

ESCUELA POLITÉCNICA SUPERIOR DE MONDRAGON UNIBERTSITATEA
MONDRAGON UNIBERTSITATEKO GOI ESKOLA POLITEKNIKOA
MONDRAGON UNIVERSITY FACULTY OF ENGINEERING

Trabajo presentado para la obtención del título de

Titulua eskuratzeko lana

Final degree project for taking the degree of

Master universitario en Análisis de Datos, Ciberseguridad y Computación en la Nube

Datuen Analisia, Zibersegurtasuna eta Hodei-Konputazioko Unibertsitate Masterra

Master Degree in Data Analysis, Cybersecurity and Cloud Computing

Título del Trabajo *Lanaren izenburua* Project Topic

**MINIMIZING USER INTERVENTION IN THE DEPLOYMENT OF THE
ARROWHEAD FRAMEWORK**

Autor *Egilea* Author

ALEXANDER MEDELA AYARZAGUENA

Curso *Ikasturtea* Year

2021/2022

Título del Trabajo *Lanaren izenburua* Project Topic

**MINIMIZING USER INTERVENTION IN THE DEPLOYMENT OF THE
ARROWHEAD FRAMEWORK**

Nombre y apellidos del autor

Egilearen izen-abizenak

Author's name and surnames

MEDELA AYARZAGUENA, ALEXANDER

Nombre y apellidos del/los director/es del trabajo

Zuzendariaren/zuzendarien izen-abizenak

Project director's name and surnames

PÁL VARGA

LARRINAGA, FELIX

Lugar donde se realiza el trabajo

Lana egin deneko lekua

Company where the project is being developed

Budapest University of Technology and Economics

Curso académico

Ikasturtea

Academic year

2021/2022

☒ El autor/la autora del Trabajo Fin de Grado, autoriza a la Escuela Politécnica Superior de Mondragon Unibertsitatea, con carácter gratuito y con fines exclusivamente de investigación y docencia, los derechos de reproducción y comunicación pública de este documento siempre que: se cite el autor/la autora original, el uso que se haga de la obra no sea comercial y no se cree una obra derivada a partir del original.

Gradu Bukaerako Lanaren egileak, baimena ematen dio Mondragon Unibertsitateko Goi Eskola Politeknikoari Gradu Bukaerako Lanari jendeaurrean zabalkundea emateko eta erreproduzitzeko; soilik ikerketan eta hezkuntzan erabiltzeko eta doakoa izateko baldintzarekin. Baimendutako erabilera honetan, egilea nor den azaldu beharko da beti, eragotzita egongo da erabilera komertziala baita lan originaletatik lan berriak eratortzea ere.

Mondragon University

Master's Degree in Data analysis, Cybersecurity and Cloud Computing

▪ **Master Thesis** ▪

Cloud Computing

Minimizing User Intervention in the Deployment of the Arrowhead Framework

March 2022

Author

Alexander Medela Ayarzagüena

Tutors

Felix Larrinaga

Pál Varga

Abstract

Every day the use of the IoT devices in the business is increasing and the interoperability of these devices gets harder as the amount of these devices increases. To solve this problem the Eclipse Arrowhead project created the Arrowhead Framework, a tool to ease IoT interoperability. And this framework has become a popular choice and the use of it is increasing over time.

The problem of this framework is the complexity to deploy it, which makes the usage of the framework difficult for the average users.

This document will analyze different techniques to automate the deployment. First the deployment of each service is automated using docker, then the cloud deployment is automated using Kubernetes. There is also an option to deploy the cluster on the Azure and Google Cloud services that was implemented using terraform. And finally, all these tools are put together by a GUI based on python giving the option to deploy the cloud however the user wants.

Cada día aumenta el uso de los dispositivos IoT en las empresas y la interoperabilidad de estos dispositivos se hace más difícil a medida que aumenta la cantidad de estos dispositivos. Para resolver este problema, el proyecto Eclipse Arrowhead creó el Arrowhead Framework, una herramienta para facilitar la interoperabilidad del IoT. Este framework se ha convertido en una opción muy popular y su uso está aumentando con el tiempo.

El problema de este framework es la complejidad para desplegarlo, lo que dificulta su uso para los usuarios medios.

Este documento analiza diferentes técnicas para automatizar el despliegue. Primero se automatiza el despliegue de cada servicio utilizando docker, luego se automatiza el despliegue en la nube utilizando Kubernetes. También hay una opción

para desplegar el clúster en los servicios de Azure y Google Cloud que se implementa utilizando terraform. Y por último, todas estas herramientas están reunidas por una GUI basada en python dando la opción de desplegar la nube como el usuario quiera.

Egunero enpresetan IoT gailuen erabilera areagotu egiten da eta gailu horien elkarreragingarritasuna zaildu egiten da gailu horien kopurua handitu ahala. Arazo hori konpontzeko, Eclipse Arrowhead proiektuak Arrowhead Framework-a sortu zuen, IoT elkarreragingarritasuna errazteko asmoz. Framework hau aukera ezaguna bihurtu da eta denborarekin bere erabilera areagotzen ari da.

Framework honen arazoa zabaltzeko konplexutasuna da, eta horrek zaildu egiten du erabiltzaile arruntek erabiltzea.

Dokumentu honetan automatizatzeko teknika desberdinak eztabaidatuko dira. Lehenik eta behin zerbitzu bakoitzaren hedapena automatizatu egiten da docker erabiliz, gero hodeian inplementazioa Kubernetes erabiliz garatu da. Terraform erabiliz inplementatu da, klusterra Azure eta Google Cloud zerbitzuetan zabaltzeko aukera. Eta, azkenik, tresna horiek guztiak Python-en oinarritutako GUI batek biltzen ditu erabiltzaileak nahi duen moduan framework-a erabiltzeko aukera emanez.

Table of Contents

| | |
|---|-----------|
| 1. INTRODUCTION | 1 |
| 1.1 BACKGROUND | 2 |
| 1.2 PROBLEM DESCRIPTION | 3 |
| 1.3 OBJECTIVE | 3 |
| 1.4 SCOPE | 4 |
| 1.5 RESOURCES AND REQUIREMENTS | 5 |
| 1.6 WORKING PLAN | 5 |
| 1.6.1. <i>Project deadlines</i> | 5 |
| 1.6.2. <i>Project phases</i> | 6 |
| 1.6.3. <i>Weekly logbook</i> | 6 |
| 1.6.4. <i>Code development</i> | 7 |
| 1.6.5. <i>Gantt</i> | 7 |
| 2. PROJECT DEVELOPMENT | 9 |
| 2.1 LEARNING THE ARROWHEAD FRAMEWORK | 10 |
| 2.1.1. <i>Local cloud automation</i> | 10 |
| 2.1.2. <i>Core systems workflow</i> | 15 |
| 2.1.3. <i>Arrowhead compliance systems</i> | 16 |
| 2.1.4. <i>Arrowhead compliance verification</i> | 16 |
| 2.2 LEARNING THE ARROWHEAD FRAMEWORK DEPLOYMENT | 18 |
| 2.2.1. <i>Docker based deployment (Only for test)</i> | 19 |
| 2.2.2. <i>Debian installer-based deployment</i> | 19 |
| 2.2.3. <i>Fully manual deployment</i> | 20 |
| 2.2.4. <i>Deploying the first network</i> | 20 |
| 2.2.5. <i>Testing the first network</i> | 23 |
| 2.3 ARROWHEAD FRAMEWORK AUTOMATION | 25 |
| 2.3.1. <i>Automating the systems deployment</i> | 25 |
| 2.3.2. <i>Automating the cloud deployment</i> | 27 |
| 2.3.3. <i>Graphical user interface</i> | 35 |
| 2.4 IMPLEMENTING CLOUD PROVIDERS | 36 |
| 2.4.1. <i>Infrastructure as code tools</i> | 36 |
| 2.4.2. <i>Cloud providers</i> | 36 |
| 2.4.3. <i>Implementation</i> | 37 |
| 2.4.4. <i>Graphical user interface</i> | 37 |
| 2.5 PIPELINE WORKFLOW | 39 |
| 2.5.1. <i>Implementation</i> | 39 |
| 3. PROJECT RESULTS | 41 |
| 3.1 CHANGES DURING THE PROCESS | 42 |
| 3.2 PRODUCT | 43 |
| 3.2.1. <i>Using an existing cluster</i> | 47 |
| 3.2.2. <i>Creating a new cluster</i> | 48 |
| 3.2.3. <i>The GUI deployment</i> | 50 |
| 3.3 PRODUCT TEST | 51 |

| | |
|----------------------------|-----------|
| 3.3.1. <i>Validation</i> | 51 |
| 3.3.2. <i>Verification</i> | 58 |
| 4. CONCLUSIONS | 59 |
| 4.1 CONCLUSIONS | 60 |
| 5. FUTURE LINES | 63 |
| 5.1 FUTURE LINES | 64 |
| ANNEX I | 66 |
| ANNEX II | 67 |
| BIBLIOGRAPHY | 68 |

List of Figures and Tables

| | |
|--|----|
| FIGURE 1.1: PROJECT GENERAL DEADLINES | 6 |
| FIGURE 1.2: NOVEMBER WEEKLY LOGBOOK..... | 7 |
| FIGURE 1.5: GANTT DIAGRAM..... | 8 |
| FIGURE 2.1: WORKFLOW OF THE CORE SYSTEMS | 15 |
| FIGURE 3.1: MARIADB DOCKER-COMPOSE BLOCK..... | 21 |
| FIGURE 3.2: SAMPLE OF THE ARROWHEAD DATABASE TABLES..... | 21 |
| FIGURE 3.3: SERVICE REGISTRY DOCKER-COMPOSE BLOCK | 22 |
| FIGURE 3.4: SERVICE REGISTRY LOGS..... | 22 |
| FIGURE 3.5: AUTHORIZATION LOGS..... | 23 |
| FIGURE 3.6: ORCHESTRATOR LOGS | 23 |
| FIGURE 3.7: TESTS FINAL RESPONSE | 24 |
| FIGURE 4.1: GITHUB REPO DOCKER IMPLEMENTATION | 26 |
| FIGURE 4.2: DOCKERFILE STRUCTURE | 27 |
| FIGURE 4.3: DOCKER SWARM CLUSTER..... | 29 |
| FIGURE 4.4: KUBERNETES CLUSTER | 31 |
| FIGURE 4.5: NOMAD CLUSTER | 32 |
| FIGURE 4.6: KUBERNETES AND NOMAD SIMILARITY | 32 |
| FIGURE 4.7: KUBERNETES RESOURCES..... | 34 |
| FIGURE 4.8: FIRST GUI MOCK-UP | 35 |
| FIGURE 5.1: TERRAFORM STRUCTURE..... | 37 |
| FIGURE 5.2: TERRAFORM PART MOCK-UP..... | 38 |
| FIGURE 6.1: PIPELINE WORKFLOW DEFINITION | 39 |
| FIGURE 7.1: GUI WELCOME PAGE..... | 44 |
| FIGURE 7.2: DATABASE SELECTION | 44 |
| FIGURE 7.3: SERVICES SELECTION | 45 |
| FIGURE 7.4: CLUSTER SELECTION | 46 |
| FIGURE 7.5: CREDENTIALS FOR EXISTING CLUSTER | 47 |
| FIGURE 7.6: CHOOSING CLOUD PROVIDER | 48 |
| FIGURE 7.7: MICROSOFT AZURE CREDENTIALS | 49 |
| FIGURE 7.8: GOOGLE CLOUD CREDENTIALS..... | 49 |
| FIGURE 7.9: DEPLOYMENT WINDOW | 50 |
| FIGURE 7.10: DATABASE TYPE | 52 |
| FIGURE 7.11: CHOOSING SERVICES | 52 |
| FIGURE 7.12: CONNECTING TO THE CLUSTER | 53 |
| FIGURE 7.13: CORRECT DEPLOYMENT LOGS | 54 |
| FIGURE 7.14: DEPLOYED RESOURCES..... | 55 |
| FIGURE 7.15: SERVICE REGISTRY LOGS | 55 |

| | |
|---|----|
| FIGURE 7. 16: DATABASE LOGS | 56 |
| FIGURE 7. 17: DATABASES | 56 |
| FIGURE 7. 18: AMBASSADOR RESOURCES | 57 |
| FIGURE 7. 19: EXPOSING CLUSTER SERVICE TO MacOS | 57 |
| FIGURE 7. 20: TEST LOGS | 58 |

1

Introduction

This chapter includes all the information related to the project specifications; from what it is about to how the project will be developed.

In the context of the Arrowhead Eclipse project, this project investigates and develop the best strategies to deploy the Arrowhead Framework in premises or over cloud infrastructure. The solutions obtained in the project will support users in implementing/deploying the Arrowhead framework with minimal intervention (plug and play solution).

The strategies will consider Docker, Cloud and Serverless infrastructures and technologies. Software engineering techniques such as Continuous Integration Continuous Deployment (CI/CD) will be also studied to support Arrowhead Framework developers in managing versions and updates.

The goal of the project is to evaluate existing technologies for servers and service easy deployment such as Docker, Cloud, Serverless or CI/CD. Further, to identify best practices and implement a demonstration methodology based on one of the use-cases defined in the project. Lastly, the method will be tested to establish a grade of improvement compared to earlier stages of the development process. Writing a technical report on the work performed and the achieved results.

1.1 Background

The project was developed in AITIA international Inc., a small business formed by the Budapest University of Technology and Economics (BME) and the Eötvös Loránd University (ELTE) lecturers and researchers. The project belongs to the Industrial IoT division where they work with the Eclipse Arrowhead project.

The business is located at 48-50 Csetz János street, Budapest, Hungary, 1039 and more information can be found on their web page <https://www.aitia.ai/>.

AITIA has participated into several European projects that works with the microservices problematic. Some of these projects are Arrowhead, Productive4.0, Mantis, etc.

As a result of all these projects the Arrowhead Eclipse came up, but these projects faced the complexity of the deployment of the framework. And this is where this project came up with the intention of making the deployment of the arrowhead framework easier.

This project started by using the Arrowhead Framework on its 4.1.3 version. Where it included the build .jar file for each system, some docker container images for testing, some scripts to test the correct functioning of the framework, and some simple examples to learn how it works.

The framework offers some more tools and functionalities, but this project was mainly started and based on the previously mentioned material.

1.2 Problem description

Every day the use of the IoT devices in the business is increasing and the interoperability of these devices gets harder as the amount of this devices increases.

To solve this problem the Eclipse Arrowhead project created the Arrowhead Framework, a tool to ease IoT interoperability. This framework has become a popular choice and the use of it is increasing over time.

The problem of this framework is the complexity to deploy it. This framework is based on a microservices cloud architecture, so to deploy these clouds previous knowledge about clouds, microservices, and the framework itself is needed. This makes the usage of the framework difficult for the average users.

1.3 Objective

The objective of this project is to develop a simple GUI that allows users to deploy the framework with minimal intervention. This GUI should allow the deployment of different arrowhead network configurations and infrastructures as well as the deployment on different cloud services. With the intention to achieve this main objective, minor objectives have been defined:

- Dockerize each service.
- Create an arrowhead compliance cloud with Kubernetes using core services.
- Add optional services to the Kubernetes resources.
- Create Azure and Cloud Kubernetes clusters.

- Put all the previous work together with a graphical user interface.

