkehrendes Problem in der Informatik. Das ist insbesondere der Fall, wenn schon eine komplexe Architektur besteht und das zu integrierende Element eine große Feature-Erweiterung mit sich bringt. Diese Arbeit beschäftigt sich mit der Integration des Cloud Native Renew Plugins in die Mushu Architektur. Ziel ist es der Gesamtumsetzung der Mushu Architektur näherzukommen. Dabei legt diese Arbeit den Fokus auf das Aufsetzen und Dokumentieren einer Produktumgebung, um das Zusammenspiel von Mushu und dem Cloud Native Plugin zu testen, und im späteren für die Integration von weiteren Teilkomponenten genutzt werden kann. Der Fokus wird dabei auf das Cloud Native Renew Plugin gesetzt, welches als Komponente in die Architektur integriert wird. Die Integration wird dann mit einem zusammenhängenden Gesamtbeispiel in der Produktumgebung validiert. Das Ergebnis ist eine Ausgangsbasis bestehend aus einer laufenden Produktumgebung und dazugehöriger dokumentation, welche dazu dient weitere Komponenten in die Mushu Architektur zu integrieren und diese damit zu vervollständigen. Zudem wird eine beispielhafte Integration von dem Cloud Native Renew Plugin in die Produktumgebung vorgenommen, welche als beispielhaftes Vorgehen für weitere Integrationen in die Mushu Architektur genutzt werden kann.

Abstract. Integrating new elements into existing architectures is a recurring problem in computer science. This is especially the case when a complex architecture already exists and the element to be integrated involves a large feature extension. This work integrates the Cloud Native Renew Plugin into the Mushu architecture. The overall goal is to get closer to the complete implementation of this architecture. This work focuses on setting up and documenting a production environment to test the interaction of Mushu and the Cloud Native Plugin, which can later be used to integrate other subcomponents. The focus will be on the Cloud Native Renew Plugin, which will be integrated into the architecture as a component. The integration is then validated with a coherent overall example in the production environment. The result is a foundation consisting of a running product environment and associated documentation, which can be used as a guideline to integrate additional components into the Mushu architecture and thus complete it. In addition, an exemplary integration of the Cloud Native Renew Plugin into the production environment is carried out, which can also be used as a guideline procedure for further integrations into the Mushu architecture.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Different solutions and implementations should always be considered to develop a distributed environment with Cloud Native aspects. Thereby, it is important to bring the solutions developed for this purpose into a format in which they work this environment. Integrating newly developed components into an existing distributed system is like in any other software project an essential part in the development. Primarily when the integration of a feature significantly extends the functionality of the current system.

The simulation software Renew gets a fundamental extension of functionality with the Cloud Native Plugin. With the integration of this plugin, it is possible to ensure that Cloud Native aspects are embedded in a distributed Renew system such as Mushu. It allows the previously developed theoretical concepts and solutions for a local version of the simulator to be integrated into the large scope of a distributed system, thus enabling an even more extensive distributed simulation.

1.2 Objective

This theis aims to provide a description and documentation of the integration process of Cloud Native aspects into an existing distributed Renew architecture. As a model of this integration was given in the technical realization of the Mushu architecture developed by (Röwekamp 2023a). The progressive integration of the subcomponents to achieve the integration of the Cloud Native Plugin into the Mushu architecture is presented in this thesis.

The main focus will be on the integration of the Cloud Native Renew plugin in the current version of the technical realization of the Mushu architecture. Other required components of the Mushu architecture will be partially addressed. The goal is to bring the current implementation of a distributed Renew System closer to the complete technical realization of the Mushu architecture.

1.3 Structure

The beginning of this thesis will introduce the necessary basics in chapter 2. In doing so, a basic foundation is provided to understand this thesis. Different technologies like Kubernetes and Docker and theoretical concepts like Cloud Native and Petri Nets will be discussed, and an insight into Renew is given, which is the central software for this thesis.

After the basics, requirements are defined to achieve the integration of the Cloud Native Renew Plugin. The work on these requirements then takes place in the form of smaller prototypes. These are always considered a subproblem whose solution is essential to achieving the goal of this thesis. The prototypes are divided into different chapters, which separate them thematically.

It starts with an overview of the current status of the Cloud Native Plugin in chapter 4, in which the current results from previous work done are compiled and partially supplemented.

This is followed in Chapter 5 by a summary of the work phase before Jan Henrik's disputation. Here, the basic work needed for integrating a plugin into the Mushu architecture is discussed. In particular, a focus is placed on the relevant elements for the Cloud Native Renew Integration.

This is followed by chapter 6, where the production environment is set up. The existing computer cluster at the University of Hamburg is examined and brought into a state where it can be used for the later integration.

The actual integration of the Cloud Native Renew Plugin follows in Chapter 7. This describes how the integration process for a plugin in the Mushu architecture can look and which elements must be considered.

Chapter 8 follows the elaboration on the extension of the Cloud Native Renew Plugin. The focus is set around to improve the usability of the provided functions from a reference net context. Thus, enabling easy to use Cloud Native Renew functionalities while developing reference nets. In addition, a focus is placed on documenting the new use of the plugin.

In the last chapter of the main part, the plugin's integration is validated. There the collected contents from 6 and 8 are validated with the help of a coherent example. Also focusing on the documentation to provide a good understanding of the use of the production environment.

Finally, I will evaluate the success of the integration and whether the requirements have been met. In addition, an outlook on what further work is necessary to complete the technical realization of Mushu and other possible topics in that area is given.

Chapter 2

Basics

2.1 Cloud Nativity

Cloud Nativity describes a concept that can be applied to software and software systems. There are various definitions of Cloud Nativity, for example the definition from the Cloud Native Computing Foundation[1]:

> *Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*
>
> *These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.*
> - (Cloud Native Computing Foundation 2023)

Another definition comes from (Garrison and Nova 2017), which has been used in the context of Renew and the development of the Cloud Native Plugin. According to Garrison and Nova, Cloud Nativity can be divided into four core characteristics: Resilience, Agility, Operability, and Observability. Since these characteristics are mentioned in both definitions, it is appropriate to look at them in more detail.

Resilience

Resilience requires that a system continues to run when a failure occurs. It should be noted that it is not about avoiding all possible errors. The focus is on ensuring that the system is capable of repairing the failure should it occur.

Agility

Agility requires that the system is quickly deployable and can be developed in small steps. This applies to the system itself, but also has a certain focus on the development that happens on the system. Furthermore, the system should be able to adapt to changing environmental conditions.

[1] Cloud Native Computing Foundation: https://www.cncf.io/

Operability

Operability requires that the system can be controlled and managed at any given time. This guarantees the user direct control over the system and processes without having to access additional resources. The control of the system should also be possible without locality. This means the user should have complete external control over the system and its functionalities.

Observability

Observability requires that the system provides information about itself and its environment. Real-time information is just as relevant as log information or similar data. The goal is to achieve that the system is transparent for the user.

In the following, the terminology of Resilience, Agility, Operability, Observability is used as defined in (Garrison and Nova 2017).

2.2 Java Spring

Java Spring[2] is a framework, for web-based applications in Java. It allows to create an executable application in a short time and with little effort. In addition, there is a very extensive ecosystem, which provides many extensible functions (Spring 2023). An example of this would be Spring Boot Admin, which is used on the context of the Cloud Native plugin. Another part in this ecosystem is the Spring Boot project itself, which was used to initialize the Spring Project used as a foundation in the Cloud Native plugin.

First approaches on how to connect a Spring context with Renew can be found in (Jan Henrik Röwekamp 2018). An actual implementation has been done together as part of the (Röwekamp, Taube, et al. 2021) publication.

Since the concept of dependency injection is related to the Cloud Native approach, and Spring Boot Admin plays an important role of monitoring and consequently documentation, it makes sense to look at these two points again in more detail.

2.2.1 Dependency Injection

The Java Spring framework allows to use the concept of dependency injection. In short, this concept allows us that the users of this system do not have to worry about the initialization of individual components. The developer specifies how to initialize objects and the framework does the work and injects this initialization where it is needed. This concept is explained very well in the article (Fowler 2014) where the fundamental idea behind dependency injection is demonstrated with simples examples.

This concept is relevant for the Cloud Native Plugin, since it avoids strong dependence on objects. The internal decoupling of components achieved in this way is directly in line with the Cloud Native design pattern. Parts of the dependency injection can be found in different parts of the plugin code.

2.2.2 Spring Boot Admin

Spring Boot Admin is an extension for a Spring Boot application and provides an admin interface with different status information. This provides functions like health monitoring, build information, log data, HTTP endpoints, etc. The overview can be accessed through

[2]Java Spring Framework: https://spring.io/

a predefined port via HTTP. The Spring Boot Admin Project ist managed by (Edmeier 2023), and the project itself can be found on their GitHub page[3].

In the context of the Cloud Native Plugin it is interesting, because it uses Spring Boot Admin primarily for monitoring the Renew instance. Using the built-in elements made it easy and fast to obtain important information about the system and Renew. By extending the functionality using the StatusMonitor plugin created for this purpose, the Spring Boot Admin Interface also allows displaying information from Renew plugins unrelated to Cloud Native. In the overall context, Spring Boot Admin can be seen as the first variant of the status monitor seen in the Mushu architecture 2.2.

2.3 Docker

Docker is an open-source project that can be used to develop software that can run in an encapsulated environment. The goal is to execute the software independently and separately from the given infrastructure.

Docker itself uses a client-server architecture. For example, a client can communicate with a Docker host via the command line interface. The Docker host itself can be any computer. Various images can then be created on this Docker host. The images can then be interpreted by the Docker host machine and executed as a containerized instance on that computer. The images can be created and stored directly on the Docker host or made available in a registry such as Gitlab if multiple Docker hosts are to use the same images (Docker 2023a).

The containerized instance is like a virtual machine. The software and its dependencies are encapsulated from the host system and run in their independent environment. The most significant difference to the typical virtual machine is that all containers of a host run with the same container engine. This container engine provides an operating system (OS) for all containers and runs on the host's operating system kernel. Thus, each container does not have its own kernel and does not have the requirement to have its own OS. This implementation saves resources on the host system and results in more computing capacity on the host system (Microsoft 2023).

2.4 Kubernetes

Kubernetes is an open-source tool for managing complex platforms. It can be used in various scenarios to deploy cloud-based applications and, according to (IBM 2023), competes with the common use of virtual machines. In addition, Kubernetes already provides many functions for implementing Cloud Nativity aspects to the given architecture.

The difference between virtual machines is that Kubernetes introduces an infrastructure where a master communicates with various workers. The master and workers are considered nodes in the implementation, usually servers or PCs. The master communicates with the workers using an API and provides various management endpoints for the Kubernetes network. The software can then run containerized on the workers in pods. Pods are like minimalistic virtual machines that can execute the given environment by the container. A template of this software as a pod can be passed to Kubernetes, ensuring that the respective template is always available in the cluster. Kubernetes takes control of the deployment of the pods on the workers and decides which of them is best suited for running the software (Kubernetes 2023a).

[3]Spring Boot Admin: https://github.com/codecentric/spring-boot-admin

Such a system's advantages include self-healing by automatically restarting crashed pods or horizontal scaling, which makes it possible to automatically deploy new pods in the cluster when the workload is too high. In addition, one keeps the advantage of using VMs so that the software can be run everywhere since, it is containerized in pods, that work like minimal VMs.

2.5 Petri Nets

Petri nets are a type of graph that can be used to model various systems and workflows. The basic structure always consists of places and transitions connected with directed edges. In addition, some tokens can be placed in places and moved from one point to another by transitions. The sum of all occupies of all places in a Petri net represents the state of the net. The relevant definitions and concepts can be found in (Petri 1996).

More relevant for this work are the reference nets which were presented in (Kummer 2002). Thereby, the subform of the colored Petri nets is taken as a starting point and enriched with object-oriented aspects. Instances of nets can be created, which can then be simulated independently. Furthermore, it is possible to simulate net instances directly in other nets. In addition, communicating different nets instances via synchronous channels and a corresponding binding search is possible. Another notable feature is that reference nets can execute program code, which makes them a powerful tool for modeling and complex simulating systems. Further details on the use and definition can be taken from (Kummer, Wienberg, and Duvigneau 2002).

2.6 Renew

Renew is a simulation software that can be used to simulate Petri nets as well as various other types of nets (*Renew - The Reference Net Workshop* 2023). In particular, the focus is on the reference nets that in addition to the usual Petri net formalism also allows Java code to be executed. Hence the name Renew which stands for 'Reference Net Workshop'. Renew was developed by (Kummer, Wienberg, Duvigneau, et al. 2020) and have been in continues development at the University of Hamburg since then. Over the years, new concepts have been integrated into Renew, some of which have changed the software and its architecture significantly.

2.6.1 Architecture

Since the version 2.0, Renew builds on a plugin architecture (Duvigneau 2009). Almost all functions of Renew are provided in different plugins, which can be combined individually and viewed as individual components. Furthermore, module layers extended the architecture with the Renew version 4.0. Plugins with their dependencies are put into separate Java module layers (Janneck 2021). The module layer architecture played an important role in the creation of the Cloud Native Renew Plugin, since this could not use the typical Renew module architecture. More about the creation of the Cloud Native Renew plugin in parallel with the module layer can be found in (Röwekamp, Taube, et al. 2021).

2.6.2 Plugins

As discribed is Renew currently based on a plugin architecture. In this context, a new plugin means that Renew is extended by a new piece of software. Each plugin is seen as an additional independent component, which extends the current functionality. These concepts are described by (Duvigneau 2009).

In this thesis some current RENEW Plugins are more important than others. For this reason, a description of the plugins that play an important role in the integration of Cloud Native Renew follows.

Distribute

The Distribute Plugin was previously used for the distributed simulation of network instances in RENEW. With the plugin, it is possible to simulate a network simulation on several distributed RENEW instances. The Distribute Plugin uses the Java RMI for the communication between the individual RENEW instances. A detailed description of the concept and implementation can be found in the work (Simon 2014). Since the Distribute Plugin uses a now deprecated version of the Java RMI and the usability of the plugin is associated with significant overhead, plans are to replace it by a new resilient Distribute plugin. The Resilient Distribute Plugin was conceptualized and partially implemented in (Senger 2021) and (Röwekamp 2023a).

RenewKube

RENEWKUBE is a plugin for RENEW which allows distributing a simulation to several RENEW instances. Unlike the Distribute Plugin, RENEWKUBE works with minimal effort from the modeler and runs mostly in the background. The distributed system that is constructed by RENEWKUBE works with a master and several worker nodes. These worker nodes each contain a RENEW instance which can be used for simulation. The master node organizes the environment. To achieve this desired behavior, RENEWKUBE uses Docker and Kubernetes. A detailed description of RENEWKUBE can be found in paper (Röwekamp and Moldt 2019).

Cloud Native Renew

Cloud Native Renew is a plugin developed as part of the AOSE 20 project and published at PNSE21 in (Röwekamp, Taube, et al. 2021). It is designed to extend RENEW with an API interface to apply the Resilience, Agility, Operability, and Observability aspect of the Cloud Native paradigm to RENEW. To achieve this, RENEW will be extended with the features such as external simulation control, net and plugin upload, as well as status reporting and health metrics using a modern HTTPS interface.

Some basic functions of the Cloud Native Plugin and how they were elaborated, as well as their relation to the Cloud Native paradigm are rudimentary described in the bachelor thesis (Senger 2021). In addition, conceptual information about the plugin can be found in (Röwekamp, Taube, et al. 2021) and in (Röwekamp 2023a).

2.7 Method of Working and Tools

Various methods and tools were used in this thesis. These methods and tools are presented once, and their functionality is explained.

2.7.1 Agile Software Development

Parts of this thesis were developed in the university project ‘Collaborative Distributed Software Development’. This project is a mandatory elective course at the University of

Hamburg in which students with different degrees work together on the software project RENEW. This course focuses on working in small groups to develop the RENEW software further. Thereby a subform of Scrum named Scrum@Scale is used as a guideline for agile software development in RENEW.

According to (Drumond 2023), Scrum describes an agile working method primarily used for software development. The goal is to support the development team with guidelines and principles. Project management tools are mainly used for ticket organization and management. It is left to the team internally how it organizes and achieves the tasks. Thus, a Scrum team can be considered an independent unit that always takes care of a specific task for a particular time.

To process a tasks, Scrum teams are created, which process previously defined tasks in sprints. It is common that a sprint has a fixed duration of up to 2 weeks and is usually defined by a sprint goal, which should be achieved in that time. The team usually remains fixed so that it can adapt and get used to the respective working methods of the team members.

Three different roles are assigned in the team to distribute and structure the work. According to (Schwaber and Sutherland 2023), the roles are defined as follows. One person takes the role of the Product Owner. This role is responsible for defining a goal for a given sprint. Further, a Product Owner is responsible for ensuring that all the information needed to work on a particular task is available and understandable.

The Scrum Master is a role that one team member performs. The role takes care of implementing the Scrum principles and is generally responsible for managing the team according to the Scrum guidelines. Furthermore, he serves as a link between Product Owner and the developer to create mutual understanding. He also is generally responsible for solving internal and external problems that get in the way of completing the task.

The remaining members of the team take on the role of developers. The developers take over the planning and the final processing of the sprint goal. They make sure that the conditions set by the Product Owner are met. Developers are also responsible for adapting their working methods if the conditions change.

In addition, there are various Scrum events in the development process. The events include the Sprint itself, Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective. Further information about the different roles or specific information about the execution of the Scrum Events can be found in the Scrum Guide (Schwaber and Sutherland 2023).

In Scrum@Scale, several Scrum teams are considered. The established guideline of (Sutherland and Inc 2023) can be used to implement Scrum@Scale. Two new roles are therefore defined. First, there is the Scrum of Scrum Masters role, which takes over the role of the Scrum Master for several teams. In addition, there is the Chief Product Owner, who also carries the tasks of the Product Owner across teams. Also, new Scrum events are introduced, mainly used for cross-team communication. For example, dedicated meetings of the Scrum of Scrum Master with all subordinate Scrum Masters, as well as a meeting of the Chief Product Owner and all subordinate Product Owners. These meetings are then used for cross-team organization, planning, problem-solving, and coordination. Further detailed information about Scrum@Scale can be found directly in the guide (Sutherland and Inc 2023).

Concerning this thesis, the role of a Product Owner was taken by me. In this role, tasks and goals were identified that were relevant to the success and completion of this thesis. These goals were then worked on in a small group. The role of a developer was also taken to collaborate with the team. Here the concept of Scrum was broken a little, but this was necessary to influence the development of this thesis directly. In addition, the own contribution to this work is also guaranteed. In the following chapters, all implementations developed in collaboration with others are marked as such.

2.7.2 Tools

During the development of this thesis, various tools were used for communication, organization, and development.

For the communication, mainly the open-source communication platform Mattermost[4] was used. This has been the central communication point with all participants of the software project, as well as with supervisors. For voice chat, Mattermost was extended with the open-source software BigBlueButton [5], a platform for voice communication. Both were hosted internally by the university.

For version control, software implementation, and for writing this thesis, GitLab [6] was used. Its primary function was using the provided version control and DevOps features.

2.8 Mushu Architecture

The Mushu architecture plays an essential role in this thesis. Therefore, in this section, the architecture is explained fundamentally. Mushu stands for Multi-agent system with Scalability in heterogeneous underlying systems. The architecture was designed by Jan Henrik as part of his dissertation (Röwekamp 2023a) and represents an architectural concept for distributed applications. The Cloud Native Renew Plugin was developed as part of the technical realization of the Mushu architecture with the Renew software.

In the following, the theoretical concept of the Mushu architecture is presented, as well as the technical realization.

2.8.1 Concept

The Mushu architecture can be compared with other architectures such as the ORGAN architecture and the Mulan architecture. What stands out in Mushu is the focus on platform management. In other architectures such as Mulan, the infrastructure and platforms are only very superficially present. Mushu, however, models very precisely the infrastructure of a system, and thus the communication possibilities of individual platforms.