1.4 Scope

This project is evaluated by the Mondragon University by following their own criteria. in this case, the project will be evaluated with the next technical scopes:

- **M1N110:** *Define, design, and implement scalable, flexible, and resilient architectures to address existing issues and deploy existing applications faster.*
- **M1N303:** *Computer tools for the development of applications and operations (DevOps), both locally and in the cloud, to solve complex problems and carry out engineering projects, considering the commercial and industrial context.*
- **M1N401:** *Ability to work in multidisciplinary teams and in a multilingual environment (Basque / Spanish / English) and be able to communicate knowledge, procedures, results, and ideas related to data life cycle, cybersecurity, and development and operations, both orally and in writing.*

On the other hand, the next non-technical scopes will also be evaluated:

- **CTFM01:** *To write an adequate dissertation report and present the results in a clear way.*
- **CTFM02:** *The student has become part of the company, working with other people and showing a high level of efficiency and independence in the development of the final project.*

1.5 Resources and requirements

For the development of the project the following resources has been used:

- Personal Laptop (MacOS)
- External Monitor, keyboard, and mouse (due to problems with personal laptop)
- Internet connection
- Cloud services:
 - Google Cloud (Free account)
 - Microsoft Azure (Free account)
- Technologies:
 - Visual Studio Code
 - GitHub
 - Arrowhead Framework
 - Docker
 - Kubernetes
 - Nomad
 - Terraform

1.6 Working plan

For a correct project management and development, first the working plan was defined. This working plan defines all the workflow of the project, from the deadlines to how the project will be developed (Tasks, Milestones, Logbook, etc.).

1.6.1. Project deadlines

First, to develop the planning we need to have in mind the project deadlines. So, I defined a small and simple general Gantt diagram to define how to manage these general deadlines.

| General deadlines | October | | November | | December | | January | | February | | March | |
|--|---------|-------|----------|-------|----------|-------|---------|-------|----------|-------|-------|-------|
| | 1° Q. | 2° Q. | 1° Q. | 2° Q. | 1° Q. | 2° Q. | 1° Q. | 2° Q. | 1° Q. | 2° Q. | 1° Q. | 2° Q. |
| Midterm report | | | | | | | | | | | | |
| Project development | | | | | | | | | | | | |
| Project report | | | | | | | | | | | | |
| Propose tribunal and presentation date | | | | | | | | | | | | |
| Submit the project | | | | | | | | | | | | |
| Present and defend the project | | | | | | | | | | | | |

Figure 1.1: Project general deadlines

The figure 1.1 shows us how much time will be expended on each general deadline of the project.

1.6.2. Project phases

The project has been divided into different phases of development. Each phase has a different objective in the project. These are the phases of the project:

- Learn the Arrowhead Framework: The first phase of the project is to learn how the framework works to have a strong background which can be used to start working on the project.
- Learn the Arrowhead Framework deployment: This phase objective is to learn how to deploy the Arrowhead Framework and the different ways to deploy it.
- Arrowhead Framework automation: This phase analyses the deployment and how it can be automated using different tools. After the analysis, the best tool for each case is implemented.
- Cloud Providers: This phase analyses how to implement the cloud providers to the project and implement the using the best tool for each case.
- Pipeline: This phase defines how to implement a CI pipeline for the project.

1.6.3. Weekly logbook

To keep track of all the work and changes that has been done during the project, I decided to use a weekly logbook. This logbook helps to remember every week progress and heads you to the next steps.

| November | |
|----------------------|--|
| Week 1 (1Nov-5Nov) | Finished automating the test using docker containers. Finished deploying the arrowhead docker network with core services. Project management updates. |
| Week 2 (8Nov-12Nov) | Started analyzing container orchestrators for the deployment. Started defining the architecture solution for kubernetes. Started defining the architecture solution for docker swarm. Started learning about Nomad. |
| Week 3 (15Nov-19Nov) | Defined the architecture to be use with kubernetes and nomad. Started creating the deployment for kubernetes. Started creating the job for nomad. Desing the GUI for the user deployment. Started comparing infrastructure provisioning tools. |
| Week 4 (22Nov-26Nov) | First deployment of core services on kubernetes working. Started defining different kubernetes architectures for different use cases. Changes to the desing for the GUI. |

Figure 1.2: November weekly logbook

The figure 1.2 shows a small part of the used weekly logbook as example. In this case, we can see the progress done in the month of November. The full weekly logbook can be found on the [Annex I](#).

1.6.4. Code development

To help with the project code development GitHub will be used. This tool will help mainly with the version control, project management (Issues, Tasks, Milestones, etc.), and cloud storage of the code.

In the case of the Arrowhead Framework, they already have their project on GitHub, but this one will be developed on a different repository. This will be because this project will be a helping tool for the deployment of the framework and it's not a core part of it. And because the main repository it's already too heavy.

1.6.5. Gantt

Finally, to organize all the development of the project details, a Gantt diagram will be used. This Gantt will include all the tasks and milestones of the project and was defined after learning all the background so the task could be easily identified.

The project is divided into 5 different milestones. The defined milestones are the next ones:

- Docker: This milestone focuses on getting the docker containers of the Arrowhead Framework systems. This will contain all the task related to the docker container.
- Kubernetes: This milestone focuses on getting all the resources to deploy the Arrowhead Framework on Kubernetes. This will contain all task related with the Kubernetes deployment.
- Terraform: This milestone focuses on creating the Kubernetes cluster using terraform. This will contain all task related with the Kubernetes cluster creation with terraform.
- Graphical user interface: This milestone focuses on automating all the deployment using the resources from the previous milestones. This will contain all task related with the automation of the deployment.
- Documentation: This milestone focuses on the documentation of the repository.

All the task has been created based on these 5 milestones. In the figure 1.5 we can see a small part of the used Gantt diagram and the full diagram can be found in the [Annex II](#).

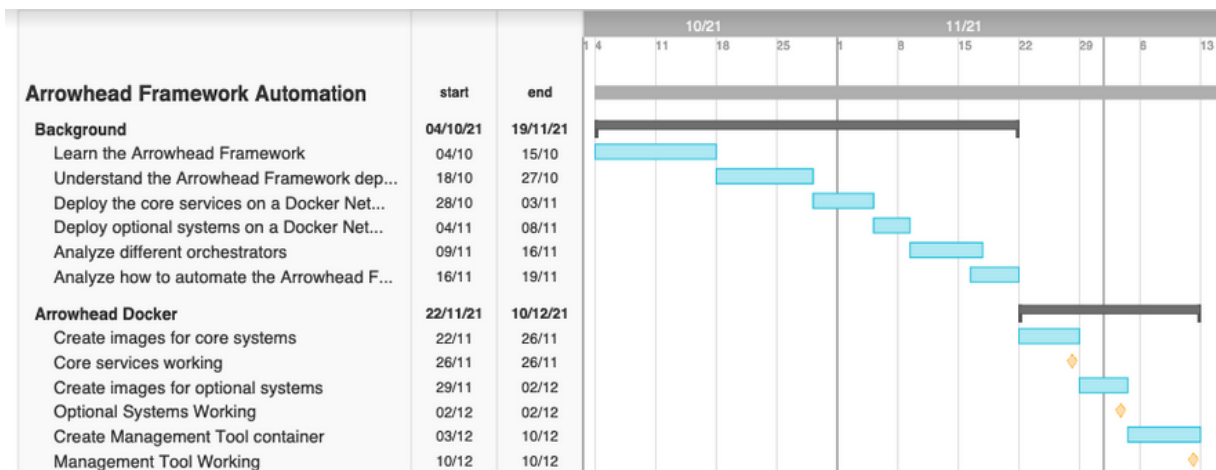


Figure 1.3: Gantt diagram

2

Project development

This chapter summarizes all the process of the work developed during the project. Starting with the understandings of the Framework, followed by the analysis of the automation and ending with the implementation of the automation.

2.1 Learning the Arrowhead Framework

To start with the project first we need to know about the Arrowhead Framework. What is it, its objective, how it works, etc. This will give us a solid base to start working with the project.

To learn about the Arrowhead Framework, I have used the (Delsing, 2017) book and (Varga, et al., 2016; Delsing, 2017) journal that can be found in the bibliography. These 2 data sources contain all the information need to have an strong background and even more advance information about the framework.

So first, the Arrowhead Framework objective is to enable IoT interoperability and integration. And this, is provided by abstracting the IoT components and their functionalities into services creating a microservices architecture.

This framework works with the concept of local networks or clouds that consist mainly of system of systems. And they established the concept of local clouds to fulfil automation system requirements concerning:

- Real-time properties.
- Security and integrity.
- Automation of functionalities.

In addition, local clouds can also provide other optional services adding new properties in the interest of building automation systems.

2.1.1. Local cloud automation

The Arrowhead Framework focuses on the next requirements to fulfill the local cloud automation.

- Service discovery
- Loosely coupled data exchange
- Authorization of service usage
- Orchestration of automation functionalities

The framework defines three different groups to fulfill these requirements: Assurance services (IA), Information Infrastructure services (II) and System

Management services (SM). And to create a minimal working cloud one service of each group is needed, these are the Service Discovery, Authorization, and Orchestrator.

Based on the three core systems we will obtain the local cloud automation, but we could add more features to the cloud by adding more services like the Event Handler, Gateway, Gatekeeper, etc.

In addition, to understand the arrowhead framework we will also need to understand the following concepts that builds the frameworks.

Framework base

This framework has been built using exiting protocols, standards and handle legacy systems and system of system compositions to fulfill the future new needs. It also includes design patterns, documentation templates and guidelines to ease the development, deployment, maintenance, and management of the framework.

The framework aims to make their users work in a common and unified approach.

Arrowhead services

In the arrowhead framework services are used to exchange information from provider system to a consumer system. And a resource is the nomenclature used to designate the information source.

The arrowhead local cloud should be able of consuming different type services and provide other services that fulfill more complex tasks. An Arrowhead service should also fulfill security, real-time operation, and reliability requirements.

Arrowhead systems

In the arrowhead context, a system provides and/or consumes services. A system can be at the same time the Service Provider of one or more services and the Service Consumer of several services.

System of systems

The arrowhead local clouds fulfill the 5 main characteristics of the system of systems:

- operational independence of its systems

- management in- dependence of the systems
- evolutionary development
- emergent behaviour
- geographic distribution

Meaning that the clouds created by the Arrowhead framework are also system of systems and they share the same properties.

Arrowhead framework systems

Among the systems that conform the Arrowhead Framework, there are several core systems that are marked as mandatory. The mandatory systems provide the core services previously mentioned and complies with the interoperability requirements of SOA cloud: the application service registration, the service discovery, the authorization, and the orchestration.

Core systems will be considered to have infrastructure support services in terms of system management and information security.

The aim of the core systems is to enable and support the application systems to exchange the information successfully.

This section introduces the elements of the core system and provides an overview of the system collaboration system, showing how the core system and the application system work together.

Core systems

The three mandatory systems mentioned earlier needed to stablish an Arrowhead local cloud are the next ones:

- Service registry system
- Authorization system
- Orchestration system

These systems are available in the Arrowhead framework and can be used to create a minimal arrowhead-compliant local cloud.

Service registry system

The service registry stores and manages all active production services in the arrowhead network. It is used to make sure all systems find each other even with dynamic endpoints.

All the systems in the arrowhead network that generate information for the network needs to publish their services into the service registry by using the service discovery service.

In the system of systems, the service registries support system interoperability using the ability to look up characteristics of the service producers.

In short, it allows the system to publish its own application services and find other application services.

Authorization system

The authorization system controls that only authorized consumers can access the service.

The system is formed by two service producers and a service consumer, and it stores a list with access rules for system resources. The Authorization Management service is the one with the which manages the access rules of each resource. And the Authorization Control service is the one who manages the access to specific resources by external services.

Orchestration system

The orchestration is one of the mandatory components of the Arrowhead framework and a primary component in SOA-based architectures.

This system is used to manage how systems are deployed and how they connect to each other. The orchestration is a system that supports the rest of the systems setups by providing coordination, and delivery services.

And its main functionality is to allow the reuse of existing services and systems to generate new services and functions.

Optional systems

Besides the core systems there are also optional systems to add more functionality to the arrowhead-compliant cloud. More of these systems are being developed over time and users could also create their own ones following the design pattern and templates. But, so far in the Arrowhead framework we can find the next optional systems:

- **Deployment system:**

The main objective of the deployment system is to manage the deployment of new systems and make sure authentication on the Arrowhead network. The provisioning system handles vendor authentication and identification and assigns specific certificates that are trusted on the Arrowhead network. This way, the configuration system can be seen as a support system for the connection of pre-assigned devices, thus making the deployment much faster and easier.

- **Gateway system:**

The Gateway system provides the functionality to connect different arrowhead-compliant clouds between them. This helps the clouds share their services and scale in a very simple way.

- **Gatekeeper system:**

The primary objective of the gatekeeper is to manage the external connection of the cloud. This helps to secure all the arrowhead-compliant clouds by preventing unwanted or unexpected connections.

- **Event Handler system:**

The event handler system adds the functionality for managing event. Event handlers receive events and redirects them to event consumers. The event handler also keeps log event in persistent storages, registers event producers/consumers, and applies filtering rules to event distributions. Event consumers are the components that manage the events.

- **User-system registry:**

The User System Registry contains the unique system identities of deployed systems in the Arrowhead network and is used to store user and system specific information. The orchestration system uses user system registration in conjunction with service registration. This composition enables the orchestration

system to provide a powerful picture of the possibilities of system-by-system composition that supports system collaboration.

- **Configuration system:**

The configuration system can store certain configuration values for another system. This configuration values are retrieved by the configuration system and applied by the application system when needed.

- **Meta service registry:**

The meta-service registry system saves information about services for offline access. This service registration feature stores other data such as, limits information, service usage, etc.

2.1.2. Core systems workflow

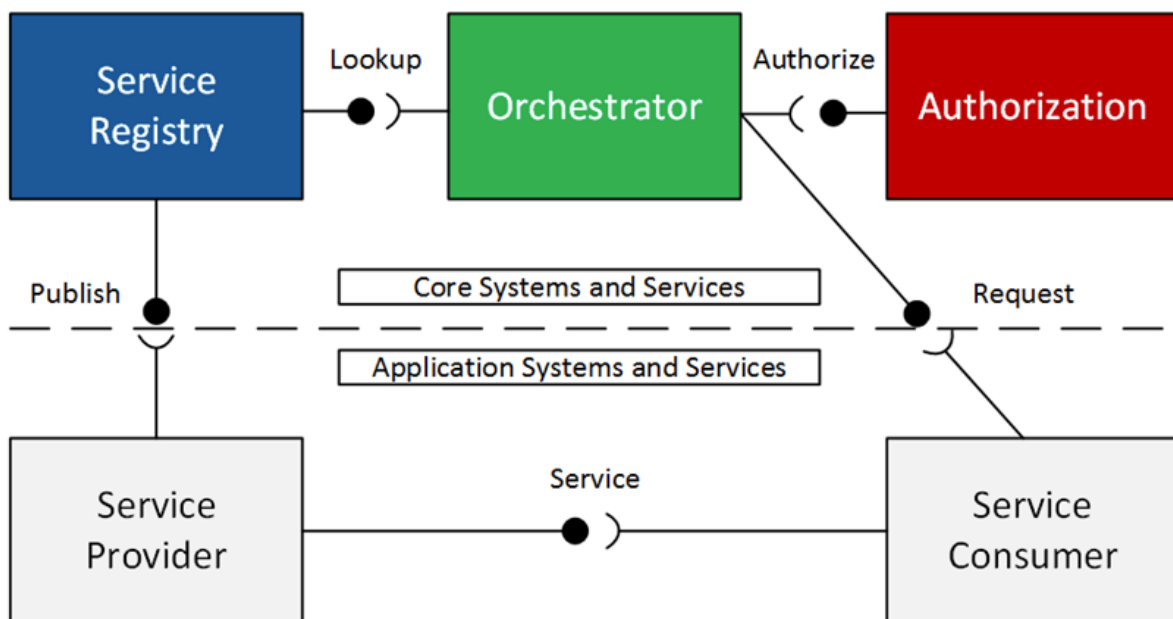


Figure 2.1: Workflow of the core systems

In the figure 2.2, we can see how the service consumer consumes a service from the service provider by following the next steps:

1. The service provider publishes a service to the service registry.
2. The consumer requests this service by using the orchestrator.

3. The orchestrator checks if the consumer is authorized to use this service by using the authorization service.
4. If the consumer is authorized the service consumer is connected to the requested service. If it is not authorized the request is denied.

2.1.3. Arrowhead compliance systems

Arrowhead systems that implement collaborative automation must conform to the principles of the Arrowhead Framework. The following steps must be fulfilled to make sure the cloud is arrowhead compliance:

- Follow Arrowhead Framework templates
- Adapt the legacy systems or create new systems following the arrowhead designs
- Run interoperability tests

Apart from all the available arrowhead design templates, system architects and developers are supported by the design documentation guidelines from the arrowhead framework.

System designs are often created using existing legacy system components to support interoperability of legacy systems.

In the Arrowhead project, technical work packages are designed to provide compliance requirements to validate compliance testing and provide testing capabilities to all partners.

2.1.4. Arrowhead compliance verification

The purpose of the check is to ensure that the relevant system is compatible with the Arrowhead framework. The following are checked during the validation process:

- Can it properly connect and communicate with the core components of the framework?
- Does it conform to the system documentation rules established for Arrowhead compatible systems?
- Does the cloud produce and consume legacy services?

If all the previous questions are correctly checked the cloud is fulfilling all the requirements so it is correct to say that it's an arrowhead compliance cloud.

2.2 Learning the Arrowhead Framework deployment

The second phase of the project will consist of learning how to deploy the framework and analyze the different ways of deploying it. In the first phase we learned how the Arrowhead Framework works in theory, and now we will analyze the deployment and the different ways to deploy it for a better understanding.

All the information about the deployment and configuration of the framework can be found in the (Delsing, 2017) book.

So before starting with the deployment first, we need to know that to create an Arrowhead local cloud we need to deploy the systems manually in the same network and configure correctly so they can work together.

The steps for a correct deployment of the Arrowhead framework are the next ones:

1. Deploy the database and initialize it with the correct schemes.
2. Configure the systems that will be deploy by creating the correct configuration files.
3. Deploy the systems to the network.

For the first step the database needs to be a MySQL database or an SQL database with Hibernate ORM, but which database to use is up to the user. They can decide which database suits better for their use case. For the initialization we can found the SQL scripts inside of the Arrowhead Framework repository.

In the second step the most important part is to correctly configure the addresses of the systems, the database, and the certificates so they can connect and work correctly. There also some other configurations to change how each system works.

Finally, we have the systems deployment, the most complex step. For this step we have 3 different options (one of them only for testing deployment).

The first option and the easiest one is the deployment based on docker containers, but unfortunately this can only be used for testing because the containers are not ready for production deployments. Other inconvenience from these images is that they work on an older version of the Arrowhead Framework. Then we have the option of the Debian installer, in this case the deployment can only be made on Debian based distributions. And the last option is to compile manually the source code and deploy it manually.

2.2.1. Docker based deployment (Only for test)

As mentioned before, this deployment is the easiest and the fastest one. And not to mention all the advantages the Docker technology brings to the deployment (these advantages will be discussed in the next chapter). But it's also different from the other deployments. To make this deployment work we will need to change the first 2 steps just a little.

To start with the docker deployment first we will need to deploy the database inside of the docker network and initialize it. This is very important because if the database is not inside of this network the systems won't be able to connect to it and they won't work.

As we already know docker uses dynamic IPs for his containers so in the configuration files instead of the containers addresses we will need to add the containers name that will be resolve to their corresponding addresses by the docker internal DNS.

Finally, to deploy the systems we will only need to use the corresponding docker image of each service with 2 small modifications. First, we will need to add the corresponding configuration file of the system by adding a volume to the *'/system_name/application.properties'*. The second modification is to open the corresponding port of the system you are deploying, each system will have one by default.

Once everything is done, if everything is correct the systems will start working and create an arrowhead-compliance network.

2.2.2. Debian installer-based deployment

For the Debian installer-based deployment first we will need a Debian based Linux distribution with Java installed. This is not the best option because it limits the deployment to one Linux distribution, but it is a good option to have for those who really use that distribution.

To start with the deployment, we will follow the first two steps without any modification. Then we will download the arrowhead Debian package and add the configuration file to the corresponding folder of the package.

Finally, we can run the installer (.deb file) and the system will start working. Once we run all the core services we will have an arrowhead-compliance network.

This deployment method isn't very complex, but it's harder to configure and it runs into a lot of errors with dependencies version, etc. And these systems don't start on startup so if you want to make them run on startup you will need to add it manually.

2.2.3. Fully manual deployment

For the manual deployment we will need a system with the following requirements: Java (JRE/JDK 11) and Maven (3.5+).

Once we have the system with the correct requirements, we will follow the first 2 steps as in the previous deployment.

Finally, to run the arrowhead systems we will download the arrowhead-core-spring repository. Inside of this repository we will run the command 'mvn install -DskipTests' and this will compile the source code creating the corresponding builds of each system.

Once we have the corresponding builds, we will add the configuration files to their corresponding build folder and finally we will be able to deploy the system by executing the command 'java -jar system_build_file.jar'.

After all this tedious process we will finally have an arrowhead-compliance network running. The problem with this deployment is the same as the previous, but in this case the deployment is more complex, and you could run even into more problems.

2.2.4. Deploying the first network

Now that we know how to deploy these networks, we will make our first deployment to see how they really works and have a better understanding.

For this case we will chose the docker deployment as we are only doing this deployment for testing purposes, and it is the easiest and fastest one. And we will only deploy the three mandatory systems (orchestrator, authorization, and service registry).

To make it even easier this deployment will be created in a single docker-compose file where we will be able to deploy the whole network by just one command.

For the database deployment I decided to use the MariaDB (v10.5) image. For this container I decided to use a volume to make the storage persistent and another one to add the initialization scripts (found on the arrowhead-core-spring repository on GitHub), so the database starts up on the first run. I also opened the corresponding port (3306) so the other systems can connect to the database. The figure 3.1 shows how the docker-compose.

```
services:
  mysql:
    container_name: mysql
    image: mariadb:10.5
    environment:
      - MYSQL_ROOT_PASSWORD=root
    volumes:
      - mysql:/var/lib/mysql
      - ./sql:/docker-entrypoint-initdb.d/
    ports:
      - 3306:3306
```

Figure 3.1: MariaDB docker-compose block

As we can see in the image, I also added the variable of the password, but I left it on root. This is because is just a test network.

Now if we run this container and check the databases, we should see a database named arrowhead with arrowhead tables for each system meaning the database is correctly initialized. We can see this tables in the figure 3.2.

```
Database changed
MariaDB [arrowhead]> show tables;
+-----+
| Tables_in_arrowhead |
+-----+
| authorization_inter_cloud |
| authorization_inter_cloud_interface_connection |
| authorization_intra_cloud |
| authorization_intra_cloud_interface_connection |
| ca_certificate |
| ca_trusted_key |
| choreographer_action |
```

Figure 3.2: Sample of the arrowhead database tables

Now that the database is correctly configure and working, we will define the configuration files (we can find the templates in the repository). For this test network we will only need to modify two fields. The database URL where we will need to add

the container name instead of the actual IP and the service registry address where we will and the container name. In this case the names of the container will be the same name of the systems (serviceregistry, authorization, and orchestrator).

Finally, we will define the container for the core services in the docker-compose file. Deploying the three core systems is very similar so I will only show how to deploy one of them.

For the first container (serviceregistry) we will use the test images created by Svetlint from AITIA that can be found in the Docker Hub repository of Svetlint. Then we will add the configuration file to the `‘/serviceregistry/application.properties’` and open the corresponding port for the system (in this case the port 8443). The figure 3.3 show how the code block looks like.

```
serviceregistry:
  container_name: serviceregistry
  image: svetlint/serviceregistry:4.3.0
  depends_on:
    - mysql
  volumes:
    - ./core_system_config/serviceregistry.properties:/serviceregistry/application.properties
  ports:
    - 8443:8443
```

Figure 3.3: Service registry docker-compose block

Finally, if we do the same for the other 2 services we will have the full arrowhead-compliance network. So, if all the configuration files are correctly configured and we run this network we should see the logs from the figures 3.4, 3.5, and 3.6 meaning the systems has been connected properly and they are working.

```

:: Spring Boot ::      (v2.1.5.RELEASE)

2022-02-18 11:55:56.212 INFO 21e9e0bb0bic [main] e.a.c.s.ServiceRegistryMain : Starting ServiceRegistryMain v4.4.1 on 21e9e0bb0bic with PID 1 (/serviceregistry/arrowhead-serviceregistry.jar started by root in /serviceregistry)
2022-02-18 11:55:56.265 INFO 21e9e0bb0bic [main] e.a.c.s.ServiceRegistryMain : No active profile set, falling back to default profiles: default
2022-02-18 11:56:04.448 INFO 21e9e0bb0bic [main] e.a.c.f.ArrowheadFilter : SRAccessControlFilter is active
2022-02-18 11:56:12.715 INFO 21e9e0bb0bic [main] e.a.c.q.u.UriCrawlerTaskConfig : URI Crawler task scheduled.
2022-02-18 11:56:13.341 WARN 21e9e0bb0bic [main] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-02-18 11:56:15.155 INFO 21e9e0bb0bic [main] .c.s.q.t.ProvidersReachabilityTaskConfig : Providers reachability task is not adjusted
2022-02-18 11:56:15.186 INFO 21e9e0bb0bic [main] .c.s.q.t.ProvidersReachabilityTaskConfig : Services end of validity task is not adjusted
2022-02-18 11:56:15.640 INFO 21e9e0bb0bic [main] e.a.c.f.ArrowheadFilter : PayloadAccessFilter is active
2022-02-18 11:56:18.873 INFO 21e9e0bb0bic [main] e.a.c.ApplicationInitiastener : Core system name: SERVICEREGISTRY
2022-02-18 11:56:18.876 INFO 21e9e0bb0bic [main] e.a.c.ApplicationInitiastener : Server mode: SECURED
2022-02-18 11:56:18.989 INFO 21e9e0bb0bic [main] e.a.c.ApplicationInitiastener : Server CN: serviceregistry.testcloud2.aiitia.arrowhead.eu
2022-02-18 11:56:28.527 INFO 21e9e0bb0bic [main] e.a.c.s.ServiceRegistryMain : Started ServiceRegistryMain in 25.982 seconds (JVM running for 29.826)
```

Figure 3 4: Service registry logs

```

:: Spring Boot :: (v2.1.5.RELEASE)

2022-02-18 11:59:34.379 INFO 02027a34e24c [main] e.a.c.a.AuthorizationMain : Starting AuthorizationMain v4.4.1 on 02027a34e24c with PID 1 (/authorization/arrowhead-authorization.jar started by root in /authorization)
2022-02-18 11:59:34.450 INFO 02027a34e24c [main] e.a.c.a.AuthorizationMain : No active profile set, falling back to default profiles: default
2022-02-18 11:59:52.070 INFO 02027a34e24c [main] e.a.c.f.ArrowheadFilter : AuthAccessControlFilter is active
2022-02-18 12:00:11.729 INFO 02027a34e24c [main] e.a.c.q.u.UriCrawlerTaskConfig : URI Crawler task scheduled.
2022-02-18 12:00:13.437 WARN 02027a34e24c [main] aWebConfigurations$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-02-18 12:00:19.165 INFO 02027a34e24c [main] e.a.c.f.ArrowheadFilter : PayloadSizeFilter is active
2022-02-18 12:00:25.513 INFO 02027a34e24c [main] e.a.c.ApplicationInitListener : Core system names AUTHORIZATION
2022-02-18 12:00:25.517 INFO 02027a34e24c [main] e.a.c.ApplicationInitListener : Server mode: SECURED
2022-02-18 12:00:25.751 INFO 02027a34e24c [main] e.a.c.ApplicationInitListener : Server CN: authorization.testcloud2.aitia.arrowhead.eu
2022-02-18 12:00:28.367 INFO 02027a34e24c [main] e.a.c.ApplicationInitListener : Service Registry is accessible...
2022-02-18 12:00:31.349 INFO 02027a34e24c [main] e.a.c.ApplicationInitListener : Core system AUTHORIZATION published 5 service(s).
2022-02-18 12:00:32.975 INFO 02027a34e24c [main] e.a.c.a.AuthorizationMain : Started AuthorizationMain in 61.084 seconds (JVM running for 65.758)

```

Figure 3.5: Authorization logs

```

:: Spring Boot :: (v2.1.5.RELEASE)

2022-02-18 11:59:34.509 INFO 8912736941f4 [main] e.a.c.o.OrchestratorMain : Starting OrchestratorMain v4.4.1 on 8912736941f4 with PID 1 (/orchestrator/arrowhead-orchestrator.jar started by root in /orchestrator)
2022-02-18 11:59:34.548 INFO 8912736941f4 [main] e.a.c.o.OrchestratorMain : No active profile set, falling back to default profiles: default
2022-02-18 11:59:52.329 INFO 8912736941f4 [main] e.a.c.f.ArrowheadFilter : OrchestratorAccessControlFilter is active
2022-02-18 12:00:12.536 INFO 8912736941f4 [main] e.a.c.q.u.UriCrawlerTaskConfig : URI Crawler task scheduled.
2022-02-18 12:00:13.869 WARN 8912736941f4 [main] aWebConfigurations$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-02-18 12:00:19.126 INFO 8912736941f4 [main] e.a.c.f.ArrowheadFilter : PayloadSizeFilter is active
2022-02-18 12:00:19.538 INFO 8912736941f4 [main] e.a.c.q.q.t.ReservationCheckTaskConfig : Services reservation task adjusted with interval: 120 seconds
2022-02-18 12:00:25.890 INFO 8912736941f4 [main] e.a.c.ApplicationInitListener : Core system name: ORCHESTRATOR
2022-02-18 12:00:25.896 INFO 8912736941f4 [main] e.a.c.ApplicationInitListener : Server mode: SECURED
2022-02-18 12:00:26.114 INFO 8912736941f4 [main] e.a.c.ApplicationInitListener : Server CN: orchestrator.testcloud2.aitia.arrowhead.eu
2022-02-18 12:00:28.402 INFO 8912736941f4 [main] e.a.c.ApplicationInitListener : Service Registry is accessible...
2022-02-18 12:00:32.349 INFO 8912736941f4 [main] e.a.c.ApplicationInitListener : Core system ORCHESTRATOR published 8 service(s).
2022-02-18 12:00:33.335 INFO 8912736941f4 [main] e.a.c.o.OrchestratorMain : Started OrchestratorMain in 61.368 seconds (JVM running for 66.116)

```

Figure 3.6: Orchestrator logs

Watching the logs from the figures 3.4, 3.5, and 3.6 we can see that the core systems has initialized correctly, and they are connected correctly so the arrowhead network is ready to start working.

2.2.5. Testing the first network

Now that we already deployed our first network, we will test it with a consumer/provider test to prove the network works correctly.