The Mushu architecture lends itself to describing and developing scaling distributed systems. The focus of Mushu is precisely the scaling and distribution of a system. Thus also, the strengths of the Mushu come to the surface. Concurrency, autonomy, platform interaction, and real-world distribution can be modeled very well.

In figure 2.1 is a graphical overview of the Mushu architecture. Particularly noteworthy is the platform management and the platform itself. There are similarities to Mulan, but it is far more complex compared to it. In (Cabac 2010), Mulan was described and can be used as a comparison source to the Mushu architecture. Mulan was first developed in (Rölke 1999). Since Mulan is a very complex architecture, only the work is referred to here, and Mulan is not explained further. The complexity of the platform and platform management is necessary to describe the complex flow of a distributed and scaling system.

[4]Mattermost: https://mattermost.com/
[5]BigBlueButton: https://bigbluebutton.org/
[6]GitLab: https://about.gitlab.com/

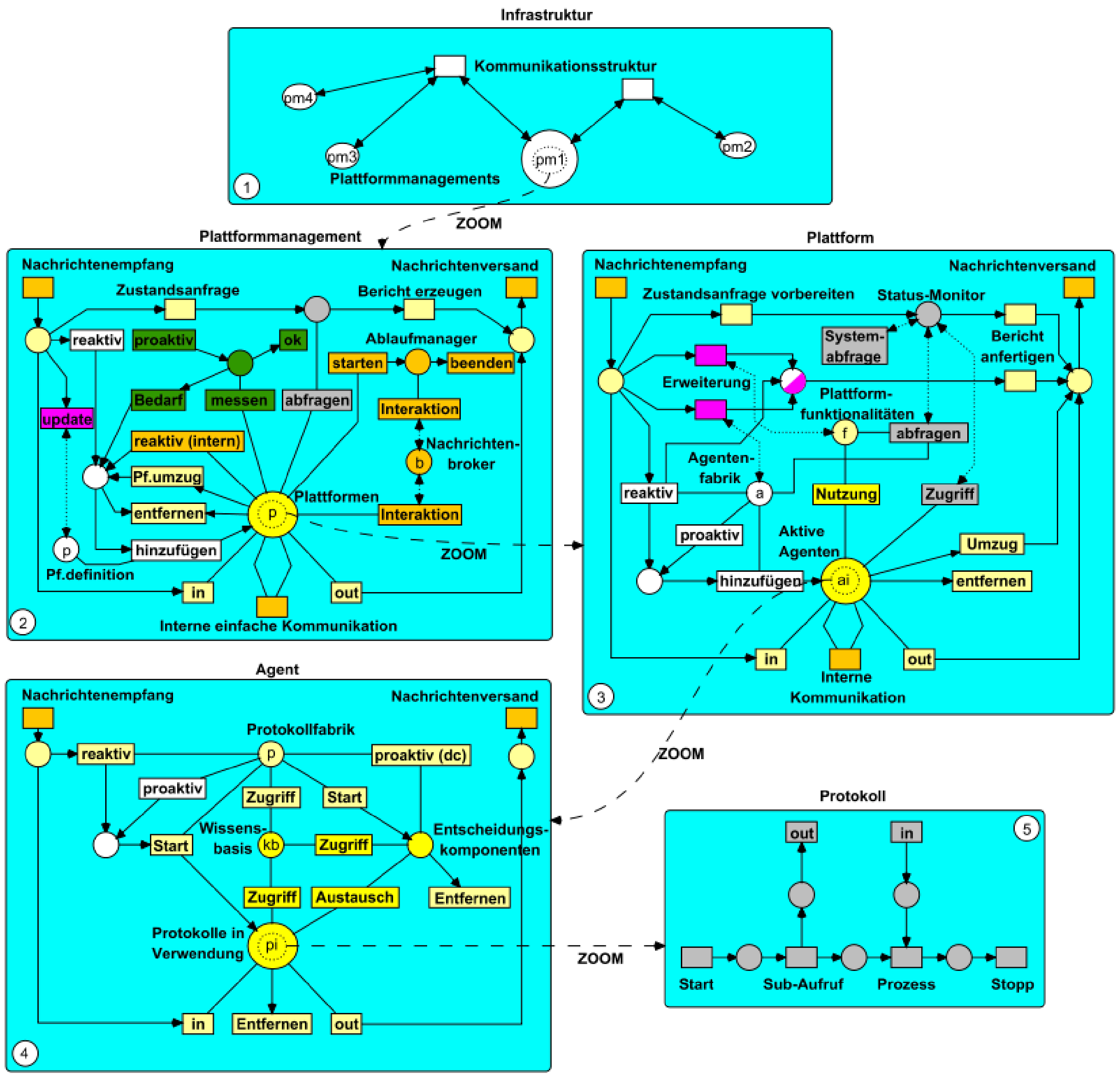


Figure 2.1: Mushu architecture Concept out of (Röwekamp 2023a)

An excellent way to get a better idea of this concept is to take a closer look at the application of this architecture.

2.8.2 Technical Realization

In figure 2.2, Jan Henrik presents in (Röwekamp 2023a) a possible technical realization of the Mushu architecture using the Renew simulation software. Since the Cloud Native Plugin was developed in this technical realization, it will be discussed here. As shown in figure 2.2, each worker node has a Renew instance. The Cloud Native Plugin can also be found in each of these instances. The presence of the plugin in each instance is necessary because the Cloud Native Plugin represents the interface for communication and control. Furthermore, as shown in figure 2.2, each Renew instance also has the Resilient Distribute Plugin developed in (Senger 2021), enabling different types of communication. The monitor node shown in figure 2.2 is currently taken over by Status Monitor Plugin and Spring Boot Admin. Both applications were implemented in parallel with the Cloud Native Plugin. The message brokers shown in figure 2.2 were developed parallel to the Resilient Distribute

Plugin in (Senger 2021). The Cluster Manager is implemented by RenewKube and the RenewKube Manager. Both were developed and described by Jan Henrik in (Röwekamp and Moldt 2019). The CI/CD environment is handled by a GitLab[7] instance, which provides various images deployed to the individual worker nodes. The technical realization presented in figure 2.2 is a goal to be realized. The integration of the Cloud Native Plugin described in this work is to advance this technical realization.

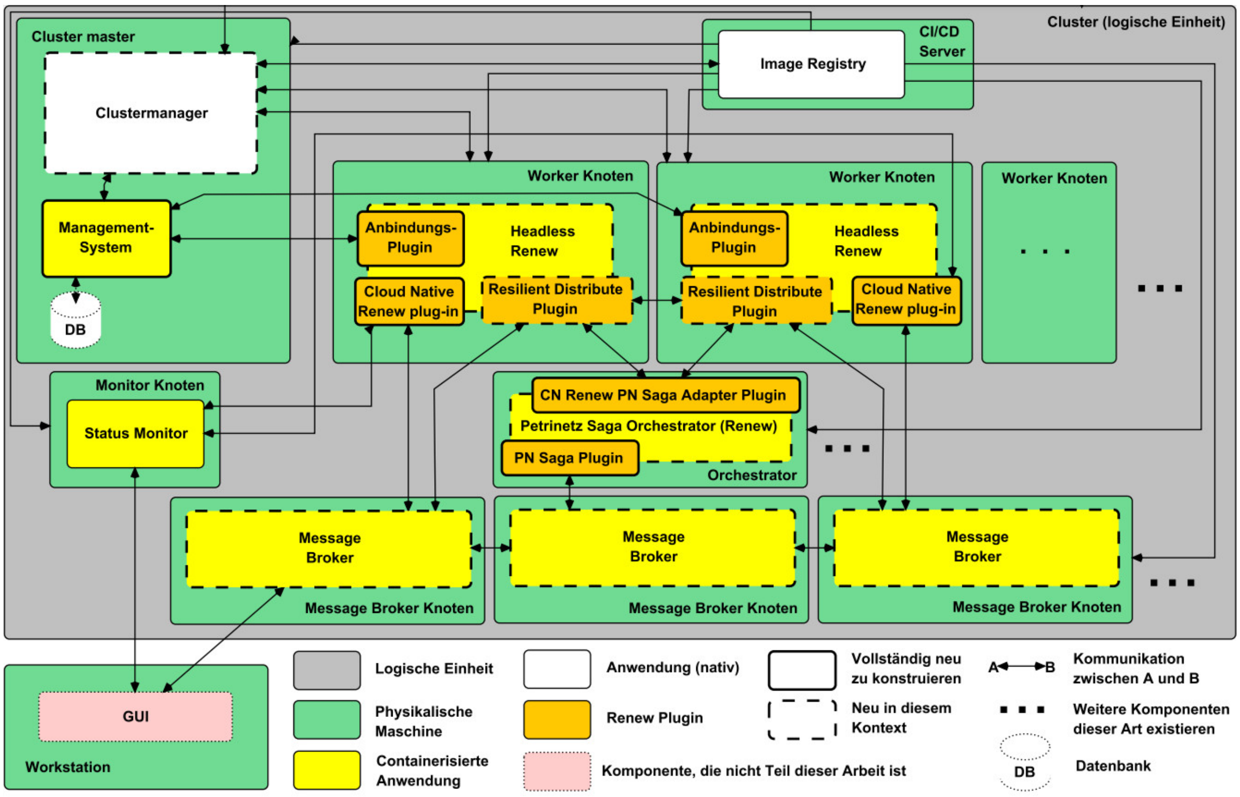


Figure 2.2: Technical realization of the Mushu architecture out of (Röwekamp 2023a)

[7]GitLab Repository for Renew: https://git.informatik.uni-hamburg.de/tgi/Renew

Chapter 3

Requirements Analysis

As mentioned the foundation for this thesis is based on the Mushu architecture developed in (Röwekamp 2023a). In addition, parts of this work originated in the AOSE 20 project and were extended in a paper publication at PNSE21 in (Röwekamp, Taube, et al. 2021). In the following, the general requirements for this thesis are listed. More specific requirements can be found in the respective prototypes.

This thesis aims to integrate the earlier described Cloud Native Renew plugin into the current technical implementation of the Mushu architecture 2.8.2. However, the individual components must be examined in more detail before the integration can be carried out. First, the Cloud Native Renew plugin must be examined in more detail.

Initially, it must determine if the current implementation state of the Cloud Native Renew plugin is suitable for the integration process. Therefore, it must be checked whether the plugin is functional in its current state. In addition, it should be evaluated if the current documentation of the plugin is sufficient. If there are deficits in the documentation, it should be completed accordingly before the plugin is integrated into the larger context of the Mushu architecture. Furthermore, if there are different versions of the Cloud Native Plugin, it should also be decided which version of the plugin is to be used for the integration.

Requirement R1:
The current state of implementation and documentation of the Cloud Native Renew Plugin should be reviewed.

Requirement R2:
It should be decided which version of the Cloud Native Renew Plugin is the base version for the integration process.

The next step is to focus the Mushu architecture. Here is to determine if the current state of technical realization of the Mushu architecture and its subcomponents are suitable for the integration of the Cloud Native Renew Plugin. Adjustments should then be made to the components which are not suitable. The goal is to create a starting point into which the Cloud Native Renew plugin can be integrated.

Requirement R3:
It should be determined which components of the current technical implementation of the Mushu architecture need to be adjusted so that an integration of the Cloud Native Renew Plugin is possible.

Requirement R4:
If components of the current technical realization of the Mushu architecture need to be adapted, they should be adapted accordingly.

For the development and later validation of the integration, an environment is needed in which the technical realization of the Mushu is implemented. Furthermore, a production environment is desirable to come as close as possible to real-world requirements. For this purpose, a computer cluster with appropriate documentation was provided in the context of (Röwekamp 2023a) at the university. For this thesis, it must be determined in which condition this environment is, whether the documentation is sufficient for its use, and which restrictions exist for the usage of the provided environment should be examined.

Requirement R5:
The technical realization of the Mushu architecture on the provided cluster should be determined in its current state to see if it is suitable for the Cloud Native Renew Integration.

Requirement R6:
Suppose the cluster is not in a functional state. In that case, it must be restored to a state suitable as a development platform and later as a production environment for validating the integration. If the documentation provided for the cluster environment is insufficient, it should also be supplemented to make future use of the environment more accessible.

After preparing the components for integration and creating an environment for testing, the actual integration from the Cloud Native Renew Plugin should occur. The process involves integrating the plugin into the latest version of the technical implementation of the Mushu architecture. All steps of the integration and subsequent adjustments should be carried out. In addition, which steps were necessary for the integration into the environment should be documented to create a guide for subsequent integrations.

Requirement R7:
The actual integration from the Cloud Native Renew Plugins into the latest version of the Mushu architecture's technical realization must occur.

Requirement R8:
The steps necessary for integrating the plugin into the environment should be documented for subsequent integrations.

After the successful integration of the Cloud Native Renew Plugin, the usability of the plugin in a cluster environment should be evaluated. As a developer of reference nets for the cluster, it should be as easy as possible to use provided functions of the plugin. Therefore, it is necessary to evaluate the best way to make the provided functions of Cloud Native Renew usable for manipulating remote Renew instances for the reference net developer. If the plugin is extended in the process, the extension should be documented, and appropriate instructions for the usage should also be provided.

Requirement R9:
It should be evaluated how the functions of the Cloud Native Renew Plugin can be used in the cluster environment. The results should then be documented accordingly.

After the plugin has been integrated into the environment and its usability in the production environment has been evaluated, the success of the integration should be validated. For this purpose, an example use case should be created that shows how to interact with the functions of the Cloud Native Renew Plugin in the cluster and how the plugin interacts in the product environment. For the use case, it is logical to use an example from the development of RenewKube as a basis since this has already worked in the production environment and would demonstrate the extension nicely.

Requirement R10:
An example use case demonstrating the interaction with the plugin and the production environment should be created to validate a successful integration.

Chapter 4

An Overview of the Cloud Native Renew Plugin

This chapter is based on several previous papers and is intended to be a collection of the current state of the Cloud Native Plugin. Implementations that others have helped to develop or developed on their own, are marked as such.

The first prototype presented here summarizes in more detail what the Cloud Native Plugin includes and to what extent they have been implemented and validated. The functions were developed in the context by (Röwekamp, Taube, et al. 2021) and later validated and partial documented by (Senger 2021). On this basis, the prototype is intended to serve as a starting point for integration in the distributed context, which will then be implemented in the later prototypes.

4.1 Requirements

The Cloud Native Renew Plugin must be in a state where it can be integrated into a distributed environment. For this purpose, it should be verified that the individual components work and are well documented. For that, an overview of the plugin should be provided. If it is not yet the case, all missing parts should be documented. Should the plugin not work in the current state, it should be brought into a state wich is suitable for the integration.

4.2 Specification

The documentation of software projects is usually a significant challenge. The reason for this is that over a long period, various people work on the same project with their own perception of documentation. Even though there are guidelines, there are severe discrepancies in the quality and completeness of it.

The same difficulty exists with the Cloud Native Renew Plugin. Therefore, the various documentation places about this plugin are considered and checked for completeness. Furthermore, missing or incomplete documentation will be added. Extensions to the functionality of the plugin will be considered in later chapters.

Feature	Cloud Native Aspekt	Documentation	Implementation
Log Functionality	Observability	(Senger 2021)	AOSE 20/21
Health Metrics	Observability	(Senger 2021)	AOSE 20/21
Simulation Control	Operability / Resilience	(Senger 2021)	AOSE 20/21
Upload Nets	Operability / Resilience	(Senger 2021)	AOSE 20/21
API Documentation	Observability	(Senger 2021)	AOSE 20/21
Plugin Upload	Operability	(Röwekamp, Taube, et al. 2021)[1]	PNSE21
Plugin Loading	Operability	(Röwekamp, Taube, et al. 2021)[2]	PNSE21

Table 4.1: Overview of the current features of the Cloud Native Plugin and where to find the documentation about it.

4.3 Analysis of the Current State of Cloud Native Renew

As already mentioned in 2.6.2, the individual functionalities of the Cloud Native Plugin are discussed, and placed in the context of the Cloud Native paradigm in (Senger 2021). The thesis covers the following aspects of the plugin: Java Spring as the basis of the plugin, provision of logs via HTTP, status monitoring via HTTP, start/stop/stop operations of simulations via HTTP, upload of nets via HTTP, and API documentation. Furthermore, (Senger 2021) focuses more specifically on the Distribute plugin and the resulting Resilient Distribute plugin in the context of Cloud Native.

A central function of the plugin is to load plugins at runtime into a Renew instance, which only has been partly described in (Röwekamp, Taube, et al. 2021). In terms of integration, this function is interesting because such functionality will allow to heterogeneously initialize different Renew instances at runtime, and add special functionalities to them if needed. In the overall view of the Cloud Native schema, this functionality fulfills the operability condition. This feature, like the others, is implemented via an HTTP interface.

In figure 4.1 all current functions of the Cloud Native Renew Plugin are listed. Each function is also assigned an aspect of cloud nativity. In addition, you can find a source of documentation for each function and the project in which this function was implemented.

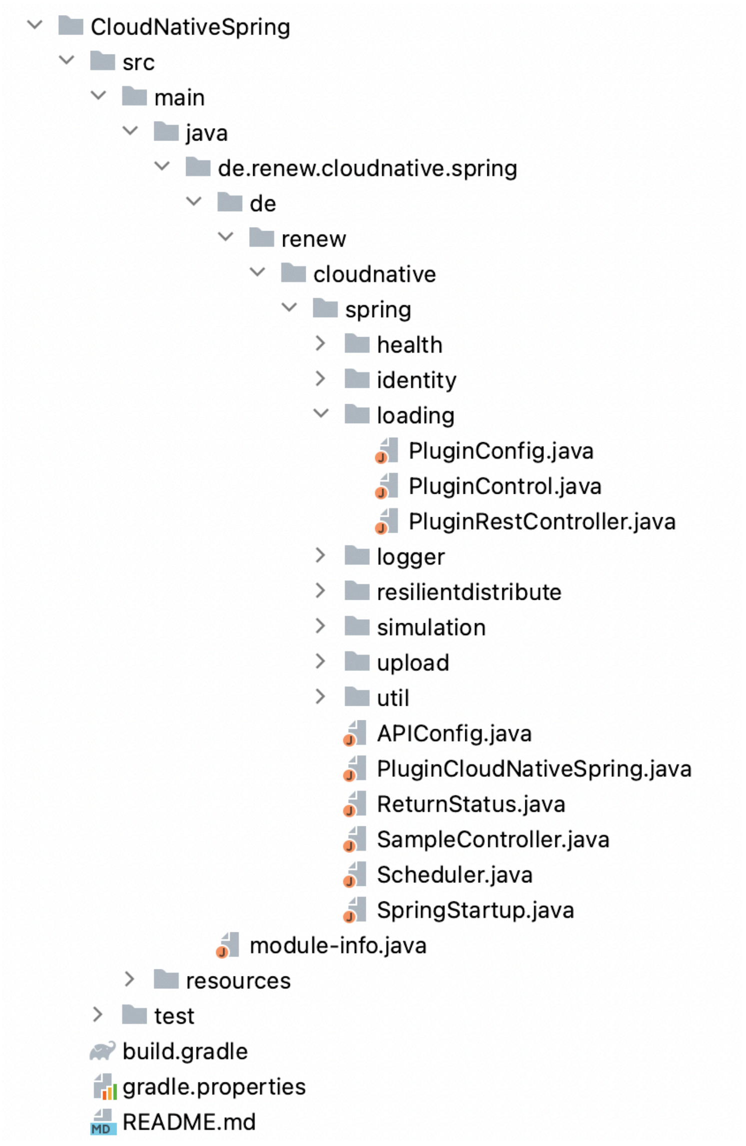
Since the documentation in Senger 2021 is sufficient for the most functionalities, those are not documented again in this thesis. The last two functionalities in figure 4.1 have not yet been appropriately documented. There is a publication (Röwekamp, Taube, et al. 2021); however, the functionality was only partially treated there. Otherwise, there is also a report of the AOSE 21 project (Marvin Taube 2021), describing the functions. Nevertheless, the documentation is only rudimentary since the functions were developed outside the project. Therefore follows for these two functions, 'Plugin Upload' and 'Plugin Loading' a detailed documentation later in this chapter

4.4 File Structure

Since no work has dealt with the unique file structure of the Cloud Native Renew Plugin, this section will document it. The file structure is interesting because there is a certain standard in the Java Spring context of how to structure a project.

The thesis (Janneck 2021) has already worked out how the general file structure for a Renew plugin should be. Most plugins follow precisely the scheme that was worked out in (Janneck 2021). This structure can also be found in the Cloud Native Renew Plugin. In

[1]This function will also be documented by this thesis later in 4.5
[2]This function will also be documented by this thesis later in 4.5



```
CloudNativeSpring
    src
        main
            java
                de.renew.cloudnative.spring
                    de
                        renew
                            cloudnative
                                spring
                                    health
                                    identity
                                    loading
                                        PluginConfig.java
                                        PluginControl.java
                                        PluginRestController.java
                                    logger
                                    resilientdistribute
                                    simulation
                                    upload
                                    util
                                    APIConfig.java
                                    PluginCloudNativeSpring.java
                                    ReturnStatus.java
                                    SampleController.java
                                    Scheduler.java
                                    SpringStartup.java
                    module-info.java
            resources
        test
    build.gradle
    gradle.properties
    README.md
```

Figure 4.1: File Structure of the Cloud Native Plugin

the plugin, however, there is another structure in the file system that comes from the Java Spring standards.

In figure 4.1 is a picture of the file system of the Cloud Native Renew Plugin. Not all folders are expanded because the structure of some packages are almost the same. On the level of the main folder, 'de.Renew.cloudnative.spring' are, on the one hand, the main classes, which are used in the whole plugin, as well as the start classes of the plugin. The class 'PluginCloudNativeSpring' is the plugin's main class and is executed first. SpringStartup is the start class of the Spring context and is also executed at the start. The class 'ReturnStatus' provides functions to make a response easier for a given request and is used in various parts of the plugin. The class 'APIConfig' configure the SwaggerUI, and the 'Scheduler' is used to create the health metrics. As the last class of the layer, there is a 'SampleController'. This class has a test endpoint that can be used as a template for other simple endpoints.

There are now different packages on the plugin's main level. The util and identity package contain internal functionalities of the plugin. The resilient Distribute package was developed in (Senger 2021). All other packages contain functionality with an HTTP endpoint. The basic structure of the packages is the same and consists of three classes, as shown exemplary in the 'loader' package in figure 4.1. All packages with an HTTP endpoint contain similar classes like ...Config, ...Control, ...RestController. The 'PluginConfig' is the connection to other RENEW classes from other plugins. Here the references from the other plugins are packed into so-called beans and made accessible for Spring though dependency injection. The 'PluginControl' class represents the main class of that functionality. In the example case, the loading of plugins is implemented in this class. In the last class, 'PluginRestController', the HTTP endpoints are defined, and the request is handled. The separation of request handling and actual implementation is common practice in Spring. Therefore, if other functions are planned, this scheme should be followed.

4.5 Additional Documentation of the Plugin

The upload function as well as the loading of a plugin at runtime have not yet been thoroughly documented. However both are significant functionalities for the later use in MUSHU. Therefore, the following sections will take a closer look at these functions. First, the plugin upload will be looked at and after that, the loading of plugins is documented. Both features were implemented as part of the publication (Röwekamp, Taube, et al. 2021) in collaboration with the authors.

4.5.1 Uploading of Plugins

The upload functionality was created to fulfill the operability aspect. The background of the implementation was to initialize new plugins into a RENEW instance at runtime. The implementation with an HTTP endpoint meets these requirements. In the following, it is documented how the upload of a plugin works. In addition, the internal process is explained, and details of the implementation are considered.

Upload Endpoint

An HTTP endpoint '/upload/plugin' is provided for the plugin upload. The endpoint takes three parameters.

The first parameter 'pluginJarFile' should contain the path to the plugin that is to be uploaded. The plugin should be uploaded as a .jar file and meet a RENEW plugin's requirements. This means the plugin needs to contain a ‘module-info.class’ and a ‘plugin.cfg’ file. If these files are not included in the .jar file, it will not be accepted, and the upload will fail.

The second parameter ‘pluginName’ can be used to give the plugin its file name. Please note that this parameter only determines the name of the plugin file in the file system. If the plugin is loaded, the plugin name from the plugin.cfg is listed in the running system. Nevertheless, the parameter is needed to identify the plugin in the file system to load it after the upload.

The third parameter ‘override’ is a boolean value and describes whether the plugin should be overwritten if a plugin with the same name already exists in the file system. The value of this parameter is set to ‘false‘ by default.

Internal Functionality

When uploading a plugin, it is validated in the fist place. It is checked if the ‘module-info.class’ and ‘plugin.cfg’ are available. It is important to mentioned that the content of these files is not checked, which makes it is possible to upload faulty plugins.

When validating the plugins, there is also no verification of authenticity. Should the convention be followed, plugins of any kind can be uploaded. The lack of authenticity represents a security vulnerability if access to the upload function is not regulated. Should the system’s security requirements be in the foreground, one should rework the validation aspect.

If the validation is successful, the uploaded plugin is stored in the file system of the remote Renew host. The uploaded plugins are added to the regular‘dist/plugins’ plugin folder. Each plugin is additionally marked with the prefix ‘upload’. The prefix is created to distinguish base and uploaded plugins.

The loading of the plugin does not happen during the upload. To load a uploaded Plugin, the following endpoint must be considered

4.5.2 Loading of Plugins

Loading plugins into the active Renew context also fulfills the operability aspect of Cloud Native, and is a key component behind the functionality of Cloud Native. In the following, it will be documented how to use the endpoint and what needs to be considered, as well explaining which elements were used for the internal implementation.

Load Endpoint

Like the upload functionality, there is also an endpoint ‘/loadPlugin’ for loading the plugin into Renew. This endpoint expects a parameter ‘pluginName’ of type string. In this parameter, the filename of the plugin in the file system of Renew must be used.

The intended workflow is to upload a plugin first. Thereby also, a parameter ‘pluginName’ is used. The parameter should be identical to the parameter of the load plugin. As mentioned before, a prefix is assigned when uploading. If it is not present, this prefix will be appended to the ‘pluginName’ parameter. This behavior allows using the same parameters even if the internal name is different.

Internal Functionality

Cloud Native Renew uses the provided function of the Loader Plugin for the process of loading a plugin at runtime. The reason for this function was that the desired functionality had already been implemented and worked perfectly. The Cloud Native Plugin hooks into the functionality of the Loader Plugin and thus executes the loading of the plugin.

The decision for this variant was made because, first the Loader Plugin has already been completed, and the functions have been tested. Therefore it can be assumed that the provided functionality works and there is no need to develop redundant behavior. Second, the Loader Plugin is used in every Renew instance. So it is possible to use the functionality of the Loader without having multiple dependencies in a Renew instance. This also guarantees that loading is possible even if the actual function is not in Cloud Native Renew itself.

4.6 Versions of the Cloud Native Plugin

Since the development of the Cloud Native Plugin, there has been constant work on it. Therefore, many different versions of the plugin have been created, each at a different stage of development. For integrating the Cloud Native plugin later in Chapter 7, it is essential to use a version that works without problems but still provides the most functionality.

At this time, there are two major versions of the plugin and a few minor subversions that reside in different branches in the Renew Git repository. The two major versions are the base version and the version with the extension of the communication via a message broker. The base version of the plugin is in the modular master branch of the Renew repository. This is the direct result of the published version in (Röwekamp, Taube, et al. 2021). This version was also used as the basis for the documentation created in this chapter.

The other notable version of the plugin is the one created as part of (Senger 2021), which enables communication via resilient Distribute and Apache Kafka. The results are also already available on the branch modular master of the Renew repository.

The first mentioned version would be more suitable for integrating the plugin than the one with resilient Distribute. This is because changing the communication medium adds another level of complexity to the integration. Furthermore, the realization of the communication via message broker is not a function that is directly related to the Cloud Native aspects. Therefore, the use of the base plugin version with the functions described earlier in this chapter was chosen as a starting point for the integration.

4.7 Evaluation

In chapter 3, requirements were established, which should be fulfilled within the scope of this chapter. The main question was whether the plugin is in a state in which it can be integrated into a distributed environment. For this purpose, an overview of the functions was first created. The overview includes where and when this function was implemented and where the respective documentation can be found. Creating an overview of the plugin was also one of the requirements for this chapter. In this context, the functions were also tested again. The plugin works so far and is ready for integration.

Another requirement was that there is a sufficient documentation for the individual components of the plugin. For this purpose, references were made to already written documentation, and the upload and loading functions of the plugin were documented in more detail. In addition, an overview of the file structure was created, which had not yet existed in this form.

4.8 General Evaluation

The prototype described in this chapter deals with requirement R1 and R2 developed in 3. Considering the overall context of this thesis, this chapter fulfills the essential role of documentation. Due to the plugin's development over a long period, the documentation of the plugin fell short. Not all functions were documented in this chapter, because sufficient documentation already exists for some parts. These documentations are listed in this chapter, making the documentation easier to find. In addition, overviews of structure and functionalities were created. These overviews are also part of the documentation of the plugin. Further the decision which version of the plugin to take as a baseline for the integration was outlined in 4.6. Thus, requirements R1 and R2 are fulfilled.

Chapter 5

Preparation of Mushu Components

In preparation for Jan Henrik's disputation, a task force was created to help him implement as much of the content from his disputation as possible. This working group consisted of Jan Henrik, Laif-Oke Clasen, Sven Willrod, and me.

Many aspects in the whole context of the Mushu architecture were looked at during this working period. This time was mainly spent updating and merging different plugins into the current Renew version. The work done was also necessary for the final integration of the Cloud Native Plugin.

Since the content of this thesis is directly related to Jan Henrik's dissertation, essential aspects of this thesis were also worked on during this working period. Because of that, the work done of the task force is documented in this chapter. However, this is done with a focus set on the elements relevant to this thesis.

5.1 Requirements

The requirements of this chapter can be divided into two sections. First, requirements can be set up for this thesis. These conditions drive the development forward and add value to it.

In addition, there are requirements for the work phase. These requirements were established at the beginning of the preparation and were a guideline to show what still needs to be done.

Since the work done at the preparation time is not documented anywhere, this chapter will serve mainly as the documentation. All relevant aspects for the integration of Cloud Native Renew will be documented. Furthermore, the requirement for the completeness of the work is to be checked.

The work phase itself had the goal of preparing the disputation of Jan Henrik. For this reason, the goal was to create a complete executable system of the Mushu architecture. The completion of that goal would have included the integration of the Cloud Native Plugin. The requirement for the work was to mainly to update all components to the current Renew conventions. In addition, it was the goal that still existing bugs are eliminated and missing documentation is supplemented.

5.2 Specification

As part of the preparation for the disputation, various people worked on the technical realization of the Mushu architecture. A large part of it was the revision of various plugins. These included RenewKube, RenewKube Manager, Distribute, Cloud Native, Distributed Analysis, and Momoc. Most of the rework involved implementing the Renew criteria and bug fixes.

In addition, a major topic was the documentation of the individual components, which can be found in the respective ReadMe files in the Git repository[1].

The following sections deal with the documentation of the work done. Difficulties and obstacles are discussed, and their solutions are presented. In addition, parts that still need to be finished are outlined.

5.3 Performed Work on Renew Plugins

The working period is about two weeks, during which four people were occupied implementing the technical realization of the Mushu architecture for the disputation of Jan Henrik. The focus was primarily on completing various plugins from Renew and other components that are necessary for the Mushu architecture.

All plugins that were worked on concretely are listed in this section. An overview of what has been done on each plugin is given and what additional value these bring to technical realization of the Mushu architecture is outlined.

5.3.1 Distribute

We started with the work on the Distribute Plugin. In the best case, this plugin should be replaced by the Resilient Distribute Plugin. However, the Resilient Distribute Plugin has not yet been completed.

At the time of writing, only simple communication via the Resilient Distribute Plugin is possible. Also, it has not yet been tested with Cloud Native Plugin in a distributed environment.

The main work done to the Distribute Plugin was to check if the current Renew conventions were respected. First of all, the file structure had to be adapted to the current file structure used by all Renew plugins, which was developed in the context of (Janneck 2021).

After the file structure was adapted, some provided sample nets were tested. The verification was necessary because it was unknown whether they still worked with the current version of the plugin. After the successful verification, a new Gradle build task was created. The task builds a minimal Renew version with the Distribute Plugin. The build is suitable for testing the software and was not difficult to implement due to the Renew architecture.

The next step was to update the ReadMe file of the plugin to the current standard conversions for ReadMe files for Renew. Since the ReadMe of the Distribute Plugin was very outdated, it was completely created from scratch again. The new ReadMe now also serves as the main starting point for the plugin's documentation. In the ReadMe are also all relevant other documentation linked.

The Distribute Plugin is also still built with a build.xml file at the time. Since this variant for building plugins with Ant no longer corresponds to the current Renew standards, this should be changed. Since the file was very complex and too many Ant tasks had to be changed, it was decided to postpone this due to time constraints.

Otherwise, a bug was fixed where different dependencies were present in 'module-info', 'build.gradle', and 'plugin.cfg'. The dependencies on other modules and plugins should be uniform in those three files. Therefore, they were analyzed and adjusted accordingly.

[1]Renew Repository: `https://git.informatik.uni-hamburg.de/tgi/renew`

Finally, a merge request to the modular/Distributed-Analysis Branch[2] was created. This branch of RENEW served for this work time as the main branch on which all results were collected.

5.3.2 RenewKube

Most of the work on the RENEWKUBE plugin and the RENEWKUBE Manager was done by Jan Henrik and Laif-Oke Clasen. In this subsection, is a summary given of what work was done on the two components. Both of which play an essential role or the integration the Cloud Native Plugin and are, therefore, relevant for this thesis.

On the RENEWKUBE plugin itself, the scripts and Dockerfiles were updated to the current RENEW 4.0 version. The main goal was the modularisation of the RENEWKUBE plugin. This work was time-consuming as these plugins needed to be tested with VMs, and no VM images have yet been deployed for these versions. A detailed list of what has been done to the plugin can be found in the merge request of this work[3].

RenewKube Manager

Some work has also been done on the RENEWKUBE Manager. Even if it is proprietary software, it was tried to adapt to the RENEW conventions.

However, modularization was difficult. Since the software is fundamentally based on Java Spring, it was impossible to apply the module conventions of RENEW to it. The reason was that the used modules must be set to the keyword ‘opens’ for the Java Spring context to work. Opening all modules means that all dependencies are on the same level. This centralization is suitable for Spring because Spring can always reference everything, but this contradicts the module layers of RENEW.

Nevertheless, adjustments were made to the software to implement the RENEW conventions as best as possible. This includes the general RENEW conventions, adjustments to the structure, and the creation of the ReadMe.

One of the most significant enhancements to RENEWKUBE Manager was the ability to assign a port to individual RENEW instances. Each RENEW instance runs in a Docker container and is accessible via Kubernetes. However, in order to be able to run networks in a distributed manner, the respective instances need to know which internal port they are assigned. This was made possible by a simple query at startup. Each time a RENEW instance is started, the RENEWKUBE Manager is addressed, and the instance receives its assigned port.

In addition, other changes and improvements were made to the software. A detailed list of the work can be found in the earlier mentioned corresponding Jira epic. The implementation of the individual elements can be found in the repository of the RENEWKUBE Manager.

5.3.3 Cloud Native

The Cloud Native Plugin was already in a state that complied with the RENEW 4.0 conventions. A merge request was checked and executed after resolving merge conflicts.

In addition, a Gradle build task was created that disabled the elements of the Resilient Distribute. This was because these elements led to considerably higher startup times.

[2]Distributed Simulation Branch: `https://git.informatik.uni-hamburg.de/tgi/Renew/-/tree/modular/distributed-simulation`

[3]Modularisation of RENEWKUBE: `https://git.informatik.uni-hamburg.de/tgi/renew/-/merge_requests/297`

Especially when the Cloud Native Plugin is not in the right environment. This behavior happens when starting with the Resilient Distribute elements, and try to connect via Apache Kafka. The connection is not possible if there is no target of the connection. Therefore the plugin was in a loop, which caused the higher startup time.

After merging the plugin, there was another bug called jsr305 bug. This bug had extended to the modular master branch of Renew, which is one of the main branches of Renew.

The jsr305 bug was a bug in the way how Java Spring load the required modules. This causes the same additional modules to be loaded in different dependencies, which leads to duplications and causes a crash at the startup. The two modules that were affected were the ‘jsr305’ and the ‘javax.annotation’ module. The solution was to remove the jsr305 module in the required ‘spring-boot-admin-starter-client’ and ‘guava’ module.

Solving the jsr305 bug resulted in new bugs in the Cloud Native modules. Two libraries were missing after that, ‘javax.enterprise:cdi-api:1.2’ and ‘io.projectreactor.tools:blockhound:1.0.6.RELEASE’ which had to be added back in Cloud Native build.gradle manually.

In addition, the module names of the log4j modules had to be adapted to new versions since, several versions of log4j were present. The last change was the name of the Status Collector Plugin to adapted to the given conventions.

5.3.4 Additional Work

Since the other topics are not complex enough to get their sections but were important for the progress of the implementation of the technical realization of the Mushu architecture, they are listed here. For the exact assignment of it, one can look into the corresponding Jira Epic[4].

First, we have the creation of a new KeepAlive plugin. Creating a new plugin was necessary because the old KeepAlive plugin from Renew requires a console gui interface, which cannot be used in the Docker context. Therefore, a plugin should be created that provides the same functionality without requiring a gui. In addition, this plugin should then also be controllable via Java property just like the old console plugin.

Another important point was the addition of comments in the various plugins and modules used in the technical implementation of the Mushu architecture. These were primarily done in the modules and plugins around RenewKube and the RenewKube Manager. However, the comments of the individual elements are essential for the subsequent generation and to create a better overall understanding.

The ReadMe files were also supplemented in many subcomponents in the same course. These include the plugins: Console, ConsoleGuiInteraction, RenewKube, RenewKube Gui, KeepRunning, DistributedAnalysis, and DistributedAnalysisGui.

5.4 Evaluation

The evaluation is divided into two parts in this chapter. First, the extent to which the requirements for the work phase have been met is considered. There, it is evaluated how the work phase proceeded and how successful it was.

The second part evaluates to what extent this chapter had an added value for this thesis. The content of the work phase is considered, as well as the documentation.

[4]Jira Epic: `https://tgipm.informatik.uni-hamburg.de/jira/browse/AOSE21AVS-521`

5.4.1 Working Phase

Essential work, like merge requests and documentation, was done in this working phase. In addition, bugs were fixed, and functions were extended. The work was related to many different Renew plugins that are relevant to the Mushu architecture.

Nevertheless, many tasks are still not finished. The biggest open topic is the upgrade of the Distribute Analysis Plugin. In addition, it has not been possible to set up a working version of the technical realization of the Mushu architecture. This is mainly due to the time limitation of the disputation deadline. The next prototype in chapter 6 will discuss this in more detail.

Finally, it can be determined that the work phase was an overall success. A sold foundation of rudimentary task were done, and the main upgrade to the components to use them in a Renew 4.0 context were performed. Even if still some tasks are open, many goals were reached, and many tasks were worked on. A detailed list of the work done and the tasks still open can be found in the earlier mentioned Jira Epic[5].

5.4.2 General Evaluation

The prototype presented in this chapter deals with the established requirements R3 and R4 from chapter 3. The various components of the technical realization of the Mushu architecture should have been considered, and it should have been determined whether these need to be adapted for integration. In addition, if changes to the components for integrating the Cloud Native Plugin are necessary, these changes should be accomplished.

The main work and changes were made directly on various plugins of Renew. Section 5.3 of this chapter presents a selected choice to the changes made to the plugins. The listing of the plugins mentioned here serves as an overview of the Renew plugins that had to be adapted to integrate the Cloud Native Plugin. This list fulfills the requirement R3 to determine the components that must be changed.