For this test the arrowhead repository already has some scripts to run this test. These scripts first run ping commands to check the availability and then they create a provider and a consumer. The provider just sends a 'hello world' message every time the consumer sends a request. So, at the end of the test, we should see the 'hello world' response.

As these tests are very practical to check if the network works properly, I decided to automate them by creating a container that runs this test once every time you execute the container, so you don't need to worry about how to execute each script. The image for these tests can be found on my Docker Hub repository 'alexmedela/test_image'.

Now we will run these tests to our previously created network to see if it works properly.

```
> GET /helloworld HTTP/1.1
> Host: 127.0.0.1:9981
> User-Agent: curl/7.79.1
> Accept: */*
>
{ [5 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [1081 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [1081 bytes data]
* old SSL session ID is stale, removing
{ [5 bytes data]
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 ok
< Content-type: text/plain
<
{ [23 bytes data]
* Closing connection 0
} [5 bytes data]
* TLSv1.3 (OUT), TLS alert, close notify (256):
} [2 bytes data]
{
  "hello": "world"
}
```

Figure 3. 7: Tests final response

In the figure 3.7 we can see the final response of the test with a 'hello world' meaning the arrowhead network has passed the test and is working properly. The logs from this test are very detailed during the process so every time there is a problem it's easy to identify and correct it.

2.3 Arrowhead Framework automation

On the previous chapter we analyzed the deployment of an arrowhead compliance local cloud. Now we will analyze how to automate this deployment with the objective of making it as easy and fast as possible to deploy.

The deployment of the framework can be separated into 2 phases for the automation. The deployment of each system individually and the deployment or creation of the local clouds.

This chapter will compare and analyze different technologies for the automation of each phase and implement the best option.

2.3.1. Automating the systems deployment

As we saw in the previous chapter there are three different ways for deploying the systems. Between these 3 options we already saw that the best option is to use containers, but there are more options that could be applied to this deployment.

Another problem with the containers option is that the already existing images are only for testing purpose.

The container engine

The best option to start automating the framework is to containerize the systems for the next reasons:

- Ability to run almost everywhere
- Easy to deploy
- Easy to maintain and manage
- Easy to scale
- Compatibility with new technologies and cloud services (Kubernetes, Heroku, some AWS services, etc.)
- Easy to implement on pipelines and IaC (Infrastructure as Code)

Taking the previous points in mind there no doubt that the best option is to containerize the system individually to start with the automation. But which container engine should we use?

For this case the best option and the most popular is Docker. Some of these reasons are the next ones:

- I already mastered the Docker technology
- Huge community support
- Provides tools like docker compose, swarm, and registry
- Easy to use
- Open source with a very active development team

Now that we know that the best option is to use Docker, but the images already created are only for testing purpose we will need to create our own images for production deployments.

Docker implementation

For the development and future updates of the docker images, containers and arrowhead compliant network based on Docker, the GitHub repository has a specific folder. The figure 4.1 shows this folder and the structure of it.

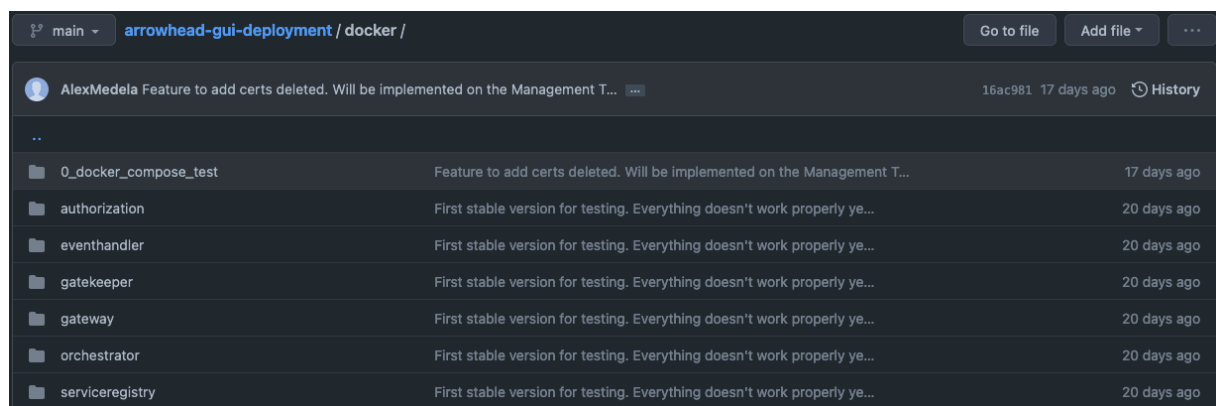


Figure 4.1: GitHub repo Docker implementation

On this folder (figure 4.1) we will first find the '0_docker_compose_test' folder. This folder contains the docker compose and basic configuration for the arrowhead compliant network. And it is used to test if the docker images work correctly.

Then we have the folders named by the systems name. Each of these folders contains a *Dockerfile* and the folder with the corresponding system build (created by following the manual deployment steps).

Using this structure each *Dockerfile* will create the image of the corresponding system. In the Figure 4.2 we can see the *Dockerfile* structure.

```
FROM alpine:3.15
RUN apk update && apk add openjdk11-jre
COPY app /authorization
CMD ["bin/sh", "-c", "cd /authorization && java -Xms64M -Xmx128M -Dlog4j.configurationFile=log4j2.xml -jar arrowhead-authorization.jar"]
```

Figure 4.2: Dockerfile structure

As we can see in the *Dockerfile* structure (Figure 4.2) is really simple. First, as base image we use the alpine image with the last version, but we don't set the latest flag because we don't want this base image to be update automatically when the newest version is released. The is reason of using this image is because we need to build the smallest image to make the image as efficient as possible.

Then we use the 'RUN' command to install the dependencies that in case it's only the JRE 11. After we install the dependencies, we need to add the already build jar file that can be found in the folder next to the *Dockerfile* (with the same system name) and we do that using the 'COPY' command.

Finally, we set the command that will run on startup to initialize the system. In this case, we used some flags to limit the memory and CPU usage of the system with the intention to make this image more efficient.

The docker images created by these *Dockerfiles* can be found in my Docker Hub repository (alexmedela/). And they are built with the version 4.4.1 of the Arrowhead Framework.

Docker images usage

The already created docker images are ready for a production deployment. But to make this production deployment, the correct certificate must be added by using a volume to the '/system_name' folder. If not, the system will use the test certificates and won't be ready for the production deployment.

2.3.2. Automating the cloud deployment

Automating the cloud deployment means the automation creating the arrowhead compliance cloud using the systems. So, as we are using containers for the system deployment this opened the option to use container orchestrators.

Container orchestrators

The container orchestrators are tools to automate the deployment, management, scaling, and networking of containers. And these are the perfect tools to continue with the automation of the Arrowhead Framework.

These tools offer the advantages of:

- Scaling your applications and infrastructure easily
- Service discovery and container networking
- Improved governance and security controls
- Container health monitoring
- Load balancing of containers evenly among hosts
- Optimal resource allocation
- Container lifecycle management

As we can see the container orchestrator is the best option to automate the deployment of the cloud, but between the container orchestrators tools we can find a lot of them to compare.

For our use case, the most interesting ones are Docker Swarm, Kubernetes, and HashiCorp Nomad.

In this case, as we are using Docker and it's a technology, I know very well the Docker swarm is also an interesting choice. On the other hand, Kubernetes is the most popular choice and the most used one because of all the options and advantages it offers. The last one is Nomad, a new container orchestrator with some advantages over Kubernetes and compatibility with Terraform.

Docker swarm

Docker swarm is a container orchestrator offered by Docker. And this tool offers the next features:

- Manage clusters using the Docker engine
- Use declarative service models
- Scaling
- Desired state reconciliation
- Multi host networking

- Service discovery
- Load balancing
- Rolling updates

This is what we need to automate our deployment, but it also has some disadvantages when talking about the implementation that makes it a bad choice.

- Limited functionality compared to other orchestrators
- Limited fault tolerance
- Small community support
- Clusters need to be created and configured manually

Talking about how it works we can find the structure of a docker swarm cluster in the next image (Figure 4.3).

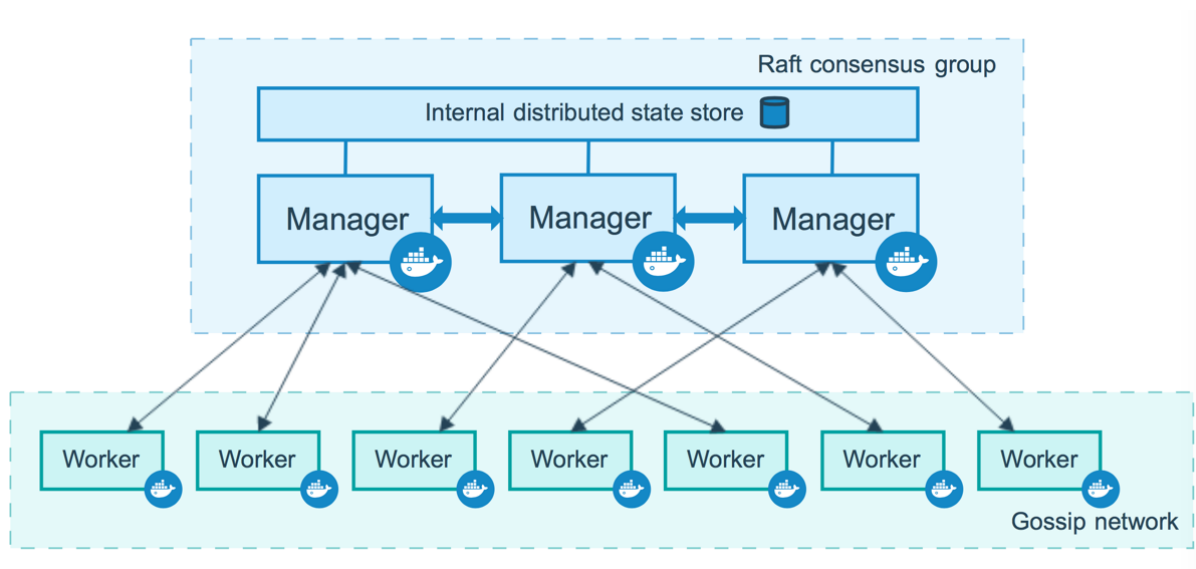


Figure 4.3: Docker swarm cluster

This Docker swarm cluster is formed by the servers (managers) and the workers all connected between them.

The servers are the ones that manages the all the workers. They distribute the task between all the workers in the most efficient way and are connected between to providing fault tolerance and manage more workers more efficiently. They also use an internal distributed state storage, so all the servers have the state of the cluster up to date and provide integrity to the system.

The workers are just the ones that will take the tasks from the servers and run them. These is where the containers and all the process will be.

Kubernetes

Kubernetes is by default one of the most popular choices for container orchestrators. It is very used for deployment, management, and scaling automation because of all the features it offers:

- Scaling and auto-scaling
- Monitoring
- High availability
- Load balancing
- Desired state reconciliation
- Rolling updates and rollbacks
- Health checks
- Storages
- Service Discovery
- Huge community support
- Easy cluster deployments

For the features it offers this is a great tool that covers all the needs to automate the deployment, but let's see some of the disadvantages of this project:

- Only works for containers
- It's very complex to learn
- Consumes a lot of resources and sometimes it can be an overkill

To see how these cluster works let's take a look to the figure 4.4, where we can see the general structure of the clusters.

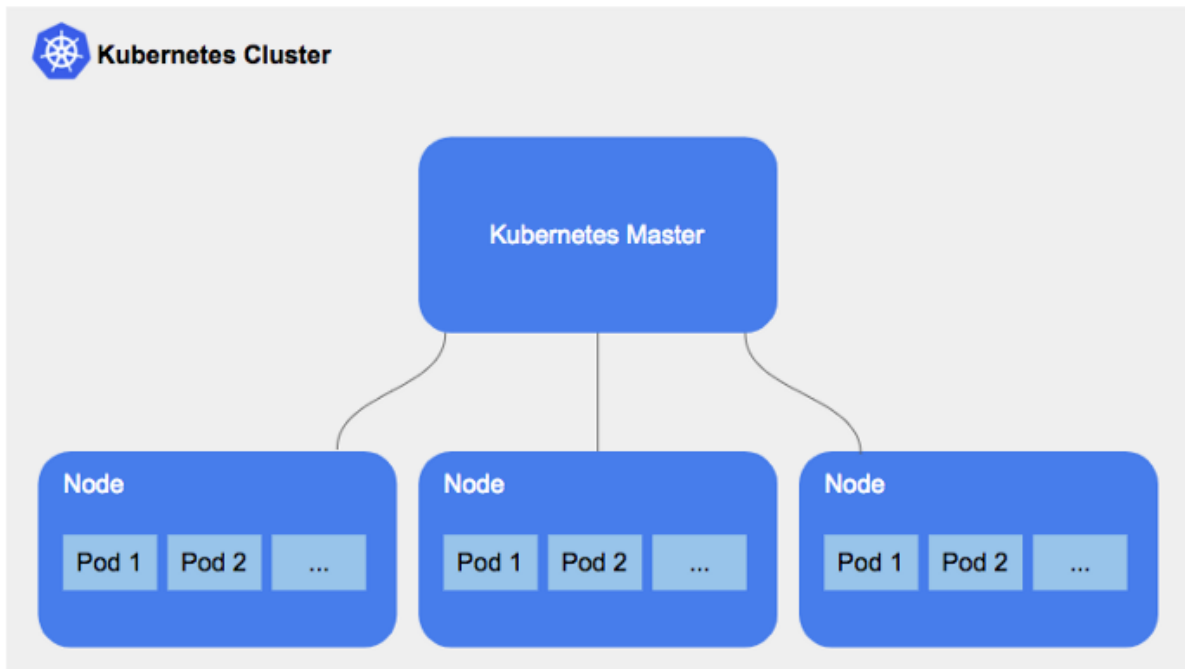


Figure 4. 4: Kubernetes cluster

In this case we have one master managing the worker nodes.

This master node will manage all the workers the same way the Docker swarm server do. It will distribute all the task between the workers and keep a state up to date to maintain this state between the workers and protect from unexpected failures.

The nodes in this case will take the task and run them, but in this case the containers won't be directly deploy in the worker. Kubernetes uses a different solution where the containers are deployed inside of the pods. These pods are just isolated networks where the containers can be deployed.

Nomad

Nomad is the container orchestrator developed by HashiCorp. This is becoming another popular choice as Kubernetes and offers interesting new features apart from the same as the ones from Kubernetes:

- Easy to use and learn
- Legacy application deployment
- Native integration with Terraform, Consul, and Vault

Apart from these new features it also has some disadvantages:

- Small community support

- Complex cluster deployment

Taking the new features in mind, the possibility to automate the deployment, management, and scaling of application without containerizing them it a good option and could be good for the Arrowhead Framework deployment automation.

About how it works, it almost the same as Kubernetes. But in this case the cluster is differently formed and instead of pods the term group is used (can be saw in the figure 4.6). In the figure 4.5 we can see the cluster structure.