Furthermore, in the same section 5.3, the changes made to the plugins are described in more detail. There it is described, what the individual problem was, and how this was solved. Requirement R4 is also fulfilled since the plugins were adapted accordingly to make integrating the Cloud Native Plugin possible. Thus, both requirements from 3 have been processed and fulfilled in this prototype.

[5]Jira Epic: `https://tgipm.informatik.uni-hamburg.de/jira/browse/AOSE21AVS-521`

Chapter 6

Setup a Distributed Production Environment

The prototype presented in this chapter deals with the reestablishment of the distributed production environment, in which the current version of the technical realization of the Mushu architecture's is to be implemented. The reason for creating a production environment is that it comes very close real-world conditions. The reestablishment of the environment is necessary because a system-wide update has removed the previous state of the environment.

The prototype's main points are divided into three parts. First, a capture of the previously implemented state is made. It follows a description of the process of what was reestablished and which elements were adapted in the process. The new current state is presented at the end of the prototype, and the reestablishing process is evaluated.

The prototype described in this chapter was created in cooperation with the AOSE project team.

6.1 Requirements

The Mushu architecture is a complex system designed to consist of several computers. Therefore, a cluster is needed to test the architecture and develop the technical realization. Such a cluster can be created in the form of a production environment. The advantage is that such an environment with real computers comes very close to real-world conditions and is ideal for testing.

Thus, developing and implementing the technical realization is essential to have an environment where progress can be tested accordingly. In addition, there must be appropriate documentation for the environment that explains how to interact with it and how to maintain the environment if problems with it arise.

6.2 Specification

First, it should be documented in which sate the implementation and the corresponding documentation was present at the university. This state of the cluster should be used as a standard to evaluate later whether the restoration covers the same scope as the status that prevailed before.

When restoring the cluster, the particularities of the different components should be documented accordingly. In addition, reference should be made to the existing documentation. Contradictions or missing elements in the documentation should be annotated accordingly to facilitate a renewed setting up of the environment.

The goal here is to obtain a state of the environment in which the integration of the Cloud Native Plugin can be carried out. For this purpose, the results should be presented accordingly, and their limitations should be documented, as well as the extension made to it.

6.3 Starting Point of the Production Environment

At the beginning of this thesis, a part of the technical realization of the Mushu architecture was already implemented and made available in a computer cluster. In addition, instructions for the technologies used and information for installing the required components were created in a Confluence Space Jan Henrik Röwekamp 2023. This provision of the implementation documentation was created in the context of (Röwekamp 2023a).

In figure 6.1, the state of the implementation of the technical realization of Mushu is to be seen as it was at the beginning of this thesis. One can recognize that it concerns thereby a part of the exemplary implementation mentioned in section 2.8.2. The most significant differences are the communication, CI/CD server, and status monitor components, as well as the absence of the Cloud Native Plugin in the worker node.

The communication in this variant of the technical realization was implemented with the Distribute Plugin from Renew. This plugin is used to enable to communication between different reference net simulations. Consequently, the Resilient Distribute Plugin, Message Broker, and Orchestrator shown in figure 2.2 are not used in this variant to implement the communication between nets. Work has already been done on communication through Apache Kafka as part of (Senger 2021), but these results have not been integrated into the cluster environment.

The CI/CD server and the status monitor components are also not implemented in this variant. Nevertheless, both components will be handled in this chapter as part of the environment recovery process and thus integrated into this variant. This also applies to the Cloud Native Plugin, where the integration is considered in the next prototype 7 in more detail.

A collection of documentation was also created for the technical implementation. This includes an overview of technologies used, the status of development, a description of components, and instructions for reinstalling the environment. The documentation provided is extensive, but it turned out to be insufficient to install the environment from scratch. Therefore, it will be enriched with new information gathered from the recovery process.

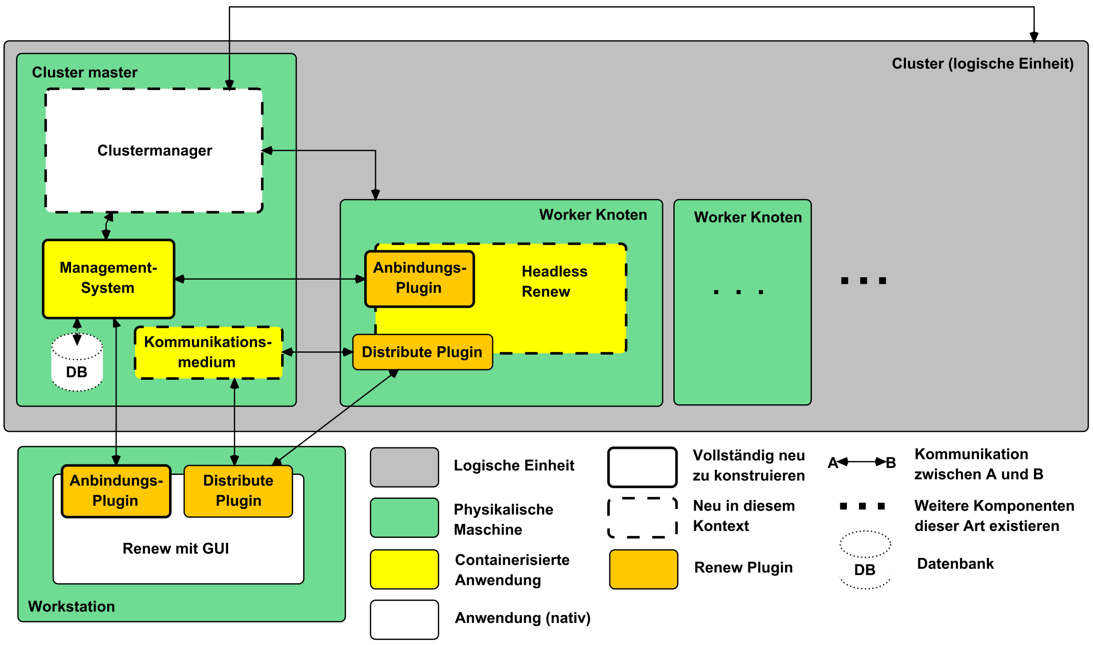


Figure 6.1: The initial basis of the technical realization of the Mushu architecture out of (Röwekamp 2023a).

6.4 Recovery of the Production Environment

As already described, there was a functioning cluster at the university. However, this was removed by a system update and a corresponding complete reset of the computers. The system update was mate to the Ubuntu version 22.04.1 LTS.

The scenario now was to use the provided instructions for the cluster to restore it. However, several problems arose in the process. First, many steps to install the base software were inaccurate or invalid due to the system update. In addition, since there was no prior knowledge of the technologies, the recovery took more work.

For the recovery process, the provided instructions were used to restore the clusters step by step. In the process, many manual sections had to be adapted or supplemented. In the following, the individual technologies and components are described in detail. Specific approaches and difficulties are documented in order to make it easier to set up the cluster again. It should be noted that admin permissions are required on the computers to perform the recovery.

6.4.1 Base Software

Before considering the recovery from the technical realization of Mushu with Renew, the base software must be installed on the systems. The three relevant ones are Java, Kubernetes, and Docker. Java could be installed simply over the package manager of Linux in the necessary version. For Docker and Kubernetes, several steps were necessary, which will be described below.

Kubernetes

For Kubernetes, the official documentation (Kubernetes 2023b) should first be used for the installation. Ensuring that kubeadm is installed is crucial, as this is required for managing Kubernetes.

In the documentation of kubeadm is a reference made to another software: cri-docked (Mirantis 2023). If Docker should work with Kubernetes in conjunction, the cri-docked software must be installed. Corresponding references were not present in the provided instructions. However, extensive testing determined that this software is required to obtain the desired functions. In addition, it is necessary for some commands to specify which cri-socket is to be used. When a prompt comes up while setting up the cluster, the following argument 6.4.1 must be added to the command.

```
$ ... --cri-socket=unix:///var/run/cri-dockerd.sock
```

Another issue was using the workstations. The workstations are the home directory of the different users with individual access rights. In the case of the university, these are all students and staff. Working on the home directory with admin permissions was impossible since the workstations are located on a separate server with different access rights. This behavior was also not considered in the provided manual. The solution has been to put the files needed for commands with admin permissions into a local home directory.

When moving files, it should also be noted that the Linux 'mv' command does not work because it needs permissions on the source and destination directory, which is impossible with the current permission. The command 'cp' can be used as an alternative to the 'mv' command.

Docker

When installing Docker, the official documentation (Docker 2023b) can be used again as described in the instructions. During the installation, it was noticed that the installation of Docker Desktop did not work on the computers. Therefore, the CLI version of Docker, which also contains all the required functions, must be used. It must be checked whether Docker Compose was also installed since this is required for the execution of configuration files.

The connection to the CI/CD server shown in the overview of the complete technical realization in figure 2.2 also must be set up via Docker, since the CI/CD server is used as an image registry for the Docker images. The provided manual does not mention that the process from Docker to log in to the CI/CD server is session based. So if the session is terminated and problems occur, Docker needs to be logged in again on the CI/CD server.

Kubernetes also needs access to the secrets of the login to the CI/CD server. Therefore, these need to be set with a config file. Here a bug was encountered that the config file format is essential. The required encoding of the file is ASCII. If the file is encoded in UTF-8, it will not work. In which format the file is created or saved depends on the terminal and text editor used. The secrets can be read from files created by Docker. It should be noted that Docker's files are located in the /root directory instead of the home directory since all Docker commands had to be executed with admin permissions.

The last point with Docker is the creation and use of images. The guide provides a good starting point describing how to use the images from the CI/CD server. However, the guide does not detail how to create them. To figure out how to create images, one has to look into the corresponding files in the resource folder of the RenewKube, and RenewKube GUI Plugin, in which the source files for the images are located. Based on this, new images for the required Renew version were created and loaded onto the CI/CD server. How the providing images process work on the CI/CD server depends on the software on which this runs. Regarding the technical realization of Mushu with Renew, the documentation of Gitlab (GitLab 2023) should be followed to upload and mange the image stored.

6.4.2 Renew

After the environment was restored with the essential software, Renew had to be adapted. The goal was to create a minimally Renew version that was usable with the Cloud Native Plugin. Most of the adjustments in the code for using Renew 4.0 have already been made in the context of chapter 5.

A vital adjustment still made was the creation of new Gradle build tasks To later use the Cloud Native Plugin in the cluster, the Gradle task ‘cloudNativeCluster’ was created. This build contains a minimal Renew version with the addition of all necessary plugins for using the Cloud Native Plugin. In chapter 7 this will be discussed in more detail.

Another necessary change on the part of Renew was the adjustment of the start commands of the various Renew instances. Since the start command changed with Renew 4.0 release, it must be adapted in all places and all Docker images. In addition, new parameters were added, which are required by the newly added plugins in the build. An overview of the start commands for the various components in the cluster can be found in Confluence (Jan Henrik Röwekamp 2023).

6.5 Documentation

As mentioned in this chapter, the instructions for setting up the cluster needed to be completed and updated. Therefore, a significant focus of this work was to correct the deficiencies of the instructions in order to facilitate a new setup of the environment. To achieve this, additional sources were created to provide information about the setup and use of the cluster.

First, the provided instructions in Confluence were revised. The revision was done either directly in the text or by adding markers and notes to text sections. Thus, a large part of the manual could be preserved and be still in compliance with the changes.

Additional pages were created in the Confluence, which includes a short guide to the complete manual. This short guide includes a summary of all used and updated commands of the manual and can be used to set up a new cluster. Another short guide was created, which summarizes all necessary steps to execute if the master PC was switched off. Furthermore, scripts were created, which contain the commands of the manual and short guides so that a manual execution is not required anymore.

All information about the revised manual and the newly created guide can be found in the Confluence Jan Henrik Röwekamp 2023 and should be sufficient after the revision to set up a new cluster from scratch with the technical realization of Mushu.

6.6 Result

Figure 6.2 shows a graphical representation of the final result of this prototype. It is to be acknowledged that all components which were present in version before the reset shown in figure 6.1 are again available in the environment. Therefore these components are not described in more detail. A difference from the starting basis is the now completed CI/CD server integration. This was done directly through the provided instruction, since only the process how the cluster stores and obtains the Docker images needed to be changed.

The result at hand is a functioning cluster environment in which the technical realization of the Mushu architecture runs with Renew. Furthermore, this environment can be used to integrate the Cloud Native Plugin in the following prototype.

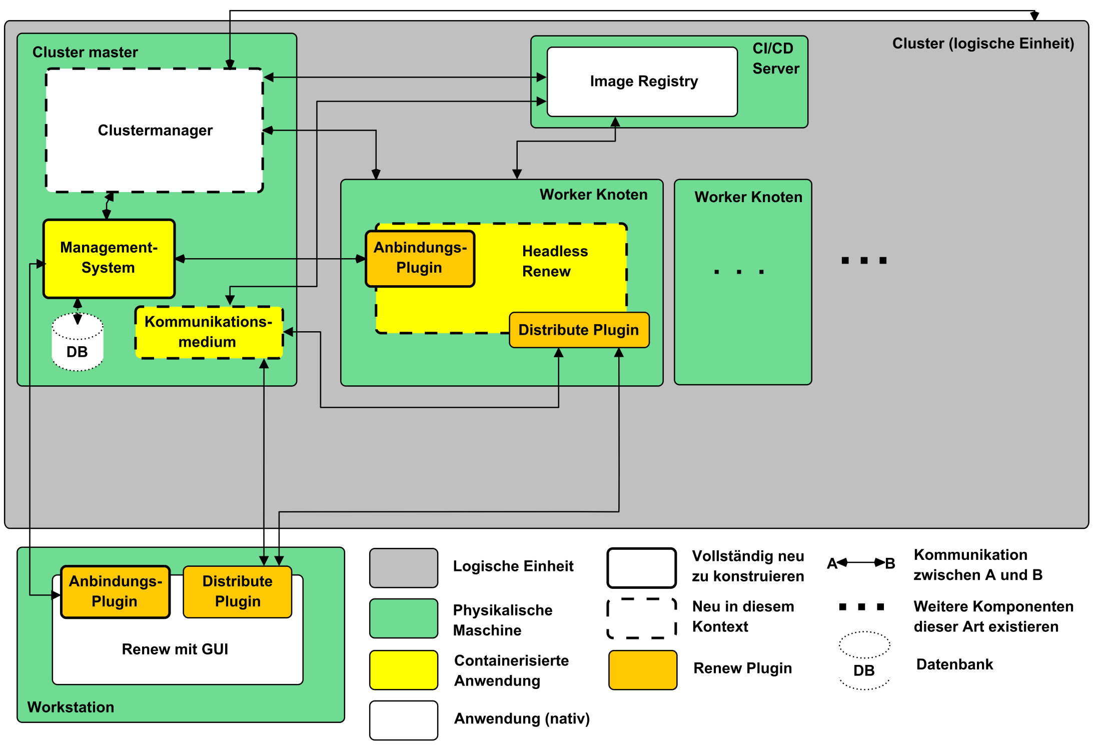


Figure 6.2: The result state of the reestablishment of the technical realization of the Mushu architecture.

6.7 Evaluation

In the evaluation of this prototype, the results are first discussed. Thereby, it is evaluated what went well during the development and how this process could have been even more effective. The limitations of the recovery of the environment follow. In that subsection, it is presented which components are still not entirely functional. In the last subsection a reference is made to requirements set up in chapter 3 and evaluated whether these were fulfilled.

6.7.1 Accomplishment of the Results

The development of this prototype in cooperation with a project team was of great advantage. Even if the participants had little to no prior knowledge of the technologies, working with someone on these new technologies was helpful. In addition, the distribution of work ensured that results were achieved more quickly and that work could be done on several components in parallel. The overhead created by the project team in the form of training and regular meetings was not a disadvantage. Through the regular meetings, the goal and structure of how to archive it needed to be clear at any time.

During reestablishment of the environment, it became apparent that a step-by-step approach is very advantageous for progress and understanding of the subject. The selected strategy was to consider the individual components and technologies as encapsulated as possible. An example was that Kubernetes was used as the base system, and initially, two Renew instances tried to communicate without Docker on that base. Step by step, one more Renew instance could be connected to the system, or an existing Renew instance could be replaced with a containerized Docker image. This procedure made it possible to isolate and fix errors relatively quickly.

However, this approach made a prerequisite that the technologies such as Kubernetes or Docker function as expected. In some cases, it would have been better to take an even smaller step and first test the essential software extensively with smaller programs since Renew is a very complex software project. This even more small-step approach would have had the advantage of creating an even better understanding of the individual technologies without the need to consider the overhead of Renew. Therefore, it would be advisable for future work to use the smallest-step approach if the used technologies are not or only barely known.

6.7.2 Limitations

Even though the recovery has been successful and complete for the most part, there are still current problems with the RenewKube plugin. The current issue is that RenewKube has problems with the newly created Docker images. RenewKube needs these images to perform the functions such as the auto-scaling of workers. After this thesis, the issue will continue to be worked on by the AOSE project team.

One approach to investigating this issue further would be to rebuild the environment with the Renew 2.6 images. It could be compared where precisely the difference to the newly created images lies, and a solution could potentially be found. However, since Renew 2.6 is unsuitable for integrating the Cloud Native Plugin, this will not be done in the context of this thesis.

6.7.3 General Evaluation

The prototype presented in this chapter deals with the requirements R5 and R6. The goal was to capture the state of the production environment, and to restore this environment after it has been unusable because of the performed system wipe.

Requirement R5, therefore, requires documentation of the previous implementation to see if it is suitable for integration with the Cloud Native Plugin. In section 6.3 the starting point of the cluster was presented. Since the cluster was no longer usable due to the mentioned system update, it was clear that the environment had to be restored.

For this purpose, requirement R6 states that the environment should be restored and brought into a state where the integration of the Cloud Native Plugin is possible. In addition, the requirement includes updating the documentation if it is unsuitable for the recovery process of the environment. Both requirements were addressed in section 6.4. The difficulties of the recovery were documented, and the existing documentation was extended accordingly. The recovery result was shown in section 6.6, and the limitations were outlined.

In summary, the prototype presented here addresses both requirements. The requirement R5 was completely fulfilled. For requirement R6, only partial success was achieved since the recovery is incomplete without a fully functioning RenewKube. Nevertheless, enough requirement elements have been successfully addressed, and a starting point has been created to perform the Cloud Native Plugin integration.

Chapter 7

Integrating Cloud Native Renew

The prototype discussed in this chapter deals with the integration of the Cloud Native Plugin into the technical realization of the Mushu architecture. This integration builds on the technical realization provided previously in chapter 6. This prototype focuses on the execution and documentation of the integration process of the Cloud Native Plugin. The goal is to provide a guide to which later integrations of subcomponents can be directed. The prototype was implemented in cooperation with the AOSE project team.

7.1 Requirements

The technical realization of the Mushu architecture has already been modeled but has yet to implement completely. To some extent, various components are still missing, such as the Cloud Native Plugin described here. These components must be integrated into the technical implementation at some point in order to achieve the goal of a complete technical realization. Therefore, this prototype should have a variant of the technical implementation in which the Cloud Native Plugin is present as an objective.

Since other components of the implementation are also missing, the integration of the Cloud Native Plugin can be regarded as an example of the integration process. This way, the various steps necessary to perform such integration are documented. The documentation can be used to create a guide for further integrations to follow. The guide can prevent specific problems or errors and provide a general direction for the integration process. Therefore, the process of the integration of the Cloud Native Plugin should be documented accordingly.

7.2 Specification

Initially, it should be documented which steps are necessary to integrate a new plugin into the technical realization of Mushu. It should be analysed which already existing components are affected by the integration and must be adapted. The adjustments that are necessary for the integration of the Cloud Native Plugin should be documented. In the end, a result should be presented showing whether the integration was successful or not.

7.3 Steps for Integration

Depending on which component of the technical realization of Mushu is to be implemented, different steps are necessary. In the case of the Cloud Native Plugin, a Renew plugin is added to the worker instances. Primarily due to the preliminary work done in chapters 5 and 6, the integration process is moderately simple.

First, a Gradle build task must exist in Renew itself, which combines the previously used plugins from Mushu with the Cloud Native Plugin and its dependencies. Attention must be paid to the dependencies of the various plugins and issues that arise when using specific plugins in containerized form.

A closer look at the Docker images follows. The Docker files must be adapted to use the new Gradle build task, and any required environment variables must be set. It may be necessary to adapt Docker images of components without direct dependency on the component to be implemented.

In the final step of the integration, the documentation of the environment and the scripts used must be adapted. Scripts are used at different places in the environment for setting up and resetting it, as well as for the creation of Docker images. If environment conditions or start commands have changed, these should be renewed. It is just as crucial that the documentation for setting up and using the environment are updated. Since the existing documentation is usually the first place for information about the environment, changes should always be noted there directly.

7.3.1 Creation of Build Tasks

Since the Cloud Native Plugin is a Renew plugin, a corresponding Gradle build task is required that combines the plugins already required for Mushu and Cloud Native. Before the integration, there were three build tasks used to implement Mushu architecture: 'renewkube_renew_base', 'renewkube_renew_headless' and 'renewkube_workstation'.

The Cloud Native Plugin is only needed for the worker instances. Therefore, a new build task, 'cloudNative_worker' was created in the first place. This can be used for the creation of Renew instances on worker nodes. For completeness, corresponding tasks were created for the other two build tasks: 'cloudNative_headless' and 'cloudNativeCluster'. Especially the 'cloudNativeCluster' build proved to be helpful. This build would be used mainly for testing and developing remote nets, as this build includes the graphical interfaces of Renew.

In general, the build tasks for the cluster should not have any dependencies on plugins that did not work in the cluster. For example, especially for build tasks that provide containerized instances, there must be no dependencies on plugins that use or require a GUI.

7.3.2 Providing new Docker Images

Many of the components present in the cluster are implemented in containerized form. Therefore, all affected images must be adapted during the integration. In addition, other images may also be affected, for example, if startup commands or environment variables are affected.

For integrating the Cloud Native Plugin, only one new Docker image was created for the workers. This image was made available on the CI/CD server[1]. The image is based on the existing images for the worker created for RenewKube. All existing images use different scripts to install and start the Renew instance, which had to be created accordingly.

[1]Container Registry: `https://git.informatik.uni-hamburg.de/tgi/Renew/container_registry`

7.3.3 Update of Scripts

As mentioned, scripts for installing and launching Renew instances are used to create the various Docker images. These scripts can be found in the resource folder of the RenewKube Plugin. If changes have been made to the Renew build or startup command, a new script should be created at this point for the corresponding component. The existing scripts can be used as a basis for creating new ones. For the integration of the Cloud Native Plugin, new scripts were created. The new image mentioned above could be created with the existing Docker files and the newly created scripts.

In addition to the scripts needed to create Docker images, there are some scripts that make setting up and restore the cluster easier. If the newly integrated component affects those processes, these scripts and the documentation must also be adapted.

7.3.4 Updated Documentation

Another vital point is updating the documentation. The existing documentation in the Confluence (Jan Henrik Röwekamp 2023) in the form of the already created step-by-step guide or the short guides is essential to change if necessary. The Confluence pages are the first place to go if problems occur or someone new works on the topic. Therefore, documenting every change in usage or installation or additional steps necessary to use a new component is crucial. For the Cloud Native Plugin, the changes in usage, startup command, and installation were directly noted and integrated into the instructions and startup commands.

7.4 Result

After all the steps were performed, the cluster could be started with the new image for the worker. No problems occurred, and the Cloud Native Plugin was functional. Parallel to the integration of the Cloud Native Plugin, the Status Monitor component was also integrated into the environment, which uses the Cloud Native Plugin to display status information about running instances. Since the Status Monitor is part of another thesis, it will not be discussed here in more detail.

Figure 7.1 shows the current status of the technical implementation of the Mushu architecture. Here again, the established design used in figure 2.2 is kept. Compared with the result in figure 6.2 of chapter 6, the Cloud Native Plugin component and the mentioned Status Monitor component were added. The remaining structure of the environment has not changed.

It is noticeable in figure 7.1 that the Renew instance in the workstation also requires the Cloud Native Plugin. This has the reason that in the following chapter 8, the usage of the Cloud Native Plugin was extended. Thus it is possible to use the Cloud Native Plugin functions directly from the simulation context. For this to work, the Cloud Native Plugin is also required in the workstation instance. The technical implementation was straightforward because the corresponding build task was already created in 7.3.1.

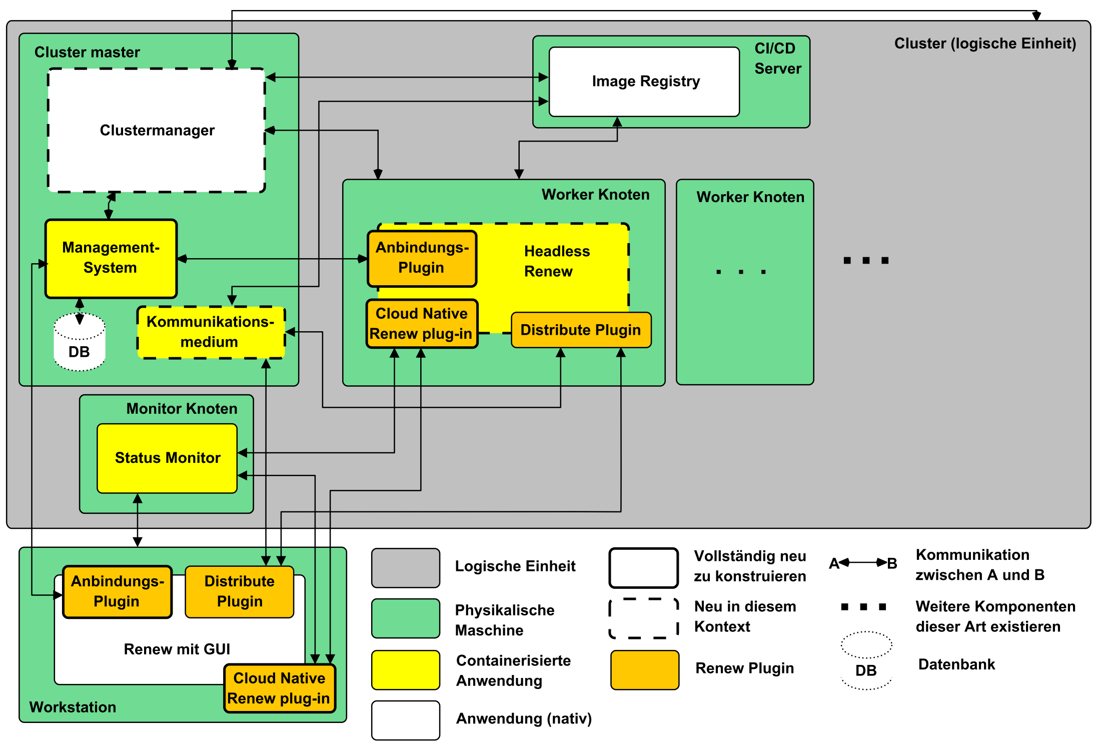


Figure 7.1: The current implementation of the technical realization of the MUSHU architecture in the production environment.

7.5 Evaluation

The evaluation of this prototype first deals with documenting the limitations, where aspects are outlined that were not considered during the integration. In addition, the limitation of the validation of the integration is mentioned. Furthermore, in this section, the evaluation occurs regarding the developed requirements in chapter 3 for this prototype.

7.5.1 Limitations

Since the exemplary integration of the Cloud Native Plugin was carried out in this prototype, mainly the steps necessary for this purpose were described and documented. Many of the MUSHU architecture components that are still to be integrated have individual problems that must be considered during their integration process. Thus in section 7.3 also, some general problems and obstacles were pointed out.

Another limitation of the integration is the validation of the integration. To verify availability, selected features of the Cloud Native Plugin were tested. Especially the provided endpoints for log output and status information were primarily used to verify the integration. This also includes the integration of the status monitor, which also uses the same endpoints of the Cloud Native Plugin for information retrieval in the current development phase. Even though the availability of these endpoints shows that the plugin is accessible in the cluster, no statement can be made about the other functions of the plugin at this point. Prototype 9 will therefore take a closer look at the validation of the integration process.

7.5.2 General Evaluation

The prototype deals primarily with the requirements R7 and R8. On the one hand, the actual integration of the Cloud Native Plugin into the technical realization of the MUSHU architecture should be carried out. On the other hand, a kind of manual should be created to be used as a basis for further integrations.

Both requirements are dealt with in the section 7.3. In the section, the integration of the Cloud Native Plugin is described step by step. The different stages of the integration are described, and possible problems are pointed out. In addition, the result of the integration is presented in 7.4, which fulfills the requirement R7.

The requirement R8 was only fulfilled to a considerable extent in this prototype. The description of the necessary integration steps focuses on the Cloud Native Plugin. As mentioned, other possible obstacles are also pointed out, covering only some possible problems. As described in the limitations, this is primarily because the integration process of the additional components is very individual, and it is not foreseeable what problems one might encounter. Nevertheless, the presented steps in section 7.3 should show the generally necessary steps for all remaining components. Therefore, the requirement R8 is considered as fulfilled.

Chapter 8

Extension of the Cloud Native Renew Usability

So far, the functions of the Cloud Native Plugin have been presented in chapter 4 and validated via their respective HTTP interfaces in the context of (Röwekamp, Taube, et al. 2021). The next step is to validate the usage of the plugin from a reference net context. For this purpose, a focus will be placed on using the plugin with the Renew software.

In the target scenario, the Cloud Native Plugin should enable someone to develop a reference net for distributed simulations. For this purpose, it makes sense to validate how the plugin functions can be used inside the reference net context. This extension of the usability is presented in this chapter.

Some elements were developed in the context of the AOSE project with the help of a small team of developers. Parts developed in this collaboration's context are marked as such.

8.1 Requirements

An essential element to validate something are examples. Therefore, examples for the Cloud Native Plugin will be created. A simple example should be provided in which the plugin's functions become apparent and how someone can use it when developing reference nets.

It should be determined how to incorporate the provided function in the development process of reference nets. This includes a discussion about the different possibilities to create example nets to determine which variant is the most suitable.

In addition, it should become understandable how these examples are created so that more can be developed in the future. Furthermore, it should be documented which special requirements exist for which type of example net so that they can be better created and understood.

8.2 Specification

When creating examples, there is always the question of how exactly the examples should be. In addition, all examples have the requirements that they are self-evident. Therefore, attention must be paid to the best variant to create these examples.

In this chapter, first the preliminary work from the work phase is taken up again, and it is documented which results were produced there. This mainly refers to the discussion about what the best way is to create examples for the Cloud Native Plugin. In addition, the plugin's extension is documented, which was necessary for creating the examples.

This is followed by an elaboration of the preliminary work of the work phase. A structure is given to how examples can be built in the context of the Cloud Native Plugin. In addition, examples are created that can be used in the minimal context, as well as in the distributed context.

Partially, these results were also developed in cooperation. All content to which this applies is marked as such.

8.3 The best Way to Create Examples

The discussion on creating sample nets for the Cloud Native plugin was created during the work phase described in 4 in cooperation with Jan Henrik Röwekamp. The implementation and development of the examples were done independently.

The discussion generally concerns creating example nets for the Cloud Native Plugin. But these should also serve as a basis for testing the entire MUSHU architecture. Several possibilities have been considered. Three approaches are compared, each with advantages and disadvantages. It was decided on a variant in which the Cloud Native Plugin had to be extended.

The further development is also documented in this chapter. Likewise, documentation of the example net production for the Cloud Native Plugin and the documentation of specific examples follows after this section.

8.3.1 Different Types of Example Nets

At first, the creation of example nets with the system software Curl is analyzed. This is followed by the possibility of net creation, focusing on the implementation in RENEWKUBE and the RENEWKUBE manager. The last option describes the Cloud Native Plugin extension to provide an abstract class as in interface for its functions. In the end the last option was chosen as the best way to create example nets.

Example Nets with Curl

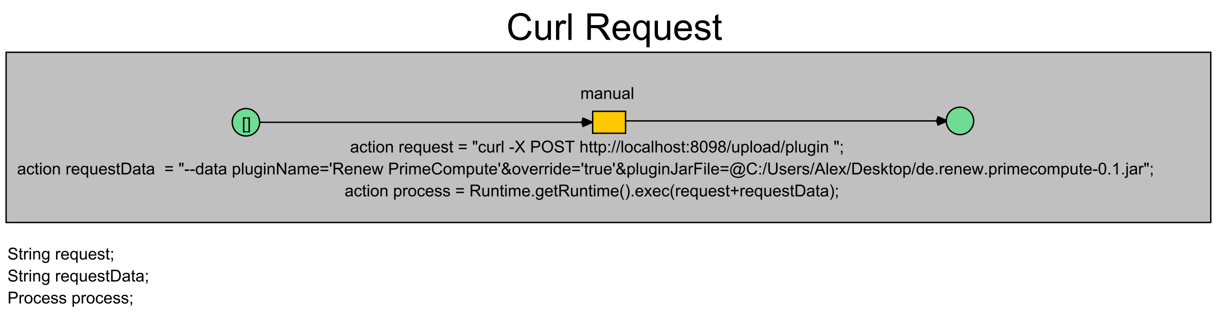
The first idea for creating sample meshes has been to use the curl command. Curl [1] is a command line tool which is capable to make HTTP requests. This function would be sufficient for interacting with the Cloud Native Plugin.

In Java, it is straightforward to run command line commands, with the provided Runtime implementation. Since the reference nets can execute all Java code, this option would have been the fastest to implement. However, after briefly testing the feasibility of this option, it became clear that it was not very practical.

The biggest problem was the interpretation of return values. Using command line tools like Curl makes it hard to work with return values, because of the nature that they need to be executed with the Java 'Runtime.exec()' function. So the further use of different return values and statuses of a specific HTTP request would not be possible without a more complex net. Another problem was that the curl commands became very long. This is especially true when multiple requests are made to the Cloud Native Plugin in succession.

The fist problem represents a source of error or significant additional work. The other problem contradicts the requirements of the example nets that it needs to be easy to understand. Therefore, it was relatively quickly decided against using curl commands, and further possibilities for implementation were considered.

[1]Link to the Curl Website: https://curl.se



```
action request = "curl -X POST http://localhost:8098/upload/plugin ";
action requestData  = "--data pluginName='Renew PrimeCompute'&override='true'&pluginJarFile=@C:/Users/Alex/Desktop/de.renew.primecompute-0.1.jar";
action process = Runtime.getRuntime().exec(request+requestData);

String request;
String requestData;
Process process;
```

Figure 8.1: Example net of using a curl request to upload a plugin

However, it should be noted that it is possible to interact with the Cloud Native Plugin using curl commands. In figure 8.1 can be seen how the upload function from Cloud Native Plugin would look like using Curl. The whole functionality is in the transition, which assembles the Curl command and later executes it with the Java 'Runtime.exec()' function. Note that the parameters for 'IP', 'Port', 'PluginName', and 'PathToPluginJar' are written directly into the request.

Example Nets via RenewKube and RenewKube Manager

The next option would have been to outsource the logic from the request to the RenewKube plugin. Outsourcing is possible because each Renew instance, i.e., each node, provides the same endpoints. To send a successful request, only the address and port of the target and source node would be needed.

The RenewKube manager would be the first choice because it knows the addresses of all nodes. Therefore, it would be possible to provide functions in the RenewKube manager, which would then be used in the sample network for the request.

However, this approach presents some problems. The first problem is that it contradicts the conception of the Mushu architecture. If one were to implement this variant in such a way, the RenewKube manager would act as the intermediate instance for each communication between two Renew Nodes. However, this is not intended in the architecture. The RenewKube manager is intended to act as a manager for the cluster, particularly as a coordinator for scaling and not as a communication medium. Another problem would be that the Cloud Native Plugin would not have any sample nets available if it is used in a standalone variant.

Since this variant of the implementation is incompatible with the concept, there is also no further implementation. It would theoretically be possible to extend the RenewKube manager to take over the communication. However, this would have to be implemented in the RenewKube manager, before it could be used in a sample net for the Cloud Native Plugin.

Example Nets with internal Cloud Native Renew Functionality

The last option involves the idea that the Cloud Native Plugin provides the desired functionality. This means that the Cloud Native Plugin provides functions that can be used in a reference network to complete a request.

This variant has several advantages. On the one hand, the Cloud Native Plugin acts as a receiving and sending source. Furthermore, executing requests to itself is possible since its own IP can be used as the target IP. Since the functionality is implemented in the plugin itself, the function would still work in a standalone scenario. Another advantage is that a lot of logic can be abstracted. So only one function in a reference net has to be used to interact with the API. This makes an example clearer and easier to understand.

On the other hand, this possibility also has disadvantages. The abstraction of the functionality is advantageous for the initial understanding, but the exact functionality cannot be recognized without a closer look into the source code. The most significant disadvantage is that an additional effort in the development arises since every function must be abstracted.

8.3.2 Outcome

After considering which version is best for creating examples for the Cloud Native Plugin, the third version, creating examples with internal functions, was chosen. Evaluating the pros and cons of all versions was ultimately the reason for this decision.

By choosing this variant, extending the plugin was an additional effort. The plugin needed an abstraction for every function used for example nets. However, it is advantageous because the plugin's extension makes it easier to use it in a reference net overall.

The plugin has already been extended in the described work phase in chapter 4. The extension will be explained in more detail in the next section. A resulting advantage is that the functions for reference nets can be extended after need. Therefore, the additional expenditure for the development is not that great, since changes would only appear if the function is desired to use inside of an refreence net.

The resulting examples are presented at the end of this chapter. In addition, an overview of how to proceed to create further examples is given.

8.4 Cloud Native Renew Plugin Extension

This section documents the extension made to the Cloud Native Plugin. The extension's functionality is explained, and it is shown what precisely it brings and why it is necessary to use it.

The section is divided into two parts. First, the basic extension is considered. Here the focus is on the elaborated contents from the working phase. These results in this subsection have been developed in collaboration with Jan Henrik Röwekamp. The second part of this section deals with the changes to the extension, which have been developed during this thesis.

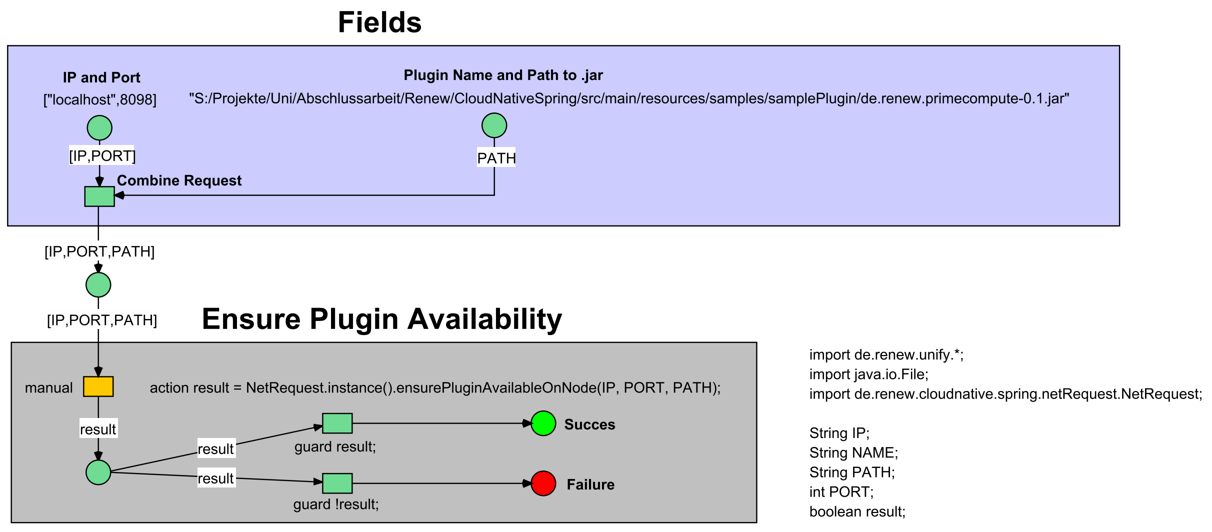
8.4.1 General Extension

The extension aims to provide methods that can be easily used in a reference net. The functions execute the request to the remote Cloud Native API and work with parameters that are passed to them. The focus of the implementation was the upload and start of plugins on a node.