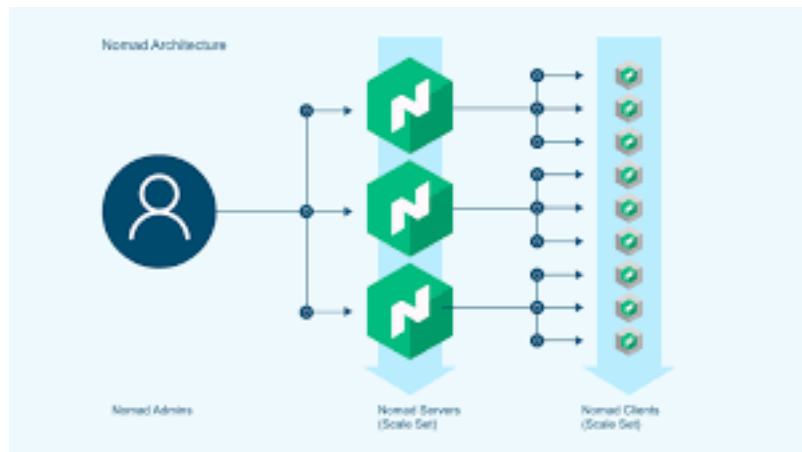


Figure 4.5: Nomad cluster

The structure in this case is like the Docker swarm structure and they work in the same way.

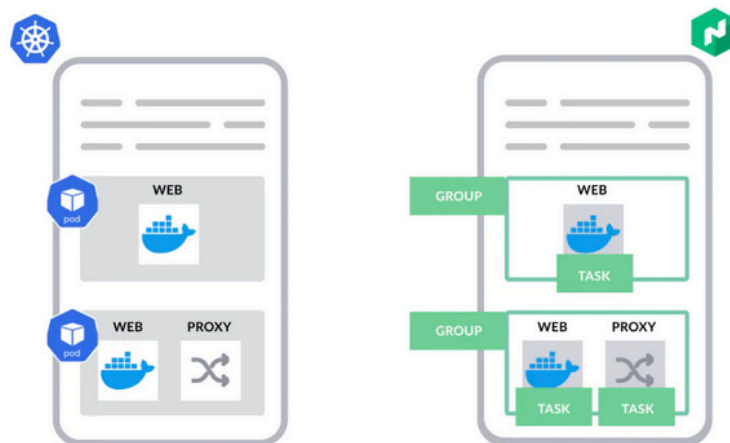


Figure 4.6: Kubernetes and Nomad similarity

Container orchestrator comparison

So far, we have seen 3 different container orchestrators that fits for the automation of the Arrowhead Framework, but now we will see which one is better from these 3 options.

The objective is to make the deployment as easy as possible for the users so after seeing each of the container orchestrator the best option is Kubernetes for the following reasons:

- It's the orchestrator with most functionalities
- It's already highly adopted by the users
- Kubernetes clusters are easy to deploy
- Even if it's complex to learn I already have basic-mid knowledge about it

The key reason to use Kubernetes from the previous list is that clusters are easy to deploy. These clusters are not easy to deploy by itself, but as Kubernetes is a very developed tool with a huge community support there are a lot of tools to automate the deployment of these clusters.

On the other hand, as it is a very popular tool there are a lot of cloud services that offers Kubernetes as a service allowing to easily deploy and manage a Kubernetes cluster.

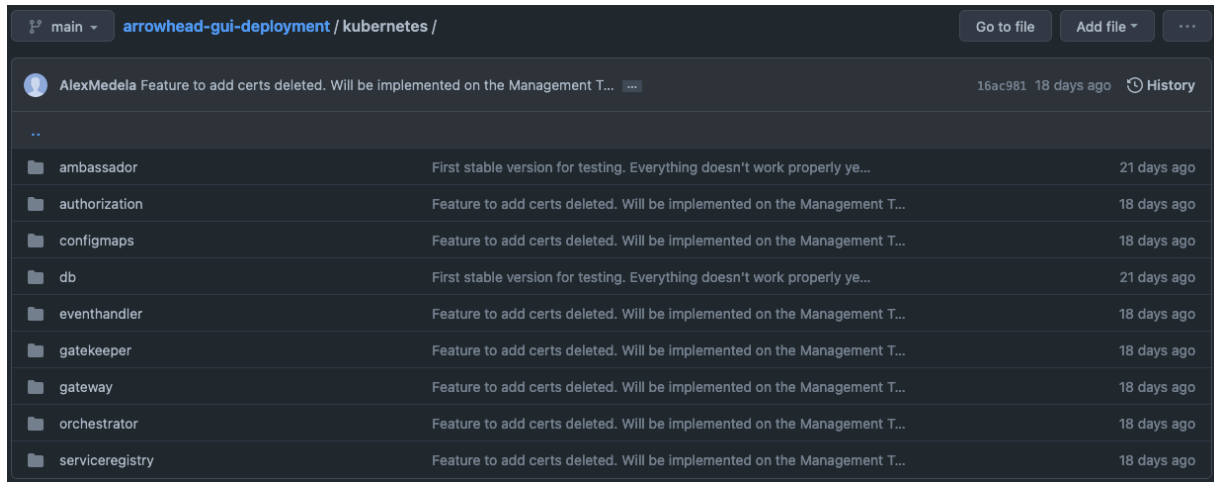
So, the only disadvantage we are taking from this option is that it will consume a more resources than the rest of the options

Implementation

The Arrowhead Framework can be deployed with very different structures. Depending on the use case of the user, he will want to deploy some services or others and between these services some of them will be more important than others.

So, when implementing the Kubernetes deployment, I just defined all the resources independently so the user can choose how to deploy them and how many replicas of them to deploy.

In the GitHub repository we can find the Kubernetes folder where all resources are defined.



| Folder | Description | Timestamp |
|-----------------|--|-------------|
| .. | | |
| ambassador | First stable version for testing. Everything doesn't work properly ye... | 21 days ago |
| authorization | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| configmaps | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| db | First stable version for testing. Everything doesn't work properly ye... | 21 days ago |
| eventhandler | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| gatekeeper | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| gateway | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| orchestrator | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |
| serviceregistry | Feature to add certs deleted. Will be implemented on the Management T... | 18 days ago |

Figure 4. 7: Kubernetes resources

The figure 4.7 shows the folder with all the resources. Here we will find all the services, the simple database, the configmaps and the ingress controller (ambassador).

First, the systems are defined by one deployment and a service to expose this system. The deployment is just one pod containing the systems container with all the corresponding configurations. All systems follow the same deployment and service structure.

Then, we have the configmaps. These stores the configuration of the systems, and the database initialization scripts.

We also have the simple database that will be needed for the systems data storage. This database uses a deployment, service, and a persistent storage so the data won't be lost in case the database fails.

Finally, we have the ambassador ingress controller. This is used to expose the services outside of the cluster using URLs. The ambassador controller allows to expose the services using the next format: cluster_ambassador_ip/system_name. So, this makes it easy to find the services.

All the code from these resources can be found in the [GitHub repo inside of the folder Kubernetes](#).

2.3.3. Graphical user interface

Now we already have automated the deployment using Kubernetes, but this tool is complex to learn, and we still need to use it manually if we want to make a deployment. So, the deployment is still being complex for users without knowledge about Kubernetes.

To fix this problem, a graphical user interface will be used where the users can choose which services to deploy and how many of them to deploy. Automating the use of all the Kubernetes resources defined previously.

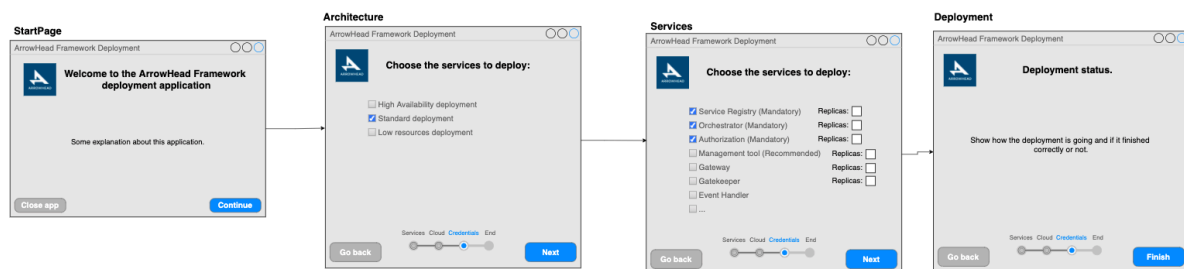


Figure 4. 8: First GUI mock-up

In the figure 4.8 we can see the mock-ups of the graphical user interface that will be built to automate the deployment.

First, we have a welcome window with some info about the app. Then the option to check different availabilities will show, this window would set up the replicas count of the services and deploy a cluster database instead of a simple one. After that, the GUI will let the user check which services to deploy and how many replicas of them. Finally, the deployment would start, and the last window would show the progress of the deployment showing all the errors and logs that happened during the process.

2.4 Implementing cloud providers

During the previous chapter we talked about the most important advantage of Kubernetes, and this was that there are Kubernetes clusters as service. This allows the users to easily deploy Kubernetes cluster, but they still have to do it manually and we that would add more complexity to the users.

So, to take those complexities away from the user, cloud providers will be added to implementation. This cloud providers will be used to create Kubernetes clusters using their 'Kubernetes as a service' service.

To do so, the best tools are the IaC (Infrastructure as code) tools that would allow us to define the infrastructure of the Kubernetes cluster as code and automate the deployment.

Another key part is the cloud providers to add. For this project we want to make it available for most of the users so the idea would be to use the big 3 cloud providers: AWS (Amazon Web Services), GCP (Google Cloud Platform) and Microsoft Azure.

2.4.1. Infrastructure as code tools

Between the different infrastructure as code tools there isn't a lot of options to decide. The most popular one by difference and the best one is Terraform which is the one that will be used. Then we can find specific IaC tools for AWS, Azure, etc., but Terraform is the one that allows to use infrastructure as code for almost all providers with a clear language (HashiCorp language).

The only tool that could compare to terraform would be Ansible, but they don't have the same utility. Ansible is more used for configuration automation while terraform is used for infrastructure automation.

2.4.2. Cloud providers

For the implementation the best idea would be to use the big 3 cloud providers, but AWS doesn't offer a free trial for the Kubernetes service so we will only add the Google

cloud and Microsoft azure to this implementation. AWS will be left for future implementation when it is viable to pay for those services to test the implementations.

2.4.3. Implementation

For the implementation of the cloud providers a Terraform folders has been created inside the repository. The figure 5.1 shows this structure.

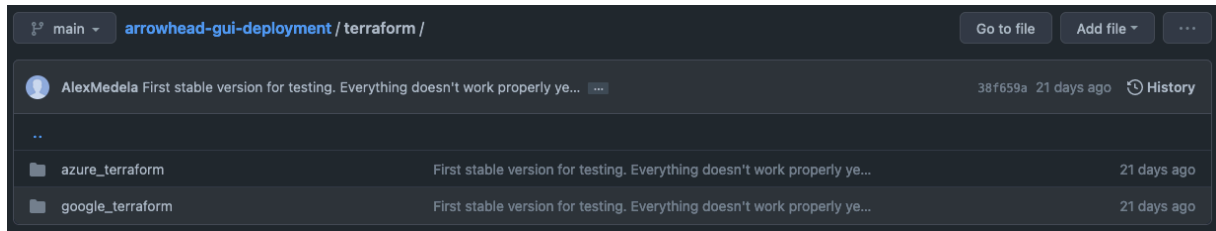


Figure 5. 1: Terraform structure

Inside of the folder we will find each of the cloud provider that has been implemented. Each one has different structures, but both creates a default Kubernetes cluster with enough resources for the arrowhead compliance cloud.

When running any of these, they will ask for credentials to connect to the cloud provider, but these credentials won't be stored.

2.4.4. Graphical user interface

The same problem that happened with the Kubernetes implementations happens with the terraform implementation, so we will need to add this part also to the GUI already design. The figure 5.2 shows the mock-up of how we would add the terraform part.

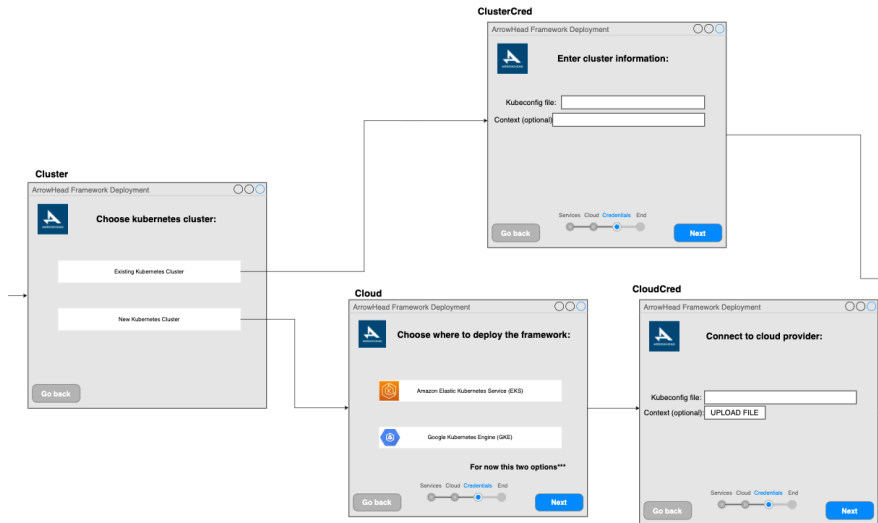


Figure 5. 2: Terraform part mock-up

The part shown in the figure 5.2 would be added after choosing the services and before the deployment.

And as we can see there is also an option not mentioned before that allows the user to deploy the cluster on an already existing Kubernetes cluster. This is a great option for those who already works with Kubernetes, and they already have their Kubernetes clusters deployed.

2.5 Pipeline workflow

This project works with a framework that is being constantly develop by an active team and releasing updates. So, the systems containers also need to be updated. This would be a tedious process where new containers need to be built every time there are new changes to the framework.

So, to make this fast and easy the idea of a pipeline has been developed. The figure 6.1 shows the workflow of this pipeline.



Figure 6.1: Pipeline workflow definition

As we can see in the figure 6.1 the pipeline triggers when the users commit a change and pushes it to the repo. After pushing the commit, the pipeline would create the new containers with the new version. Then these containers would be tested to check if they work correctly. Finally, if everything works correctly the new version would be uploaded to the Docker Hub repo updating the latest flag and making the project work with the newest version without updating the containers manually.

If something goes wrong during the process or they don't pass the test, this version would be uploaded meaning the project would maintain the previous working version.

2.5.1. Implementation

The implementation was planned to be using GitHub actions but taking in mind the rest of the project tasks there wasn't enough time to implement the pipeline, so it has been added as a future improvement.

3

Project results

During this chapter the final product will be shown in detail. We will see the changes from the first ideas shown in the previous chapters and we will also test the product to verify it's working correctly.

3.1 Changes during the process

This project has been in development for over 5 months and in collaboration with more people helping with the development. Due to these reasons, there has been many changes during the process.

Most of these changes were minor changes (improvement suggestions) and new features suggestions. But there are some other big changes with specific reasons that changed the first ideas of the project.

These big changes are the next ones:

- The pipeline wasn't implemented: This change was due to time problems. The project was an ambitious idea so at the end there wasn't enough time to implement the pipeline.
- The second window of the GUI was supposed to ask for High availability, Standard deployment, and low resources deployment, but this wasn't very clear for the users as it doesn't explain any differences in the deployment. To make this clearer it was changed by a window where it only asked for the type of database the user wants to deploy: simple instance or a cluster database.
- The 3 cloud services. At the very first beginning the idea was to allow the cluster creation on AWS, GCP and Azure. But Amazon didn't have the option to use this service for free (not even with free trials) so, the project doesn't include the AWS cluster creation option.
- The management tool and certificates implementation. During the process the management tool container was developed, but after an update of the framework the container wasn't updated because another big update for this tool was already in development. This update was a new feature for the tool to add and manage the system certificates. And I was implementing a new feature to add these new certificates using the GUI, but it wasn't a practical solution, so we decided to only implement it on the management tool. Finally, the implementation of the certificates and management tool system container is on wait to finish this new update. For this reason, there is no option to add certificates via the GUI and there is no option to deploy the management tool yet.