The general functions for the extension can be found in the module 'NetRequest' in the Cloud Native Plugin. The core implementation is in the 'NetRequest' class. This class provides functions that can ultimately be used in reference nets.+

In addition, The 'HealthMetricReply' class was created to provide a JSON template to make response evaluation easier. The health metrics can determine which plugins are present on the requested node.

A more detailed overview of the implemented functions can be found in the following subsection 8.4.2.


Fields
IP and Port
["localhost",8098]
Plugin Name and Path to .jar
"S:/Projekte/Uni/Abschlussarbeit/Renew/CloudNativeSpring/src/main/resources/samples/samplePlugin/de.renew.primecompute-0.1.jar"
[IP,PORT]
PATH
Combine Request
[IP,PORT,PATH]
[IP,PORT,PATH]
Ensure Plugin Availability
manual
action result = NetRequest.instance().ensurePluginAvailableOnNode(IP, PORT, PATH);
result
result
result
guard result;
guard !result;
Succes
Failure

```
import de.renew.unify.*;
import java.io.File;
import de.renew.cloudnative.spring.netRequest.NetRequest;

String IP;
String NAME;
String PATH;
int PORT;
boolean result;
```

Figure 8.2: Example net of ensure plugin availability on a specific Cloud Native Renew Node

8.4.2 Explicit Extension

This subsection will take a closer look at the functions implemented around the 'ensurePluginAvailableOnNode' function, and the resulting example nets. A total of 4 different example nets were created for different Cloud Native Renew functions: Plugin Upload, SNS Upload, Simulation Start, and Simulation Control. Parts of this implementation have been developed in cooperation with the AOSE project group.

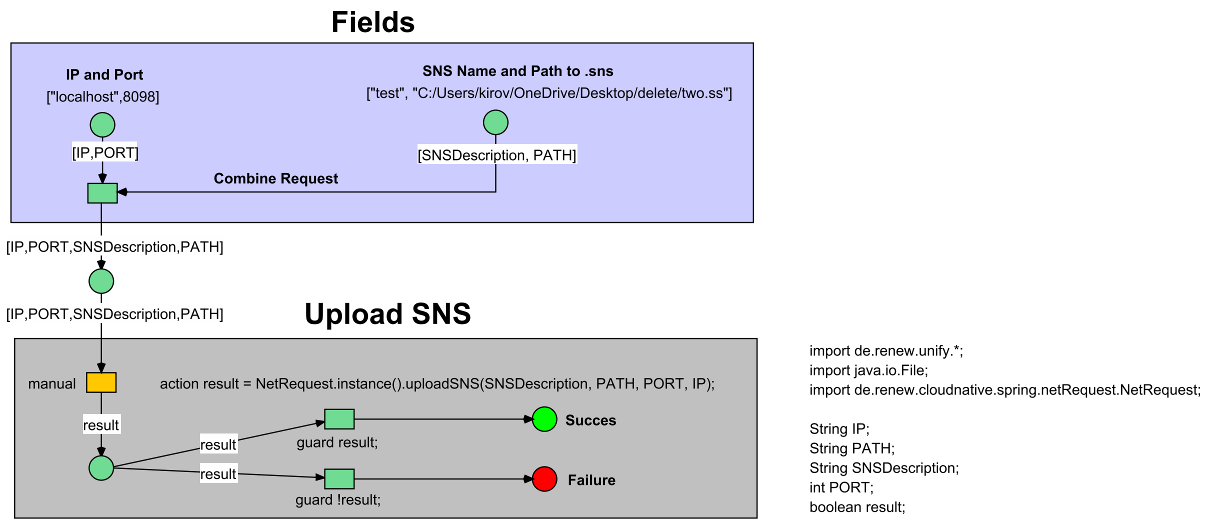
Ensure Plugin Available On Node

First, the 'ensurePluginAvailableOnNode' example was created, which can be seen in figure 8.2. This example was already created before the work in the AOSE project context and served as a basis for the following examples. The created net can be used to guarantee that a plugin is available and loaded on a node. For this purpose, the method is passed the node address and port, as well as the plugin name and .jar file. The plugin .jar is passed to the method as a path. The method first checks whether the plugin already exists on the desired node. To do this, a request is sent to the Cloud Native API of the addressed node, which queries the health metrics. If the plugin is not available there, the plugin is uploaded to the corresponding node via a new request and then loaded. Finally, it is rechecked if the plugin is available on the node. The functions response with a boolean value to express if the desired plugin is now available on the desired node. Those return values can then be used in the reference net for further simulation.

Upload SNS

Next, the UploadSNS example was created, which can be seen in figure 8.3. This example was created in cooperation with the AOSE project. The basis for this example is the provided functionality of the Cloud Native Plugin to upload nets to a Renew instance.

As before, the IP and port of the target instance are required for this function. In addition, the path to the .sns file must be specified. The last parameter to be specified is the SNSDescription, which sets the name of the SNS on the target instance. The structure of the example in figure 8.3 is very similar to the example in figure 8.2. The parameters are assembled at the beginning of a request, which are then passed to the created method


Fields
IP and Port
["localhost",8098]
SNS Name and Path to .sns
["test", "C:/Users/kirov/OneDrive/Desktop/delete/two.ss"]
[IP,PORT]
[SNSDescription, PATH]
Combine Request
[IP,PORT,SNSDescription,PATH]
[IP,PORT,SNSDescription,PATH]
Upload SNS
manual
action result = NetRequest.instance().uploadSNS(SNSDescription, PATH, PORT, IP);
result
result
result
guard result;
guard !result;
Succes
Failure

```
import de.renew.unify.*;
import java.io.File;
import de.renew.cloudnative.spring.netRequest.NetRequest;

String IP;
String PATH;
String SNSDescription;
int PORT;
boolean result;
```

Figure 8.3: Example net of uploading a SNS file on a specific Cloud Native Renew Node

uploadSNS in the ‘netRequets’ class. In the method, the HTTP request is made to the corresponding target instance, and a True or False value is returned depending on the response. As before, the return values can be used further in a reference net or to show whether the upload was successful.

Start Simulation

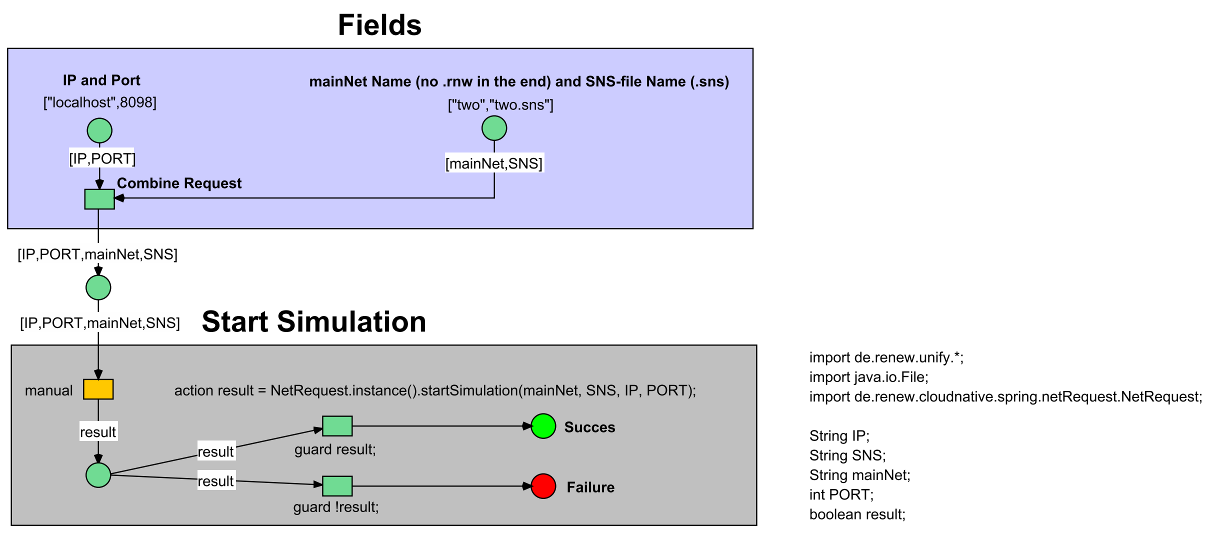
The Cloud Native Plugin provides a separate endpoint for starting a simulation. For this reason, there is also a separate example net for starting a simulation, which can be seen in figure 8.4. This example was also created in cooperation with the AOSE project.

As seen in figure 8.4, parameter once again must be set. IP and port parameters do not differ from the function of the other examples. Additionally, the name of the SNS, which is located on the target instance, must be specified. This can be the parameter SNSDescription from the example in figure 8.3 if an SNS has been uploaded before. Furthermore, the name of the net from which to start the simulation needs to be given as a parameter.

The method, executed in the manual Transition in the network, is similar to the other examples. Thus, an HTTP request is executed on the corresponding StartSimulation endpoint of the target instance. Depending on the HTTP response, a corresponding return value is set. This is passed back to the network and can be used further.

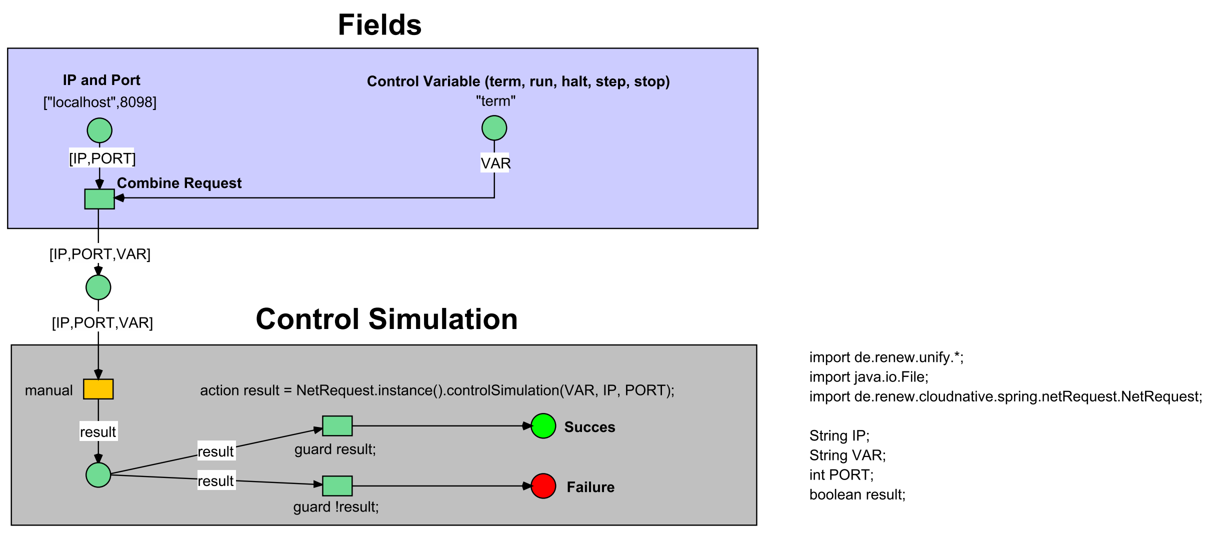
Control Simulation

The last example shown in figure 8.5 is for the simulation control function of a Renew instance. This example was also developed in the AOSE project. Like before, it is necessary to specify the IP and port of the target instance. Additionally, only one more parameter is needed to send the simulation control instruction. There are three options: term/stop for stopping the simulation on the target instance, run/start for starting the simulation, and step for switching a transition. As with the other examples, the implemented method does nothing more than submit an HTTP request to the target instance with the selected parameters. Upon this, the method rules with appropriate return values for successful execution.



```
import de.renew.unify.*;
import java.io.File;
import de.renew.cloudnative.spring.netRequest.NetRequest;

String IP;
String SNS;
String mainNet;
int PORT;
boolean result;
```

Figure 8.4: Example net of starting the simulation on a specific Cloud Native Renew Node



```
import de.renew.unify.*;
import java.io.File;
import de.renew.cloudnative.spring.netRequest.NetRequest;

String IP;
String VAR;
int PORT;
boolean result;
```

Figure 8.5: Example net of controlling a simulation on a specific Cloud Native Renew Node

8.5 Evaluation

First, it will be discussed how the result from the discussion in section 8.3 has affected this prototype and whether the right decision was made. Afterward, the evaluation occurs regarding the established requirements from chapter 3 and in particular for requirement R9.

8.5.1 Methods of creating Cloud Native Renew Examples

In section 8.3.1, three different options were mentioned to improve the usability of the Cloud Native Plugin in the context of its usage in reference nets. The variants with the implementation via Curl request and own abstraction class were described in more detail. In this subsection, we will evaluate why it was decided to create an abstraction class and not to use the Curl implementation. The third possibility with the implementation inside the RenewKube manager is not considered further here since this was already removed at the beginning of the decision process.

Problems were already mentioned in 8.3.1, which describes why the Curl request is unsuitable for the requirements. In the end, precisely, these problems were the reason for the decision to not implement Curl request. With the production of additional example nets, defining much overhead code in the reference nets for the most basic functions would have been necessary. Especially the response handling, the problem would have arisen that each function would need its sequence of checks. Both points listed here are detrimental to the goal of making the functions readily available and understandable.

In the end, the implementation with an abstraction class was chosen. Besides the advantages explained in section 8.3 also, possible disadvantages were also determined. One disadvantage was that the more profound understanding of how the functions is needed and the response handling is no longer easily understandable. But it has been shown that the good documentation of the endpoints of the Cloud Native Plugin and the source code documentation of the abstraction class did not reduce the understanding. The second problem was that the extension of the Cloud Native API led to the fact that the created abstraction class must also be extended. However, this extension is a small expenditure. Thus students in the AOSE 22 project could extend the abstraction class independently after an exemplary abstraction class was implemented.

8.5.2 General Evaluation

The prototype addresses requirement R9 defined in chapter 3. It was required that the Cloud Native Plugin functions are made usable from the context of reference nets. The reason behind this requirement is that simulations for the cluster can be created more easily.

This prototype has looked at three variants of the possible extension to meet the requirement. The decision of which is the best variant was documented and evaluated in section 8.5.1. In addition, the extension was used to create examples for the four most essential functions in the cluster: Plugin Upload, SNS Upload, Simulations Start, and Simulation Control.

The successful creation of the examples shows that the functions of the Cloud Native Plugin can now be used from inside a reference net context. In addition, the examples serve as a tool for directly using the desired functions in later instances in the cluster. Furthermore, the examples created also fulfill the documentation aspect required by R9.

Chapter 9

Validation of the Cloud Native Renew Plugin

After the integration from Cloud Native Plugin in chapter 7, this chapter validates this integration. The goal is to show that the provided product environment from chapter 6 is in a working state and that the actual integration performed in chapter 7 was successful.

If the validation is successful at this point, it is clear that the integration of the Cloud Native Plugin as well as the setup of the production environment has been successful. Thus, a new foundation has been created as a starting point for further integrations.

9.1 Requirements

An excellent way to validate components or systems is to show good use cases. In this thesis context, creating an example reference net is helpful because it fulfills two requirements simultaneously. First, a working example shows that the newly integrated functions are available in the cluster. This availability would thus validate the successful integration of the Cloud Native Plugin. Second, would it also validate that the environment is functional. In the context of this thesis, this would mean that the recovery of the production environment in chapter 6 has been as successful.

Besides that validation, an example can also be used for documentation purposes. Since examples are usually easier to understand than textual explanations. This documentation would thus also provide some guidance for further integrations in the future.

9.2 Specification

For the validation, the goal is to use a coherent example based on reference nets. As a starting point, an example has already been developed in the context of RenewKube from (Röwekamp and Moldt 2019). This example will be enriched with the created examples from chapter 8 to integrate the functions of Cloud Native Renew.

9.3 Creation of a Coherent Example

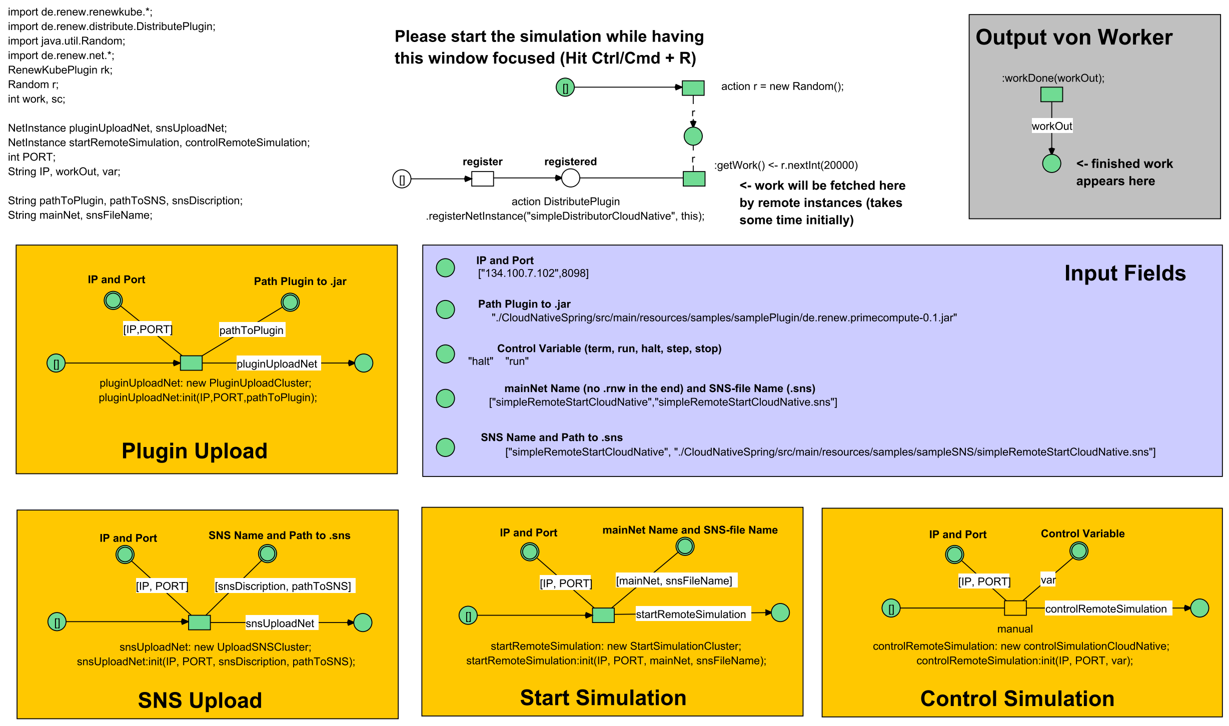
The basis of the validation is a coherent all-in-one example to be executed on the cluster. This is based on the examples first created in the context of (Röwekamp 2023b) as part of the RenewKube demonstration. The original examples can be found inside the provided virtual machine images after following the setup guide on (Röwekamp 2023b), or in the RenewKube folder of the Renew repository. Two reference nets were created, which were intended to demonstrate the interaction between the host and remote instance. Furthermore, the examples created in chapter 8, which show the Cloud Native Plugin functionalities, were integrated into the all-in-one example to validate the integration of the plugin. The changes in the two nets are discussed in more detail in the following sections. In addition, the simulation process is examined in more detail to show which functions are executed and how that affects the state of the cluster.

9.3.1 Created Reference Nets

The all-in-one example consists of two components. A system net that is intended for the primary instance and user interface, and a remote execution net will be executed on the remote Renew instance.

Main System Net

In figure 9.1, the mentioned main system net can be seen. This reference net is intended to be specified by the user to set up the primary prerequisites for the remote instance. The elements in the upper third in figure 9.1 are shown, which were taken from the original example. In the center part of it, this system net is registered at the Distribute RMI, and a random number is created. Then the call to the remote instance is controlled with the :getWork() transition. In the right part of the upper third, the return values of the remote instance are collected, and the results are stored. The left top part shows the required dependency for the Java code execution.