3.2 Product

Finally, we will see how the product looks and how to use it. Then in the next section we will test the product to verify it works correctly if you follow the instructions.

First to start the application we will need to install the dependencies. We can find all the instructions on the GitHub repository, in the README that has been written following the good practices of how to write correct repository documentations.

The dependencies are the next ones:

- The operative system must be MacOS or Linux
- Python 3.8 or higher with pillow package installed (recommended to use a virtual environment)
- Kubectl 1.21 or higher
- Terraform 1.1 or higher (only if you need to create a new cluster on the cloud)
- Helm 3.7 or higher (only if you want to deploy the db cluster)
- The Kubernetes cluster must be 1.20 or lower

Once we fulfill all the dependencies, we can run the application by running the main file *application.py* using the next command:

- `python3 application.py`

After running the command, we will start with the GUI as we can see in the figure 7.1.

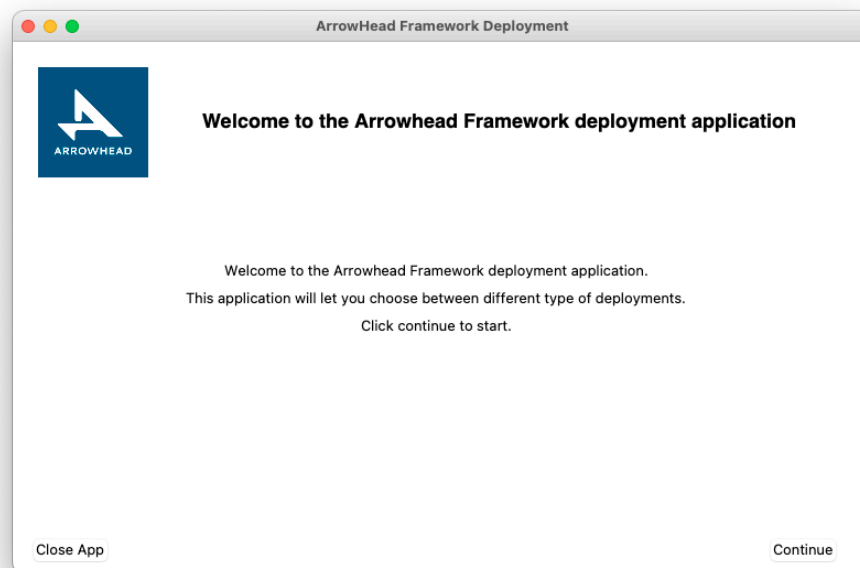


Figure 7.1: GUI welcome page

As we can see this first window is just an introduction page with a small welcome message. In this window we can just close the application or go to the next page (figure 7.2).

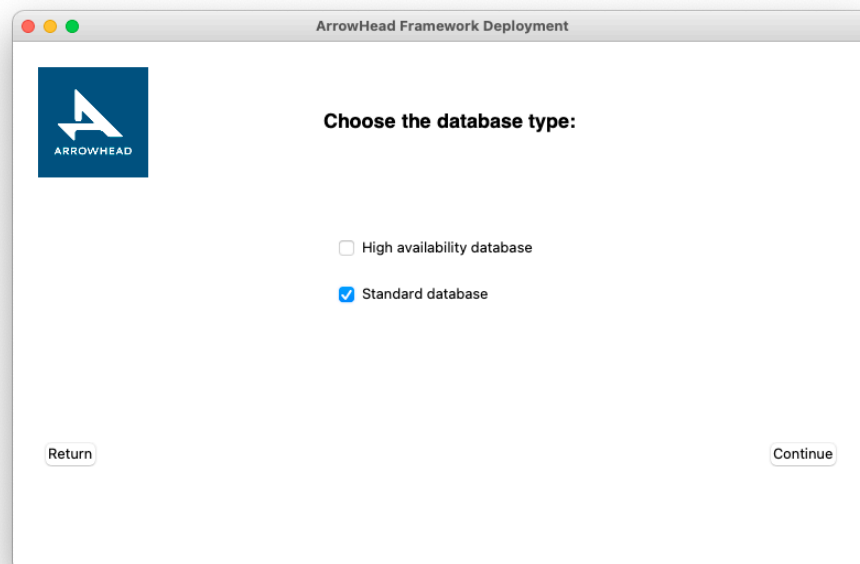


Figure 7.2: Database selection

In this second window (Figure 7.2), we are allowed to choose between the high availability database (a cluster database) and the standard database (a simple

instance of the database). These checkboxes won't allow the user to check both of them at the same time.

Once we select the database, we will click on continue and go to the services selection window (Figure 7.3).

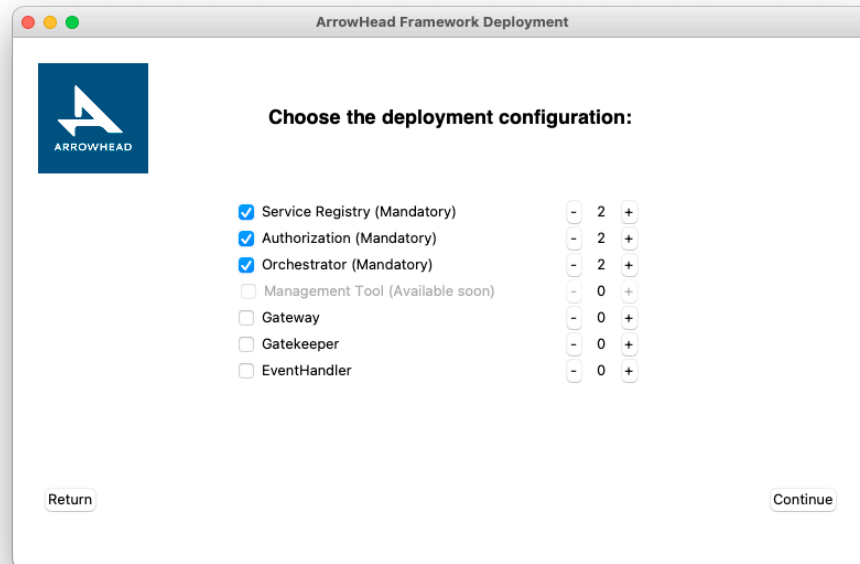


Figure 7.3: Services selection

The services selection window (Figure 7.3) allows to select which systems to deploy and how many replicas of them. As we mention in the project changes the management tool is not ready yet for deployment.

In this case, the core systems checkboxes can't be unselected, and the counters can't be less than one when the checkbox is selected, and they can be any other value than 0 when they are unchecked. This is programmed so it automatically adjusts when changing the values.

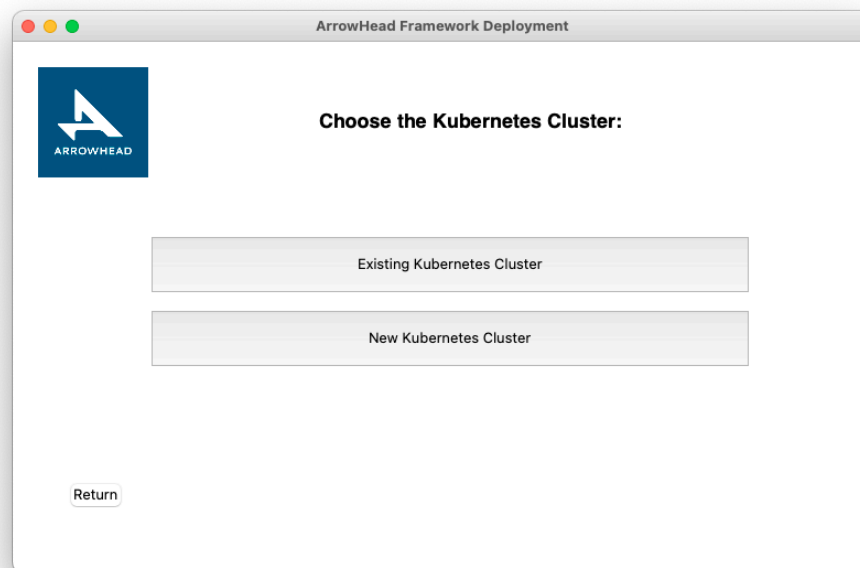


Figure 7.4: Cluster selection

After selecting the services, the deployment phase will be finished, and we will be asked if we want to use our own Kubernetes cluster or create a new one. The creation of the clusters will be in the cloud services.

Here we will find 2 different ways, but very similar between them.

3.2.1. Using an existing cluster

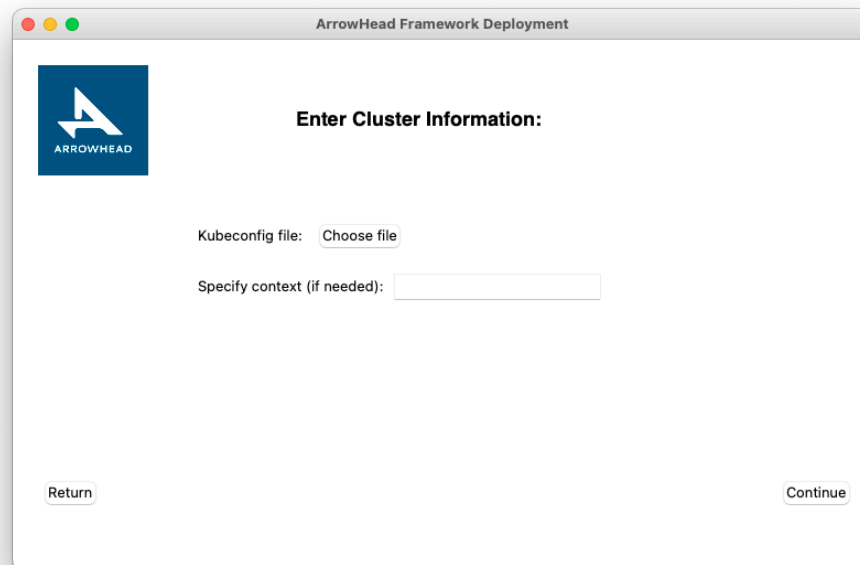


Figure 7.5: Credentials for existing cluster

When using an existing cluster, we will be asked for the Kubeconfig file and the context if it is needed (Figure 7.5). The Kubeconfig file is the file used to connect to the cluster via the Kubectl command, this file contains the information of the cluster and the secrets (none of them will be stored). Then the context is only needed when the Kubeconfig file contains connections for more than one cluster and the user wants to specify which cluster of those to use.

This window (Figure 7.3) can't be skipped if the user doesn't input any Kubeconfig file.

3.2.2. Creating a new cluster

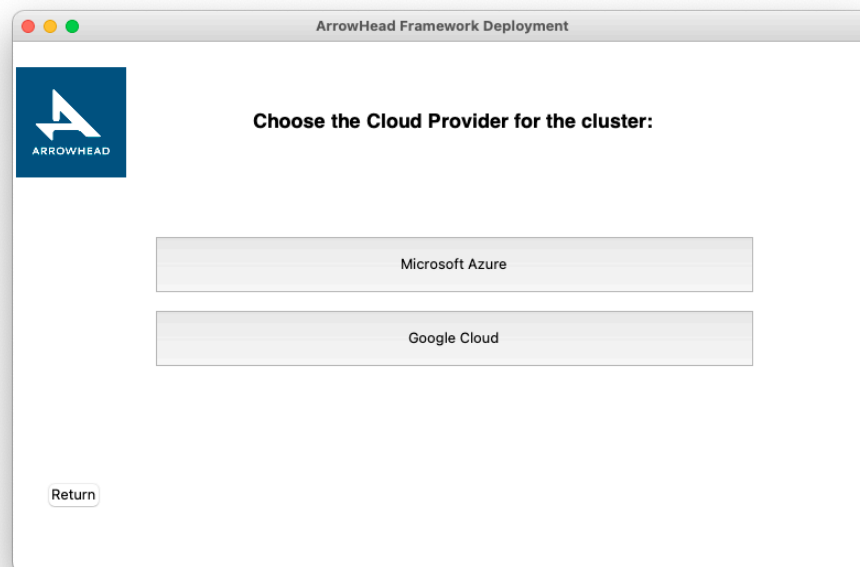
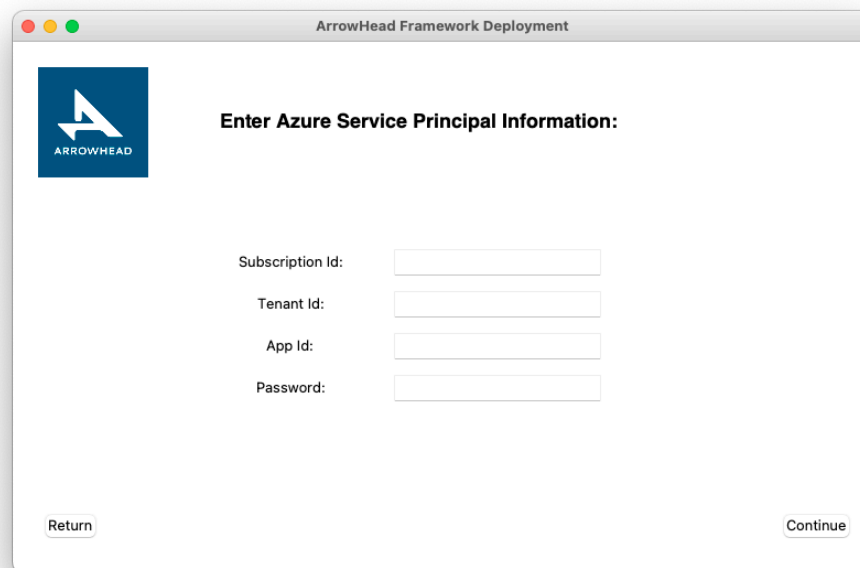


Figure 7.6: Choosing cloud provider

When creating a new cluster, we will first be asked for the cloud provider we want to use. In the figure 7.6 we can see the 2 available options: Microsoft Azure and Google Cloud. After this we will be asked for the corresponding credentials of each one.

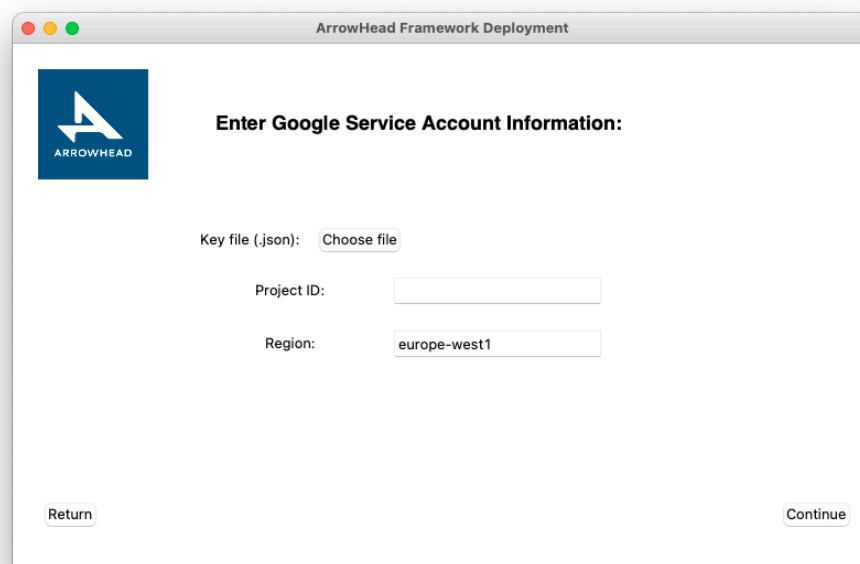
Each cloud service provides different ways to connect to the service so for each option different credentials will be needed.



The screenshot shows a window titled "ArrowHead Framework Deployment". On the left is the ArrowHead logo. The main heading is "Enter Azure Service Principal Information:". Below this are four input fields: "Subscription Id:", "Tenant Id:", "App Id:", and "Password:". At the bottom left is a "Return" button and at the bottom right is a "Continue" button.

Figure 7.7: Microsoft Azure credentials

The figure 7.7 shows the needed credentials for the Azure cluster creations. In this case, we are using the Service Principal credentials and we will need the Subscription id, tenant id, app id and password. None of this data will be stored.



The screenshot shows a window titled "ArrowHead Framework Deployment". On the left is the ArrowHead logo. The main heading is "Enter Google Service Account Information:". Below this are three input fields: "Key file (.json):" with a "Choose file" button, "Project ID:", and "Region:" with the value "europe-west1" entered. At the bottom left is a "Return" button and at the bottom right is a "Continue" button.

Figure 7.8: Google Cloud credentials

For the Google Cloud case, we are using the JSON key file. And we will only need the JSON key file, the project id, and the region.

3.2.3. The GUI deployment

Finally, both options will merge into the deployment window (Figure 7.9). This window will create the cluster (if the user chose to create it) and deploy the framework to the cluster. The window will show the logs of how the deployment is going. But it will not show the errors because this feature is not implemented yet. The errors log can be shown in the console where the app has been executed.

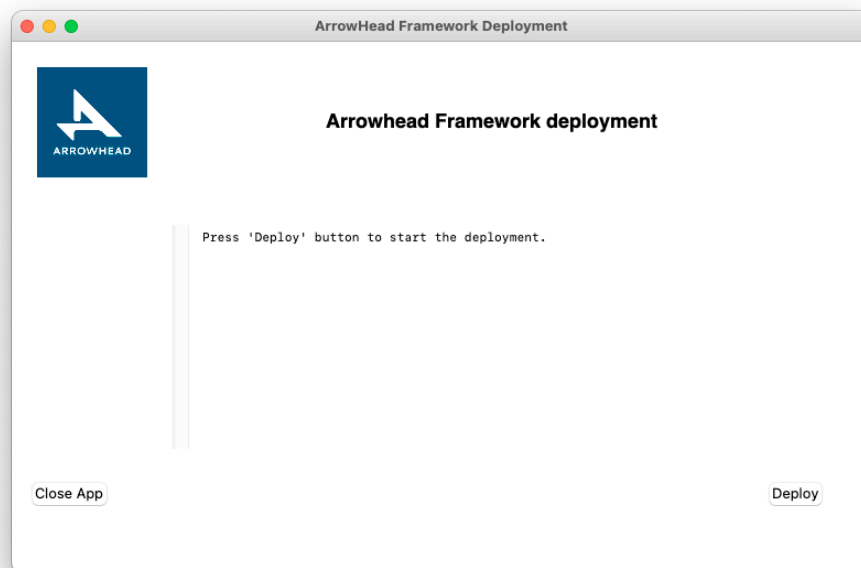


Figure 7.9: Deployment window

After the deployment window is finished, we will already have the cluster deployed on the cloud ready to use. The GUI doesn't show yet the IP which we need to use to connect to the systems, so until this feature is implemented, we will need to use the command:

- `kubectl get services --namespace=ambassador`

Here we will take the only external IP that will be shown, and we can use the format `https://external_ip/system_name` to connect to the systems.

3.3 Product test

Now that we have seen the GUI, we will see the result we can obtain using it. We will test it using it to deploy a simple arrowhead compliance cloud. For this test we will use the next configurations and tools:

- The cloud will be formed by one replica of each core system
- We will use the simple database
- We will use minikube as the k8s cluster

Before we begin with the test, we need to deploy the minikube cluster using the version 1.20.0 of Kubernetes (this is one of the prerequisites of the application) and enable the addon metallb (this is to enable load balancing in the cluster).

So, we will start running the next commands to start the minikube test cluster:

- `minikube start --kubernetes-version=v1.20.0`
- `minikube addons enable metallb`
- `minikube addons configure metallb` (add the IP range you wish for the app)

After executing the commands, we will already have our test cluster deployed and configured on our localhost. Also, the `kubectl` command will be automatically configured by minikube to connect to the cluster so we don't need to do anything.

3.3.1. Validation

Once we have our cluster ready to use, we will run the application and mark the next options to deploy the simplest arrowhead compliance cloud.

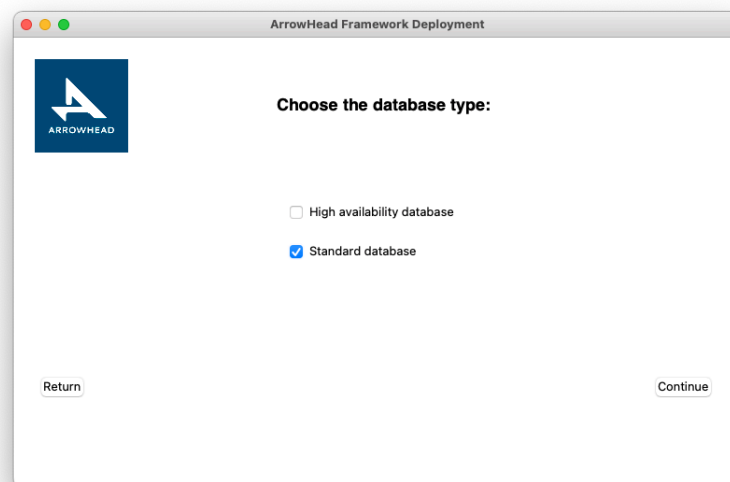


Figure 7. 10: Database type

For the database type we will check the standard database, as shown in the figure 7.10, that will deploy and simple database with only one instance. If we check the High availability database, a cluster of databases will be deployed.

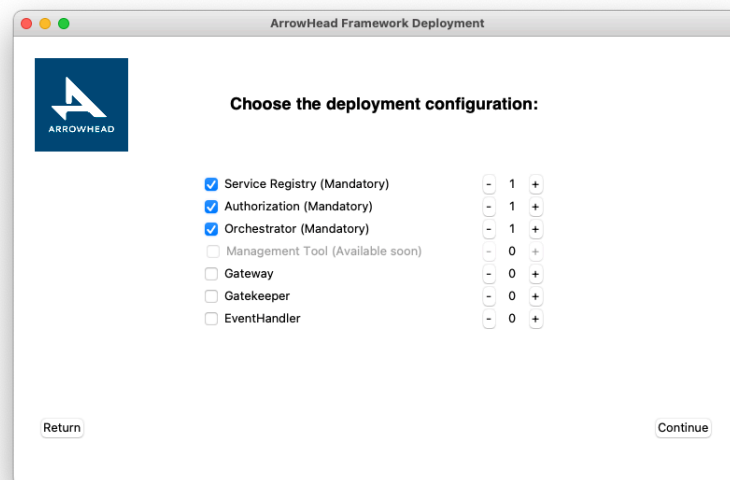


Figure 7. 11: Choosing services

Next, we will only mark the core services and set the replica count to 1. This will make the cloud lighter so it can be run on low resources computers without any problems.

After the deployment configuration we will select the option to deploy it on an existing cluster. The option to create the clouds will be tested later.

To connect to the cluster, we will need to specify the Kubeconfig file, which is the file used by the kubectl command to connect to the cluster. Depending on the cluster we created the file can be obtained in different ways. In our case, the file is automatically created by the minikube cluster and configured for the kubectl command, so it can be found in the '~/.kube' directory with the name 'config'.

Sometimes the kubectl command is configured to connect to different cluster so when this happens, we need to specify the context that will define the cluster to which we will connect. In this case it won't be needed.

We can see the previously explained fields in the figure 7.12.

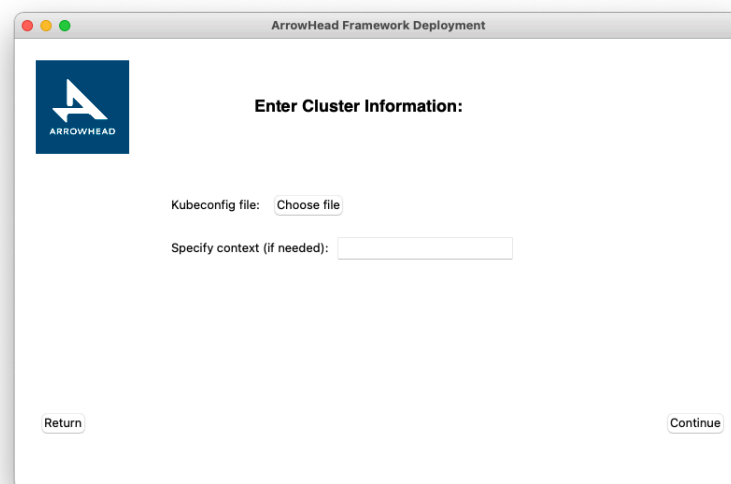


Figure 7. 12: Connecting to the cluster

Finally, we will get to the deployment window where we will select the deploy button and the deployment will start. It can take up to 10 minutes and when it finishes if everything goes correctly, we will see the same logs as the figure 7.13.

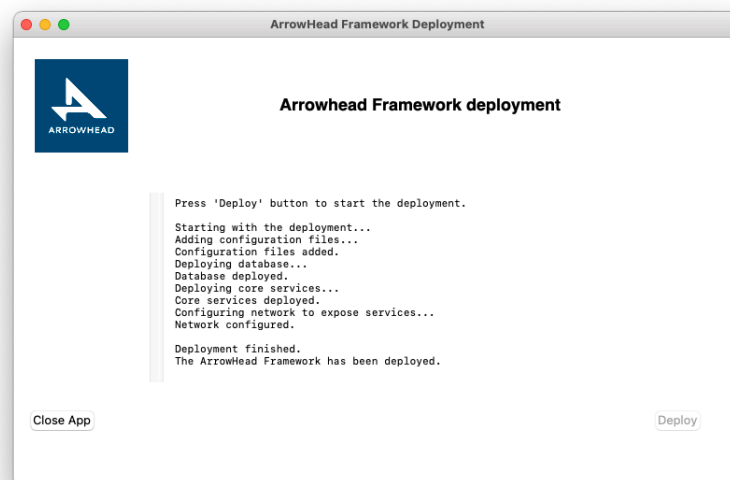


Figure 7. 13: Correct deployment logs

Now we already have the deployment done, but we can't really see it. So, we will check the cluster to see if everything is correct. To do so, we will use the `Kubectl` command to inspect all the Kubernetes resources. As mentioned before the `Kubectl` command is already configured for this cluster so we can just run it.

The next command will print all resources deployed by our application:

- `kubectl get all`

```

Last login: Mon Feb 28 11:01:04 on ttys002
alexmedela@dhcppool1:~$ kubectl get all
NAME                                READY    STATUS    RESTARTS   AGE
pod/auth-deployment-55456f5fb4-6qffj 1/1      Running   0           7m35s
pod/db-deployment-665995dfd5-8zslg    1/1      Running   0           10m
pod/orch-deployment-ff97d7844-8zk87    1/1      Running   0           8m36s
pod/sr-deployment-6fb57bdf89-hfg4j     1/1      Running   0           9m37s

NAME                                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/authorization               ClusterIP    None          <none>         8445/TCP    7m35s
service/kubernetes                  ClusterIP    10.96.0.1    <none>         443/TCP     33m
service/mysql-mariadb-galera        ClusterIP    None          <none>         3306/TCP    10m
service/orchestrator                ClusterIP    None          <none>         8441/TCP    8m36s
service/serviceregistry             ClusterIP    None          <none>         8443/TCP    9m37s

NAME                                READY    UP-TO-DATE   AVAILABLE   AGE
deployment.apps/auth-deployment      1/1      1             1           7m35s
deployment.apps/db-deployment         1/1      1             1           10m
deployment.apps/orch-deployment       1/1      1             1           8m36s
deployment.apps/sr-deployment         1/1      1             1           9m37s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/auth-deployment-55456f5fb4 1         1         1       7m35s
replicaset.apps/db-deployment-665995dfd5    1         1         1       10m
replicaset.apps/orch-deployment-ff97d7844   1         1         1       8m36s
replicaset.apps/sr-deployment-6fb57bdf89    1         1         1       9m37s

```

Figure 7. 14: Deployed resources

As we can see in the figure 7.14, all the resources have been correctly deployed. First, we can see that the `kubectl` command is connecting to the minikube cluster on the top right of the figure 7.14. We can also see the corresponding deployment and replica set of the core services and database. The command also shows that the pods are correctly running and that the service of each system is correctly configured.

If we connect to the logs of the systems, we can see they all started correctly. In the figure 7.15 we can see the logs of the service registry saying the service is up and running. At the beginning we can see that the service is starting and at the end that the service started correctly, the rest of the logs can be ignored in this case. If we check the rest of the services, we will find the same logs.

```
yMain      : Starting ServiceRegistryMain v4.4.1 on sr-deployment-6fb57bdf89-hfg4j with PID 1 (/s
serviceregistry/arrowhead-serviceregistry.jar started by root in /serviceregistry)
2022-02-28 09:56:58.201 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.s.ServiceRegistr
yMain      : No active profile set, falling back to default profiles: default
2022-02-28 09:57:30.599 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.f.ArrowheadFilter
r          : SRAccessControlFilter is active
2022-02-28 09:58:01.113 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.q.u.UriCrawlerTa
skConfig   : URI Crawler task scheduled.
2022-02-28 09:58:04.992 WARN sr-deployment-6fb57bdf89-hfg4j --- [      main] aWebConfiguration$JpaW
ebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be pe
rformed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-02-28 09:58:14.624 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] .c.s.q.t.ProvidersReac
habilityTaskConfig : Providers reachability task is not adjusted
2022-02-28 09:58:14.711 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] .c.s.q.t.ProvidersReac
habilityTaskConfig : Services end of validity task is not adjusted
2022-02-28 09:58:16.199 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.f.ArrowheadFilter
r          : PayloadSizeFilter is active
2022-02-28 09:58:26.735 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.ApplicationInitL
istener     : Core system name: SERVICEREGISTRY
2022-02-28 09:58:26.784 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.ApplicationInitL
istener     : Server mode: SECURED
2022-02-28 09:58:27.111 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.ApplicationInitL
istener     : Server CN: serviceregistry.testcloud2.aitia.arrowhead.eu
2022-02-28 09:58:28.279 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.ApplicationInitL
istener     : testcloud2.aitia own cloud is registered in SECURED mode.
2022-02-28 09:58:33.150 INFO sr-deployment-6fb57bdf89-hfg4j --- [      main] e.a.c.s.ServiceRegistr
yMain      : Started ServiceRegistryMain in 102.053 seconds (JVM running for 116.682)
```

Figure 7. 15: Service registry logs

Now we will also check the database is initialized correctly and we will do that by checking the logs and the tables. But we can also know that the database is working, because the core services are correctly initialized. If the database wasn't working correctly and the core services can't connect to it, or the tables aren't initialized the core systems would fail to start up.