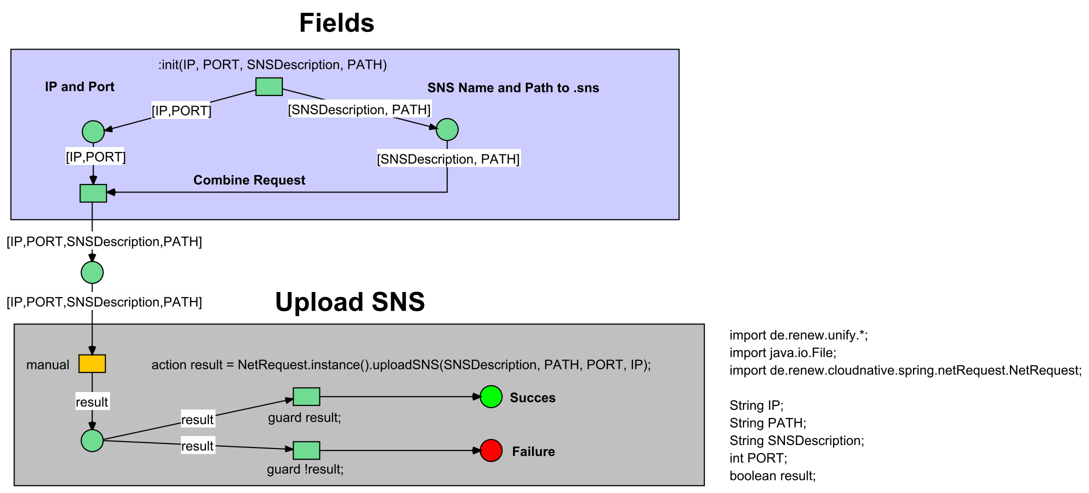


```
import de.renew.renewkube.*;
import de.renew.distribute.DistributePlugin;
import java.util.Random;
import de.renew.net.*;
RenewKubePlugin rk;
Random r;
int work, sc;

NetInstance pluginUploadNet, snsUploadNet;
NetInstance startRemoteSimulation, controlRemoteSimulation;
int PORT;
String IP, workOut, var;

String pathToPlugin, pathToSNS, snsDiscription;
String mainNet, snsFileName;
```

Please start the simulation while having this window focused (Hit Ctrl/Cmd + R)
[]
action r = new Random();
r
r
register
registered
[]
:getWork() <- r.nextInt(20000)
<- work will be fetched here by remote instances (takes some time initially)
action DistributePlugin .registerNetInstance("simpleDistributorCloudNative", this);
Output von Worker
:workDone(workOut);
workOut
<- finished work appears here
IP and Port
Path Plugin to .jar
[IP,PORT]
pathToPlugin
[]
pluginUploadNet
pluginUploadNet: new PluginUploadCluster;
pluginUploadNet:init(IP,PORT,pathToPlugin);
Plugin Upload
Input Fields
IP and Port
["134.100.7.102",8098]
Path Plugin to .jar
"./CloudNativeSpring/src/main/resources/samples/samplePlugin/de.renew.primecompute-0.1.jar"
Control Variable (term, run, halt, step, stop)
"halt" "run"
mainNet Name (no .rnw in the end) and SNS-file Name (.sns)
["simpleRemoteStartCloudNative","simpleRemoteStartCloudNative.sns"]
SNS Name and Path to .sns
["simpleRemoteStartCloudNative", "./CloudNativeSpring/src/main/resources/samples/sampleSNS/simpleRemoteStartCloudNative.sns"]
IP and Port
SNS Name and Path to .sns
[IP, PORT]
[snsDiscription, pathToSNS]
[]
snsUploadNet
snsUploadNet: new UploadSNSCluster;
snsUploadNet:init(IP, PORT, snsDiscription, pathToSNS);
SNS Upload
IP and Port
mainNet Name and SNS-file Name
[IP, PORT]
[mainNet, snsFileName]
[]
startRemoteSimulation
startRemoteSimulation: new StartSimulationCluster;
startRemoteSimulation:init(IP, PORT, mainNet, snsFileName);
Start Simulation
IP and Port
Control Variable
[IP, PORT]
var
[]
controlRemoteSimulation
manual
controlRemoteSimulation: new controlSimulationCloudNative;
controlRemoteSimulation:init(IP, PORT, var);
Control Simulation

Figure 9.1: Main system net for the all-in-one example

The blue highlighted area 'Input Fields' in figure 9.1 defines specific settings for the environment and remote instance. First, the IP as a string and the port as an integer from the remote Renew instance are specified. The next element is the path to a pre-compiled plugin .jar file. The path can be specified as relative or absolute as a string type. In this example, the PrimeCompute Plugin is used, which the remote execution net needs to calculate if a given integer is a prime number. Next, the simulation control commands are specified. Those commands can be used to control the simulation of the remote RENEW instance. Only the commands 'halt' and 'run' are used in the example, but all commands mentioned in section 8.4.2 can be used. The second to last item specifies the name of the .rnw net to be executed by the remote instance. In addition, the name of the SNS under which the SNS was uploaded to the remote instance is specified here. The last item specifies the exact name under which the SNS should be stored on the remote instance and an absolute or relative path to the SNS file. In the example, the SNS information refers to the remote execution net, which will be described next. Unless otherwise noted, all information from the input fields should be of type string.


Fields
:init(IP, PORT, SNSDescription, PATH)
IP and Port
SNS Name and Path to .sns
[IP,PORT]
[SNSDescription, PATH]
[IP,PORT]
[SNSDescription, PATH]
Combine Request
[IP,PORT,SNSDescription,PATH]
[IP,PORT,SNSDescription,PATH]
Upload SNS
manual
action result = NetRequest.instance().uploadSNS(SNSDescription, PATH, PORT, IP);
result
result
guard result;
Succes
result
guard !result;
Failure

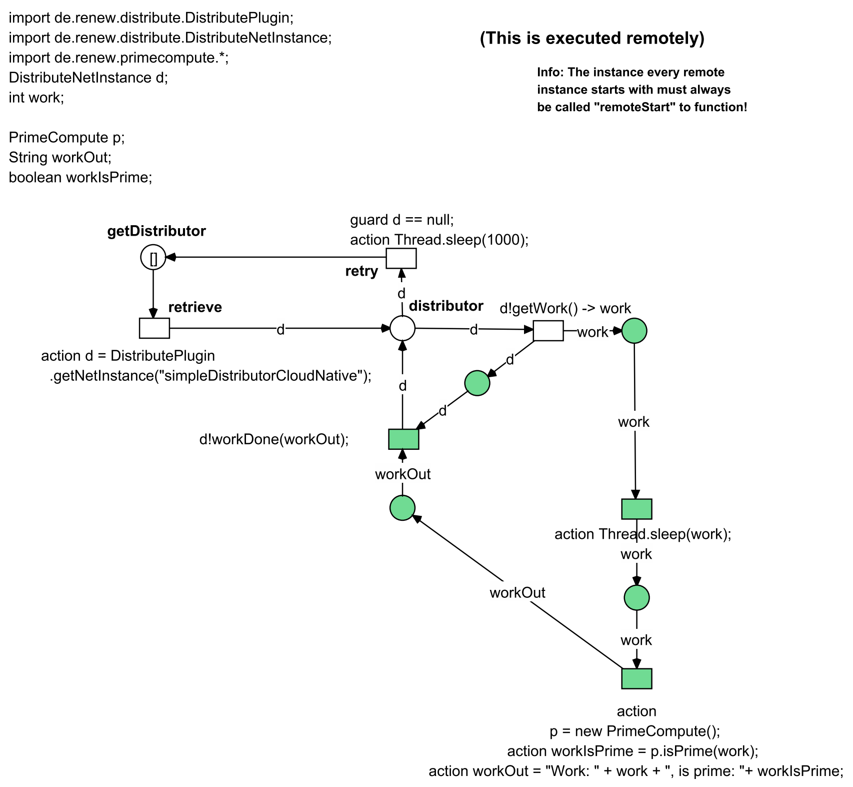
```
import de.renew.unify.*;
import java.io.File;
import de.renew.cloudnative.spring.netRequest.NetRequest;

String IP;
String PATH;
String SNSDescription;
int PORT;
boolean result;
```

Figure 9.2: Modified upload SNS net for the all-in-one example

The last component of the system net are the yellow highlighted areas. The respective areas cover a function of the Cloud Native Plugin and use a slightly modified variant of the sample nets created in section 8.4.2. The basic structure of the net is always the same. A net instance of the respective function is created and enriched with the defined parameters from the Inputs Fields area. It is important to note that virtual places are used for passing on the parameters so that the contents can be used several times. After the execution of the transition, an instance of the respective net is created and executed. In figure 9.2 is once exemplary the modified variant of the Upload SNS function to see. The basic structure and function is identical to the variant created in figure 8.3. The only difference is the transition with the remote call':init()', which is executed from the system net to fill the locations with the appropriate values for IP, Port, SNS Name, and Path to SNS. It should also be noted that the manual transition is still present in all four components, so when running the simulation the user has to switch the single transitions of the nets manually.

Remote Execution Net

The second example is the Remote Execution net. As shown in figure 9.3, it is not so different from the original RENEWKUBE example. The area with the white transitions and places regulates the periodical request to the Distributor RMI form the system net instance. The green area is responsible for the processing of the workload. This consists of a thread sleep in dependence on the random number given by the system net. The next step determines whether the passed number is a prime number. An important point to note here is that the prime number is determined with the help of the PrimeCompute Plugin, which has been added to the original net. After the processing, a modified result is passed to the system net containing the random number and the information if the given number is a prime number.



```
import de.renew.distribute.DistributePlugin;
import de.renew.distribute.DistributeNetInstance;
import de.renew.primecompute.*;
DistributeNetInstance d;
int work;

PrimeCompute p;
String workOut;
boolean workIsPrime;
```

(This is executed remotely)
Info: The instance every remote instance starts with must always be called "remoteStart" to function!
getDistributor
[]
guard d == null;
action Thread.sleep(1000);
retry
d
retrieve
d
distributor
d!getWork() -> work
d
work
action d = DistributePlugin
.getNetInstance("simpleDistributorCloudNative");
d
d
d!workDone(workOut);
work
workOut
action Thread.sleep(work);
work
workOut
work
action
p = new PrimeCompute();
action workIsPrime = p.isPrime(work);
action workOut = "Work: " + work + ", is prime: "+ workIsPrime;

Figure 9.3: Remote execution net for the all-in-one example

9.3.2 Simulation Process

In this section, the simulation process of the shown nets is described. Particular aspects are pointed out, and a reference to the validation is made. In addition, it is described how the state of the remote RENEW instance changes. The described simulation was executed in the ART Lab on ART PC 15 as master and ART PC 2 as a remote instance.

To start the simulation, the system net and the individual subnets for the Cloud Native functions must be loaded in the RENEW instance. Then the simulation need to be started with the system net as the main net. After the instance has been registered with the Distribute RMI, it waits for a result from a remote RENEW instance. At this point, the remote RENEW instance has no active simulation, the Remote Execution net is unavailable, and the required PrimeCompute Plugin is not loaded.

Since the subnets of the Cloud Native functions have a manual transition, they have to be triggered manually. The sequence of that is important in some cases. For example, the simulation must have been started before the simulation can be controlled. Likewise, the Remote Execution net and the PrimeCompute Plugin must be available on the remote instance before the simulation can be started.

If a Success is returned by the subnets, only the viewpoint from the host RENEW instance is considered. For example, a subnet return success if the request to the remote instance was made and accepted. There is no error handling implemented should the remote instance run into an error state after the response to the request. The only exception is the EnsurePluginAvailability net in figure 8.2, which guarantees that a plugin is available and therefore checks if the request was handled successful.

On the remote instance side, the following occurs after the plugin upload, SNS upload, and simulation start command are executed from the host instance. First, the system net instance is fetched from the Distribute RMI, and the workload (a random number) is received there using the :getWork transition. After a ThreadSleep in the length of the random number, the uploaded PrimeCompute Plugin is used to check if the random number is a prime number. The result is then passed back to the system net.

In the system net, at this time, results are coming into the 'Output from Worker' area. It can happen that it take up to 20 seconds to for results to show up. This is because of the random generate integer, which can reach upto 20000, is passed to the remote execution net and the corresponding thread sleep in milliseconds of that random generated number.

It is possible to pass two commands, 'halt' and 'run' to the remote instance with the Control Simulation component to stop and restart the simulation. Each time a command is executed, a new net instance of the Control Simulation net is created. The transition should be bound manually to decide which command to execute, otherwise the selection is non-deterministic.

9.4 Evaluation

The evaluation of this chapter is divided into two parts. First, reference is made to the limitations of the validation. It explains which elements still need to be validated in further instances. Then it is evaluated once how the prototype fulfilled the established requirements in chapter 3.

9.4.1 Limitations of the Validation

The focus of this validation was set on the Cloud Native Plugin integration into the production environment. The previous functions of auto-scaling of the RenewKube Plugin were not considered here. Thus, this represents the scope and limiter of this validation. The all-in-one example shows that the framework of the production environment is usable. It also shows the usability of the Cloud Native Plugin functions. However, the focus on combining direct interaction with RenewKube functions has yet to be considered further here.

The reason for the limitation of the validation was that, at the time of this thesis, the recovery of all RenewKube functions still needed to be completed. Should the RenewKube integration be completed later, the example created can be modified with minor modifications so that the RenewKube functions are also validated. The only change that would have to be made would be the assignment of the input fields that are currently entered by the user. If RenewKube works, these fields could be filled in by the RenewKube Manager and would no longer have to be entered manually.

9.4.2 General Evaluation

The prototype described in this chapter deals with requirement R10 developed in chapter 3. The goal was to show the interaction between the Cloud Native Plugin and the production environment. Therefore, the requirements were set up to create a coherent example and corresponding documentation for the usage.

Using a coherent example as validation for successful integration is especially useful in the case of Renew. This is because, by using previously unusable functions, the example shows that the plugin's integration has been successful and, at the same time, how to use them in the future, should one want to build on them.

Furthermore, such an example also serves the purpose of documentation. In addition to the summary, the content of this chapter is also the basis for the documentation of the use of the example and, thus, the use of the functions or the production environment.

Since the example was also created with the tools provided by Renew, this shows that the software's fundamental functions still work. Another advantage of creating an example in the Renew context was that the users were familiar with the environment and thus were more likely to develop an understanding of the use and presentation of the example. The integration could have been also done with a sequence of HTTP requests and their evaluation. However, this would not have been very descriptive or easy to understand and would not have contributed to the documentation of the use of the plugin.

Finally, to refer to requirement R10, a successful demonstration of using the new functions was achieved. Thus, a successful integration is validated, and the level of documentation was increased.

Chapter 10

Evaluation

In this chapter, I will evaluate in what matter the integration of the Cloud Native Plugin has been successful. Further, it will be evaluated how far Cloud Nativity aspects are now present on the cluster. In addition, it will also refer to the overall work and working method and evaluate how helpful it has been for this thesis. Finally, I will refer to the requirements set up in chapter 3 to see if the developed requirements were fulfilled.

10.1 Cloud Native Aspects

As already described in chapter 2, there are four know aspects of Cloud Nativity, agility, operability, observability, and resilience. The success of integrating Cloud Nativity aspects into the cluster environment can be seen best by individually examining them. It will be shown which components of the cluster contribute to the fulfillment of Cloud Nativity.

10.1.1 Agility

The Cloud Native Plugin has almost no direct influence on the agility aspect. The only thing that the plugin has improved directly is the individual deployment of nodes. This means that heterogeneous nodes with different Renew plugins can be created very quickly, which is now also possible from context of reference nets.

In the cluster, the agility aspekt can be found in the separate development of different component and Renew plugins. These plugins are developed in small steps and will be further encapsulated with the Renew release 5.0.

If we look at the technical realization of the Mushu architecture, the agility aspect can also be seen in the usage of the GitLab CI/CD instance. The individual Docker images are provided there, which then only have to be brought into the cluster.

The development in Renew, and consequently in the Cloud Native Plugin, is organized according to the current software development standards. Large parts of the software has been developed in the context of Scrum.

In general, one can say that the aspect of agility applies to the cluster. Even if there is still more potential for expansion, a lot has been achieved to make the system more agile.

10.1.2 Operability

The goal of this aspect is that the system can be controlled at any time. The Cloud Native Plugin plays a significant role in fulfilling this operability aspect.

Since the plugin is present on every node, the functions provided by it are on every Renew instance in the cluster. In terms of operability, the plugin main purpose is to provide control functions. These include adding and removing plugins, uploading and starting nets, and having complete control over the simulation. The functions are accessed through a Java Spring instance provided by the Cloud Native Plugin.

Java Spring provides an HTTP interface for this purpose. This allows easy access to a worker instance from the outside. It must be mentioned here that "external" in this context means, being in the same network as the cluster.

To evaluate how successful the integration of the operability aspect was, it makes sense to look at the workflow of a plugin upload before and after the integration of the Cloud Native Plugin. Before the integration, there were two options to add a plugin on a worker instance. One was to provide a complete new Docker image containing the desired plugin. The image would then have to be deployed before the worker node was initialized.

The other option was to add it manually via the Renew CLI, which would require direct control of the worker instance. After integrating the Cloud Native Plugin, is now possible to use the following workflow. Required for that is knowledge about the IP address and Port of the worker, which can be found in the RenewKube Manager. If the address is known it is now possible to make a simple HTTP POST request to upload the desired plugin as a .jar file to the worker instance, which is followed by another POST request to start the plugin.

From the workflow used before and after the integration, it is clear that the Cloud Native Plugin has added to the operability aspect. There is external access to the individual worker instances, to manage and control them at runtime. All workflows to control the worker instances look similar to the one presented.

In conclusion, integrating the Cloud Native Plugin has greatly improved the operability aspect of the cluster. After the integration, the cluster fulfills the Cloud Nativity operability aspect.

10.1.3 Observability

The observability aspect is about monitoring a system. This can be done in different ways. For the Cloud Native Plugin this means that several ways to monitor a worker instance are now possible.

First, the plugin provides an HTTP endpoint where log information about a worker instance can be queried, wich can be done from any Renew instance where the Cloud Native Plugin is present. Another way how the Cloud Native Plugin implements the observability aspect is by providing health metrics. Here, information about the running Renew instance and the host system are provided. This includes CPU utilization, RAM utilization, memory usage, runtime, and performance status. Furthermore, it is possible to display the described information in a graphical user interface, which is implemented with Spring Boot Admin. This third-party software from the Spring Ecosystem retrieves the health information and provides it via a separate HTTP endpoint.

In addition to this provided informationen a StatusCollector Plugin was created. This plugin was developed in conjunction with Cloud Native Plugin and is a connector between the Cloud Native Plugin and other Renew plugins. It allows any Renew plugin with the StatusCollector as a dependency to pass its custom status messages to the health endpoint. Using a separate plugin to add this kind of information has the advantage that each plugin developer can decide which information should be displayed and that there is no direct dependency on the Cloud Native Plugin.

What is still missing in the current state of development is a central point where information about the various nodes can be queried. For this, the Status Monitor component from the technical realization of the Mushu architecture would have to be conceptualized and integrated, which is context of another thesis.

Overall the Cloud Native Plugin represents a significant improvement in the observability aspect of the cluster.

10.1.4 Resilience

The resilience aspect has not been directly implemented by the Cloud Native Plugin. Instead, integrating the Cloud Native Plugin now offers the possibility to react to single points of failure.

For example, this is possible through the use of the provided HTTP endpoints. Before the integration, the Java RMI of the Distribute Plugin was a central point of errors. However, all worker nodes can now be addressed and controlled directly via the HTTP interface. So the whole system can still be controlled and administered, even if the Java RMI is no longer available.

Currently, it is still impossible to use the cluster without the Distribute Plugin and the Java RMI. Work has been done on the implementation of simple communication via Apache Kafka, but this is not integrated into the current technical realization of the Mushu architecture. In addition, complex communication would have to be implemented to guarantee even more reliability, which is currently also context of another thesis.

Since many plugins interact together in Renew, error possibilities are almost unlimited. Therefore, it is not possible to avoid all errors, and it is more important is to ensure functionality in case an error occurs. An automated response to a failing node is not implemented in the Cloud Native Plugin. However, the plugin provides all functionalities needed to detect a failure and reset a working instance. It would be possible that the RenewKube Manager can then use the provided functionality to reset an instance in case of an error.

It can be seen that the integration has improved the resilience aspect, but there is still more work to be done to have a completely resilient system. Possible approaches to further improving the reliability are described in chapter 11.2.

10.2 Working Method

Within the scope of this thesis's development, I worked together with different participants. Thereby, different working methods and work structures were used. Therefore, this section evaluates how the different working methods have affected this thesis. In particular, the focus here is on the work done within the AOSE 22/23 project.

Although the Mushu architecture was mainly created in (Röwekamp 2023a), some components were also created within the AOSE project context. Thus, for example, already in the AOSE 20/21 project, the Cloud Native Plugin covered in this thesis was developed. In that project, I came first into contact with the Mushu architecture and played a central role in developing the Cloud Native Plugin.

Furthermore, there was work and cooperation on implementing technical realization of Mushu at several points. For example, a task force was formed to work on elements of the implementation in preparation for Jan Henrik's disputation, which was also summarized in chapter 5.

Because of the history of Mushu in the AOSE project context, the idea came up to combine the development of this thesis with the AOSE 22/23 project. That is why I took on the role as the product owner and led a small team in the context of Scrum in the AOSE 22/23 project.

Within the scope of my role, it was my task to set up a plan for the integration of the Cloud Native Plugin into the current version of the technical realization of the Mushu architecture. Furthermore, it was my task to guarantee that a general level of knowledge about Mushu was present in the team and to create tasks for all team members corresponding to their skills.

Even though Scrum leaves it up to the developers themselves to do the concrete implementations, I had influence in all implementations. This usually meant that I created exemplary implementations for a problem, and the team then carried out similar implementations. So, regarding Scrum, I also partially took on the developer role, which allowed me to take direct control in the development process.

The general work done was very successful. By distributing the workload, it was possible to work on different topics at the same time. This was especially helpful when the cluster recovery was an unexpected major problem.

Regarding the teaching purpose of the project, the team members learned to work on modern technologies such as Java Spring, Docker, and Kubernetes. In addition, the interest in Mushu has been passed on to the next generation, in the hope that they will continue to work on the Mushu architecture and its implementation.

Another positive point was the collaboration with other graduating students. In particular with those which also work in the context of Mushu. Thus one could profit from the knowledge of others and during the development one could pay attention to the implementations and possible connections to other thesis.

Nevertheless, a few negative things were noticeable when working with the AOSE project team. First, the introduction to the Mushu topic created a significant overhead. Due to different levels of knowledge in the team, much time had to be spent to bring the team to the same level of understanding. Furthermore, much time was spent in meetings and coordination due to the whole project structure.

In the individual case of my team, the Scrum Master unexpectedly quit the project. This led to the fact that the team lost a member, and I primarily took over the tasks. Since such a scenario is very individual, this does not necessarily have to be true for other circumstances. However, it is still important to consider such possibilities in the time planning phase of a thesis.

In summary, I still recommend further work in the context of Mushu to be included in the AOSE project. Despite all the overhead, it helped me to work in a team when developing the results of this thesis. Software development is mostly teamwork, and the topic around Mushu is the perfect opportunity to work in a small team and modern technologies in a distributed software environment. Looking back, I would make the same decision to work together on the development with the AOSE project team.

10.3 Compliance with Requirements

At the beginning of this thesis, requirements were set up to be considered in the development. Each requirement is addressed and evaluated to see if it has been met in this section.

The first requirement R1 was that the current state of development and documentation of the Cloud Native Plugin is recorded. This had the background to create a starting point for the integration. In addition, requirement R2 was set up, which states that a suitable version of the Cloud Native Plugin should be determined for the integration. Both requirements were dealt with in the context of the prototype in chapter 4. The result of the prototype was a summary of the previously created documentation and its supplementation with missing content. By collecting information and supplementing it, a complete documentation of the plugin was obtained, which thus fulfills requirement R1. In addition, current implementation variants of the Cloud Native Plugin were considered, and a suitable variant for the integration was determined, which fulfills requirement R2.

The next requirement R3 deals with the status of the technical realization of the Mushu architecture. It should be determined which components must be adapted so that the integration of the Cloud Native Plugin can take place. In addition, it was stated in requirement R4 that the necessary adjustments would be carried out. Both requirements were addressed in the prototype in chapter 5. This prototype describes a work phase in which the determined changes the components of the technical realization were carried out, which was done in cooperation with Jan Henrik Röwekamp in the context of his dissertation. The result of this prototype gives an overview of the status of different components of the technical realization of Mushu and the changes made at the respective components. Therefore, this prototype fulfills both of the requirements R3 and R4.

After that, requirements for the production environment were established. For requirement R5, it should be determined in which state the current technical implementation the cluster was. Further, requirement R6 demands that if the cluster is not usable, to restore it and that the documentation of the usage of it is sufficiently updated to facilitate later use. Both requirements are handled in the prototype of chapter 6. In that chapter, the current state of the cluster was documented and an overview of the previous work done was given. Afterward, the reestablishment of the environment is accomplished. Finally, the former and the reworked documentation is addressed. The requirement R5 is fulfilled thus to the beginning of the prototype, it was determined that the environment's condition was insufficient because of a system wipe. It follows the completion of requirement R6, with the reestablishment of a usable state of the cluster, and the updating of the provided instructions manual.

Requirements R7 and R8 deal with integrating the Cloud Native Plugin into the environment. Here, requirement R7 focuses on ensuring that integration occurs in the latest technical realization of the Mushu architecture. In addition, requirement R8 focuses on the documentation of the integration process to create a guide for subsequent integrations. Both requirements are addressed in the prototype in chapter 7. In the prototype, all steps are described, which were necessary to integrate the Cloud Native Plugin into the technical realization of the Mushu architecture. The step-by-step approach and documentation in the prototype fulfill requirement R7. The presentation of the result at the end of the prototype and the description of the implementation of the integration also fulfill requirement R8.

After the integration, the question of the usability of the provided functions was raised. Requirement R9 states that it should be evaluated how the functions can best be used in the environment and that usage of the Cloud Native Plugin should be documented accordingly. In the prototype in chapter 8, a discourse with different approaches occurred. The approach of an abstraction class was evaluated as a solution, and examples were created as reference nets. The discussion in the prototype fulfills the requirement for a solution for the usability of the functions in the cluster context. The created examples in the prototype fulfill mainly the purpose of documentation and understanding, which fulfills the second part of the requirement R9.

The last requirement is R10, states that a use case should be developed to demonstrate the new functionality and validate the integration of the Cloud Native Plugin. For this purpose, in the prototype in chapter 9, an all-in-one example was created that uses the functions created in chapter 8 and constructs a coherent example that can be used in the restored environment from chapter 6. Since the prototype deals with the development of the use case and describing an exemplary simulation process, requirement R10 is fulfilled.

Consequently, all developed requirements in chapter 3 were fulfilled in this thesis.

Chapter 11

Conclusion

11.1 Summary

The first chapter of the thesis explains the motivation for working on the topic on the integration of the Cloud Native Plugin. In addition, the goal of this thesis is described, and the structure is presented.

It is followed by the basics chapter 2 in which the fundamental knowledge necessary for understanding this thesis is presented. Technologies such as Docker, Kubernetes, and Java Spring are explained in the context of this thesis. In addition, theoretical concepts such as Cloud Nativity and Petri Nets are discussed. Also, the simulation software Renew and the associated Mushu architecture are described since both play a central role in this thesis. The basics are followed by the definition of requirements in chapter 3 to be fulfilled within the scope of this thesis.

In chapter 4, there is an overview of the current implementation of the Cloud Native Plugin prior to the integration. Information from a wide variety of previous work is compiled and reviewed. In addition, general information and required documentation were added if incomplete. A closer look at the plugin's file structure was taken, and the functions around the plugin upload were documented.

This is followed by chapter 5, which sets the focus on the technical realisation of Mushu and its components. It describes the work phase in preparation for Jan Henrik's disputation. Work was done on the various Renew plugins, which all had something to do with the final implementation and integration of the Mushu architecture. Therefore, a closer look is taken at the relevant work for the Cloud Native integration.

Chapter 6 covers the setup of a production environment. This environment is used to develop and test the technical implementation of the Mushu architecture and is the basis for integrating the Cloud Native Plugin. In this prototype, the current state of the environment and its documentation are recorded. The restoration of a usable state follows this after it has become unusable due to software updates. Instructions are revised and reworked to simplify subsequent setups. In addition, minor environmental upgrades are made to enable integration of the Cloud Native Plugin. The result is a working production environment that can serve as the basis for further integration. Lastly, revised and supplemented documentation for setting up and using the environment were created.

This is followed by chapter 7 in which the actual integration of the Cloud Native Plugin is carried out. In this prototype, a step-by-step integration of the Cloud Native Plugin into the previously restored production environment is described. The focus is on the steps required to integrate a subcomponent into the Mushu architecture. As a result, a current state of the production environment is presented, in which the Cloud Native Plugin has been integrated as a component.

Chapter 8 focuses on the usability of the Cloud Native Plugin in the cluster. In this prototype, various approaches are discussed to make the functions provided by the plugin more easily usable. The discussion results in an abstraction class created to execute HTTP requests directly from a developed reference net. For documentation and better understanding, example reference nets were created that can be employed to utilize the functions of the Cloud Native Plugin directly from a simulation context.

The last prototype in chapter 9, deals with the validation the integration of the Cloud Native Plugin. For this purpose, a coherent example is created that uses the reference nets developed in section 8.4.2 and links them to an already created example from RenewKube. The result is a simulation that uses the production environment created in chapter 6 to manipulate a remote Renew instance to meet the requirements for its distributed simulation. By successfully running the simulation, the integration process of the Cloud Native Plugin into the environment has been validated.

In the penultimate chapter, the complete integration is then evaluated. For this purpose, reference is made to the requirements established in chapter 3. Each requirement is individually addressed to analyze how well each requirement is fulfilled. It follows the conclusion which includes this summary and an outlook on further topics that resulted from this work or stand in a strong connection to it.

11.2 Outlook

With this work, the integration of the Cloud Native Plugin on the cluster was carried out. Furthermore, the Mushu architecture has been realized one step closer to completion. However, several components are still missing for implementing the complete Mushu architecture on the cluster. Since the integration of all missing components would have gone beyond the scope of this thesis, the components would still have to be developed and integrated in further work.

In this section, the currently ongoing work on related topics are outlined. In addition, a perspective is given on what further work is conceivable in the context of Mushu and what the next logical step would be to complete the technical realization of Mushu.

Ongoing Related Work to the Mushu Architecture

At the time of writing, various elements of the technical realization of the Mushu architecture are being worked on. This subsection will briefly summarize which elements will be completed shortly.

The AOSE project team will continue with the work on the technical realization of Mushu. They will work on components of the Mushu architecture, started with this thesis. The significant focus is on the complete restoration of the production environment. To achieve that, the complete integration of RenewKube must occur, which was only partially in the scope of this thesis. Bringing RenewKube in a working state is the primary goal of the project team. Furthermore, the team is currently working on creating VM images that can be used for development purposes. These have the advantage that no access to the university lab is necessary to work on technical realization.

Parallel to working on the project, two bachelor theses are being written in the context of Mushu. On the one hand, the status monitor, also mentioned in this thesis, is examined more closely. Thereby it is discussed which information is to be displayed about which components and in which granularity. The theoretical focus is set on the observability aspect of Cloud Nativity.

On the other hand a different thesis deals with elaborating the complex communication of the technical realization of the MUSHU architecture. In particular, the focus is on the orchestrator component and makes substantial reference to the simple communication with Apache Kafka developed in (Senger 2021). In the context of that work, the Distribute Plugin is exchanged as a communication medium, and simple and complex communication is integrated into the technical realization.

11.2.1 Other Potential Topics Related to Mushu

If a state is reached in which the target version of the technical realization described in section 2.8.2 is complete, MUSHU can be used for more complex distributed simulations. For example, one can use the Distribute Plugin developed in RENEW to evaluate what performance improvement the MUSHU architecture achieves in terms of simulation speed. Another use case would be the Sidler game developed for MULAN, where it would be possible to provide each player with their own RENEW instance to achieve proper distributed simulation.

In addition, a solution for rights management should be found. As mentioned in chapter 6, certain elements need admin permissions to work without problems. Since RENEW is in the context of reference nets and thus in combination with MUSHU in the area of remote code execution, it must be guaranteed here that the access is regulated. Since the implementation runs on university computers and in the university network, this topic needs to be addressed.

The ability of MUSHU to scale dynamically, to run RENEW on one or more high-performance servers, and provide access over a web interface to communicate with it, makes it possible to think of RENEW as a service. Several approaches for a possible web interface for RENEW have already been discussed, and MUSHU would be a good choice for a powerful backend. The advantage of RENEW as a service is saving the manual RENEW installation in the teaching application of RENEW. So RENEW can be used directly from the web without setup and installation.

Furthermore, testing a version of MUSHU with other modern technologies in distributed software solutions is possible. It would be interesting to see if it is possible to set up an automatic scaling MUSHU system based on a provider such as Amazon AWS. This would enable us to perform high-performance analyses of MUSHU and test the reliability of the technical implementation with industrial implementations standards.

In a small context, one could also consider transferring the created examples in chapter 8 to NetComponents. This would make it even easier for reference net creators to use the Cloud Native Plugin functions as ready-made building blocks for RENEW.

Bibliography

Cabac, Lawrence (Apr. 2010). “Modeling Petri Net-Based Multi-Agent Applications”. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik. URL: `https://ediss.sub.uni-hamburg.de/handle/ediss/3691`.

Docker (2023a). *Docker Overview.* URL: `https://docs.docker.com/get-started/overview/` (visited on 03/07/2023).

— (2023b). *Install Docker Engine on Ubuntu.* URL: `https://docs.docker.com/engine/install/ubuntu/` (visited on 03/16/2023).

Drumond, Claire (2023). *Was ist Scrum?* URL: `https://www.atlassian.com/de/agile/scrum` (visited on 03/17/2023).

Duvigneau, Michael (Oct. 2009). “Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen”. `https://ediss.sub.uni-hamburg.de/handle/ediss/3023`. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik. URL: `https://ediss.sub.uni-hamburg.de/handle/ediss/3023`.

Edmeier, Johannes (2023). *Spring Boot Admin Reference Guide.* URL: `https://codecentric.github.io/spring-boot-admin/2.5.1/#_what_is_spring_boot_admin` (visited on 03/07/2023).

Foundation, Cloud Native Computing (2023). *The CNCF Cloud Native Definition v1.0.* URL: `https://github.com/cncf/foundation/blob/main/charter.md` (visited on 03/07/2023).

Fowler, Martin (2014). *Inversion of Control Containers and the Dependency Injection pattern.* URL: `https://martinfowler.com/articles/injection.html` (visited on 03/07/2023).

Garrison, J. and K. Nova (2017). *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment.* O’Reilly Media. ISBN: 9781491984253. URL: `https://books.google.de/books?id=1Fk7DwAAQBAJ`.

GitLab (2023). *GitLab Container Registry.* URL: `https://docs.gitlab.com/ee/user/packages/container_registry/` (visited on 03/16/2023).

IBM (2023). *What is Kubernetes?* URL: `https://www.ibm.com/topics/kubernetes` (visited on 03/07/2023).

Jan Henrik Röwekamp (2018). “Investigating the Java Spring Framework to Simulate Reference Nets with Renew”. In: *Algorithms and Tools for Petri Nets, Proceedings of the Workshop AWPN 2018, Augsburg, Germany*, pp. 41–46.

Jan Henrik Röwekamp Marvin Taube, Yannik Stahl (2023). *Generelle Informationen zu tecnischen Realisierung von Mushu.* URL: `https://tgipm.informatik.uni-hamburg.de/confluence/display/AOSE21AVS/Generelle+Informationen` (visited on 03/16/2023).

Janneck, Jan Robert (Mar. 2021). “Modularizing a Plugin System Using Java Modules: Application to a Medium-Sized Open-Source Project”. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Kubernetes (2023a). *Kubernetes Concepts Overview.* URL: `https://kubernetes.io/docs/concepts/overview/` (visited on 03/07/2023).

Kubernetes (2023b). *Kubernetes Documentation - Getting started.* URL: https://kubernetes.io/docs/setup/ (visited on 03/16/2023).

Kummer, Olaf (2002). *Referenznetze.* Berlin: Logos Verlag. ISBN: 978-3-8325-0035-1. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.

Kummer, Olaf, Frank Wienberg, and Michael Duvigneau (Oct. 2002). *Renew – The Reference Net Workshop.* Available at: http://www.renew.de/. Release 1.6. URL: http://www.renew.de/.

Kummer, Olaf, Frank Wienberg, Michael Duvigneau, et al. (Nov. 2020). *Renew – The Reference Net Workshop.* Release 2.5.1. URL: http://www.renew.de/.

Marvin Taube, Patrick Mohr (2021). "Implementation of Cloud Native Renew based on Java Spring". In: *Report PAOSE-Project WS 20/21*, pp. 83–92. URL: https://git.informatik.uni-hamburg.de/tgi/teaching/ws20/aose20/aose20-bericht.

Microsoft (2023). *Comparing Docker containers with virtual machines.* URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined (visited on 03/07/2023).

Mirantis (2023). *Dockerd as a compliant Container Runtime Interface for Kubernetes.* URL: https://github.com/Mirantis/cri-dockerd (visited on 03/16/2023).

Petri, Carl Adam (1996). "Nets, Time and Space". In: *Theoretical Computer Science* 153.1–2. Ed. by Maurice Nivat and Grzegorz Rozenberg, pp. 3–48.

Renew - The Reference Net Workshop (2023). URL: http://www.renew.de/ (visited on 03/17/2023).

Rölke, Heiko (1999). "Modellierung und Implementation eines Multi-Agenten-Systems auf der Basis von Referenznetzen". Diplomarbeit. Universität Hamburg, Fachbereich Informatik.

Röwekamp, Jan Henrik (2023a). "Skalierung von nebenläufigen und verteilten Simulationssystemen für interagierende Agenten". PhD thesis. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/10040.

— (2023b). *Webpage for RenewKube.* URL: https://paose.informatik.uni-hamburg.de/paose/wiki/RenewKube (visited on 03/16/2023).

Röwekamp, Jan Henrik and Daniel Moldt (2019). "RenewKube: Reference Net Simulation Scaling with Renew and Kubernetes". In: *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings.* Ed. by Susanna Donatelli and Stefan Haar. Vol. 11522. Lecture Notes in Computer Science. Springer, pp. 69–79. ISBN: 978-3-030-21570-5. URL: https://doi.org/10.1007/978-3-030-21571-2_4.

Röwekamp, Jan Henrik, Marvin Taube, et al. (2021). "Cloud Native Simulation of Reference Nets". In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference).* Ed. by Michael Köhler-Bußmeier, Ekkart Kindler, and Heiko Rölke. Vol. 2907. CEUR Workshop Proceedings. CEUR-WS.org, pp. 85–104. URL: http://ceur-ws.org/Vol-2907.

Schwaber, Ken and Jeff Sutherland (2023). *The 2020 Scrum Guide.* URL: https://scrumguides.org/scrum-guide.html (visited on 03/17/2023).

Senger, Alexander (Nov. 2021). "Erweiterung des Renew Petrinetz-Simulators um cloud-native Systemkomponenten". Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Simon, Michael (Mar. 2014). “Concept and Implementation of Distributed Simulations in Renew”. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Spring, Java (2023). *Why Spring?* URL: `https://spring.io/why-spring` (visited on 03/07/2023).

Sutherland, Jeff and Scrum Inc (2023). *The Scrum At Scale® Guide.* URL: `https://www.scrumatscale.com/scrum-at-scale-guide-online/` (visited on 03/17/2023).

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 20. März 2023

Marvin Taube