In the figures 7.16 and 7.17 we can see the database logs and tables.

```

2022-02-28 09:55:51+00:00 [Note] [Entrypoint]: MariaDB init process done. Ready for start up.

2022-02-28 9:55:52 0 [Note] mysqld (mysqld 10.5.15-MariaDB-1:10.5.15+maria~focal) starting as process 1
...
2022-02-28 9:55:52 0 [Note] InnoDB: Uses event mutexes
2022-02-28 9:55:52 0 [Note] InnoDB: Compressed tables use zlib 1.2.11
2022-02-28 9:55:52 0 [Note] InnoDB: Number of pools: 1
2022-02-28 9:55:52 0 [Note] InnoDB: Using crc32 + pclmulqdq instructions
2022-02-28 9:55:52 0 [Note] mysqld: O_TMPFILE is not supported on /tmp (disabling future attempts)
2022-02-28 9:55:52 0 [Note] InnoDB: Using Linux native AIO
2022-02-28 9:55:52 0 [Note] InnoDB: Initializing buffer pool, total size = 134217728, chunk size = 13421
7728
2022-02-28 9:55:52 0 [Note] InnoDB: Completed initialization of buffer pool
2022-02-28 9:55:52 0 [Note] InnoDB: 128 rollback segments are active.
2022-02-28 9:55:52 0 [Note] InnoDB: Creating shared tablespace for temporary tables
2022-02-28 9:55:52 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing the file f
ull; Please wait ...
2022-02-28 9:55:52 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
2022-02-28 9:55:52 0 [Note] InnoDB: 10.5.15 started; log sequence number 444350; transaction id 357
2022-02-28 9:55:52 0 [Note] Plugin 'FEEDBACK' is disabled.
2022-02-28 9:55:52 0 [Note] InnoDB: Loading buffer pool(s) from /var/lib/mysql/ib_buffer_pool
2022-02-28 9:55:52 0 [Note] Server socket created on IP: '::'.
2022-02-28 9:55:52 0 [Note] InnoDB: Buffer pool(s) load completed at 220228 9:55:52
2022-02-28 9:55:52 0 [Note] Reading of all Master_info entries succeeded
2022-02-28 9:55:52 0 [Note] Added new Master_info '' to hash table
2022-02-28 9:55:52 0 [Note] mysqld: ready for connections.
Version: '10.5.15-MariaDB-1:10.5.15+maria~focal' socket: '/run/mysqld/mysqld.sock' port: 3306 mariadb.
org binary distribution

```

Figure 7. 16: Database logs

```

$ kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql-mariadb-
galera -proot
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| arrowhead |
| information_schema |
| mysql |
| performance_schema |
+-----+
4 rows in set (0.00 sec)

mysql>

```

Figure 7. 17: Databases

In the figure 7.16 we can see the log logs saying the database is running and ready for connections. And in the figure 7.17 we connected to the database and checked if the arrowhead database was created. As we can see, the database is correctly initialized and running.

Finally, we will check the ambassador ingress controller that didn't show up in the first kubectl command. This is because it is deployed in a different namespace. To see these resources, we will run the next command:

- `kubectl get all --namespace=ambassador`

```

$ kubectl get all --namespace=ambassador
NAME                                     READY   STATUS    RESTARTS   AGE
pod/ambassador-agent-5fd9dbd766-4shds   1/1     Running   0           4m27s
pod/ambassador-f8dcf9f9b-rb7rl          1/1     Running   0           4m28s
pod/ambassador-redis-584cd89b45-4jnd9    1/1     Running   0           4m29s

NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)                                AGE
service/ambassador                  LoadBalancer  10.106.206.96  10.96.0.15     80:31911/TCP,443:32722/TCP            4m28s
service/ambassador-admin            ClusterIP     10.98.186.89   <none>         8877/TCP,8005/TCP                     4m28s
service/ambassador-redis            ClusterIP     10.111.101.153 <none>         6379/TCP                               4m29s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ambassador           1/1     1             1           4m28s
deployment.apps/ambassador-agent     1/1     1             1           4m27s
deployment.apps/ambassador-redis     1/1     1             1           4m29s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/ambassador-agent-5fd9dbd766 1         1         1       4m27s
replicaset.apps/ambassador-f8dcf9f9b        1         1         1       4m28s
replicaset.apps/ambassador-redis-584cd89b45 1         1         1       4m29s

```

Figure 7. 18: Ambassador resources

In the figure 7.18 we can see the ambassador resources. Here we have different replica sets, deployments, services, and pods. This is how the ingress controller works so we won't enter into details. The important part from these resources is the service of type LoadBalancer, which is the one that will expose the services to the outside of the cluster.

As we can see in the figure 7.18 this load balancer has an external ip (10.96.0.15). This is the one we will use to connect to the cluster following the next syntax: `https://external_ip/system_name`.

Now we can connect to these services just by using the previous syntax. But if you are running this example in MacOS you will need an extra step, and this is because of how docker networks works on MacOS. To expose this service to a MacOS we will need to run `'minikube service ambassador --url --namespace=ambassador'` and use the URL offered in the namespace ambassador for the name ambassador (This is shown in the figure 7.19).

```

$ minikube service ambassador --url --namespace=ambassador
Starting tunnel for service ambassador.

+-----+-----+-----+-----+
| NAMESPACE | NAME   | TARGET PORT | URL                                     |
+-----+-----+-----+-----+
| ambassador | ambassador |           | http://127.0.0.1:52626                |
|            |         |           | http://127.0.0.1:52627                |
+-----+-----+-----+-----+
http://127.0.0.1:52626
http://127.0.0.1:52627
! Because you are using a Docker driver on darwin, the terminal needs to be open to run it.

```

Figure 7. 19: Exposing cluster service to MacOS

3.3.2. Verification

So far, we know all the system is working correctly, but we didn't test if it's fulfilling its purpose. To do so, we will run the consumer/provider test that will check if the cloud is arrowhead compliance.

As mentioned in previous chapters the test was containerized for the automation purposes, so this makes this test easier to run for our use case. We will only need to deploy a pod with the docker image with the consumer/provider test and check the result in the logs.

To do so we will execute the next command:

- `kubectl run --image=alexmedela/test_image --restart=Never test1`

This command will create a pod with the test image running the provider/consumer test that will never be restarted (this is important, because if we don't set this flag Kubernetes will always try to execute the container after finishing and will enter an infinite loop). After the test is finished, we will check the logs with the command 'Kubernetes logs test'.

```
> GET /helloworld HTTP/1.1
> Host: 127.0.0.1:9981
> User-Agent: curl/7.79.1
> Accept: */*
>
{ [5 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [1081 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [1081 bytes data]
* old SSL session ID is stale, removing
{ [5 bytes data]
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 ok
< Content-type: text/plain
<
{ [23 bytes data]
* Closing connection 0
} [5 bytes data]
* TLSv1.3 (OUT), TLS alert, close notify (256):
} [2 bytes data]
{
  "hello": "world"
}
```

Figure 7. 20: Test logs

Finally, in the figure 7.20 we can see the last logs of the test. In these logs we can see that the message 'hello world' has been correctly received meaning the cloud works correctly and it's an arrowhead compliance cloud.

4

Conclusions

This project has been a long journey with a lot of new experiences, working and learning. So, this chapter will summarize the conclusions of the development of the project.

4.1 Conclusions

The development of this project has been a long new experience. This project was an ambitious project where I learned a lot about DevOps engineering; pipelines, IaC automation, Terraform, Kubernetes, Docker, automation scripting, Workflow management, working with international teams, etc. The project was mostly developed on my own, but with a lot of help and support from the team.

The project fulfilled the automation of the base deployment of the arrowhead framework and opened a lot of possibilities for new projects.

Based on the objectives we fulfilled almost all of them except the CI pipeline which was supposed to maintain the last working version of the project always available. And this is obviously one of the future lines that will be worked in the future.

Finally, all scopes were accomplished and the work was almost finished. Even though the pipeline wasn't finished due to time problems generated by the laptop screen inconvenience it was properly design and prepare for the implementation.

The accomplished technical scopes are the next ones:

- M1N110 Define, design, and implement scalable, flexible, and resilient architectures to address existing issues and deploy existing applications faster: This scope is fulfilled using Docker, Kubernetes and Terraform to deploy different types of architectures.
- M1N303 Computer tools for the development of applications and operations (DevOps), both locally and in the cloud, to solve complex problems and carry out engineering projects, considering the commercial and industrial context: This scope is also fulfilled using Docker, Kubernetes and Terraform, but in this case using them to automate the deployment.
- M1N401 Ability to work in multidisciplinary teams and in a multilingual environment (Basque / Spanish / English) and be able to communicate knowledge, procedures, results, and ideas related to data life cycle, cybersecurity, and development and operations, both orally and in writing:

Finally, this scope is fulfilled by the doing the work on a multidisciplinary team with a multilingual environment like it was in this case the AITIA headquarter. This project was developed with the helped of the AITIA teams where we had to communicate, discuss ideas or procedures, etc.

Finally, during the project I also learned to work with international teams and with clients. I also learn to adapt to new work environments. In conclusion, this project was helpful for my working career as an engineer.

5

Future lines

The project opened the door to new possibilities of projects and improvements of the actual one. This chapter will talk about all these new possibilities for new ideas.

5.1 Future lines

As we mention before the project is not on its final version meaning there are a lot of improvement or features that could be added. Between these improvements and new features, the most interesting ones are the next ones:

- Implement the pipeline defined to automate the containers updates. This can be done by creating a pipeline with GitHub actions that would create the docker image, test the image and deliver it with the latest version so the project would start using this new version.
- Implement a small guide showing how make a test deployment as done in this project.
- Add the option to connect to an existing database,
- Specify the needed resources for each case of deployment.
- Add the option to create the cluster and deploy the cloud on AWS.
- Join this project GitHub repository with the arrowhead main repository.

The project also opened the door to a very interesting new project. This project would be an application using this project as based, but with the objective of not only deploying new clouds but also to manage them in real time.

This new application would connect to the cluster and show information of the status of the application like, the deployed services, the version they are using, if they are alive or down, etc. It should also give the option to deploy new services, destroy them, update them. This project would be an update to not also deploy the service, but also monitor and update it after the deployment.

For this new project the current project would be a small section of it that would allow to create a new deployment, and the new one should let to connect to an existing one.

Basically, this new project would be the next big step, the automation of the arrowhead framework cloud management.

Annex I

| October | Summary |
|----------------------|--|
| Week 1 (4Oct-8Oct) | |
| Week 2 (11Oct-15Oct) | Learning what is the Arrowhead Framework, the concepts and how it works. |
| Week 3 (18Oct-22Oct) | Deploying an example network of the Arrowhead Framework with docker and testing how this network works. |
| Week 4 (25Oct-29Oct) | Started creating an automated 'consumer/provider' test using docker containers (not finished yet). Started defining the default infrastructure of the network. Started doing some research and comparing technologies to deploy the infrastructure. |
| November | |
| Week 1 (1Nov-5Nov) | Finished automating the test using docker containers. Finished deploying the arrowhead docker network with core services. Project management updates. |
| Week 2 (8Nov-12Nov) | Started analyzing container orchestrators for the deployment. Started defining the architecture solution for kubernetes. Started defining the architecture solution for docker swarm. Started learning about Nomad. |
| Week 3 (15Nov-19Nov) | Defined the architecture to be use with kubernetes and nomad. Started creating the deployment for kubernetes. Started creating the job for nomad. Desing the GUI for the user deployment. Started comparing infrastructure provisioning tools. |
| Week 4 (22Nov-26Nov) | First deployment of core services on kubernetes working. Started defining different kubernetes architectures for different use cases. Changes to the desing for the GUI. |
| December | Summary |
| Week 1 (29Nov-3Dec) | Started creating the GUI to deploy the simple kubernetes architecture on localhost. Started creating the cluster database for the high availability deployment. Started defining the scheme for the Thesis documentation. |
| Week 2 (6Dec-10Dec) | Started adding the optional services to the kubernetes deployment Started automating the kubernetes deployment by creating scripts |
| Week 3 (13Dec-17Dec) | Finished creating the database cluster for kubernetes Started creating production container for core services Making changes to the GUI interface First phase of the interface done |
| Week 4 (20Dec-24Dec) | |
| Week 5 (27Dec-31Dec) | |
| January | Summary |
| Week 1 (3Jan-7Jan) | |
| Week 2 (10Jan-14Jan) | Started trying to implement the management tool to the network Testing the new containers on docker Adding new containers to kubernetes |
| Week 3 (17Jan-21Jan) | Making changes on the management tool development (sventlint will continue) Adding optional services to Kubernetes deployment. |
| Week 4 (24Jan-28Jan) | Creating ingress controller for the kubernetes deployment Finished automating all kubernetes resources by scripts |
| February | Summary |
| Week 1 (31Jan-4Feb) | Started learning about terraform Testing how terraform works |
| Week 2 (7Feb-11Feb) | Started and finished creating the cluster for azure with terraform Started creating the cluster for google with terraform |
| Week 3 (14Feb-18Feb) | Finished creating the cluster for google with terraform Started with the second phase of the GUI |
| Week 4 (21Feb-25Feb) | Finished the GUI Started and finished the repo documentation |
| March | Summary |
| Week 1 (28Feb-4Mar) | First version of the project report finished |
| Week 2 (7Mar-11Mar) | |
| Week 3 (14Mar-18Mar) | |
| Week 4 (21Mar-25Mar) | Presentation date |

Annex I : Weekly book log of the project where all weekly development is defined.

Annex II



Annex II: Gantt diagram of the project where we can see all the tasks and deadlines.

Bibliography

- HashiCorp. (n.d.). *A Kubernetes User's Guide to HashiCorp Nomad*. Retrieved from HashiCorp: <https://www.hashicorp.com/blog/a-kubernetes-user-s-guide-to-hashicorp-nomad>
- Aqua. (n.d.). *Docker Containers vs. Virtual Machines*. Retrieved from Cloud Native Wiki: <https://www.aquasec.com/cloud-native-academy/docker-container/docker-containers-vs-virtual-machines/>
- Liccardi, A. T. (n.d.). *Nomad vs Kubernetes without the complexity*. Retrieved from Codemotion: <https://www.codemotion.com/magazine/dev-hub/backend-dev/nomad-kubernetes-but-without-the-complexity/>
- inc, K. (n.d.). *Kubernetes refence doc*. Retrieved from Kubernetes: <https://kubernetes.io/docs/reference/generated/kubect/kubectl-commands>
- John, K. (n.d.). *Install Ambassador Ingress Controller on Kubernetes*. Retrieved from Computing for geeks: <https://computingforgeeks.com/install-ambassador-api-gateway-kubernetes/>
- Sharif, A. (n.d.). *Severalnines*. Retrieved from Running Galera Cluster on Kubernetes: <https://severalnines.com/database-blog/running-galera-cluster-kubernetes>
- Petersen, M. (n.d.). *Platform9*. Retrieved from Ultimate Guide to Kubernetes Ingress Controllers: <https://platform9.com/blog/ultimate-guide-to-kubernetes-ingress-controllers/#differences>
- Madushanka, M. (n.d.). *Wordpress*. Retrieved from How to deploy a MySQL Cluster from Scratch with Docker: <https://menakamadushanka.wordpress.com/2017/12/15/how-to-deploy-a-mysql-cluster-from-scratch-with-docker/>
- MariaDB. (n.d.). *MariaDB*. Retrieved from Why MariaDB? Advantages over MySQL: <https://mariadb.com/resources/blog/why-should-you-migrate-from-mysql-to-mariadb/>
- HashiCorp. (n.d.). *HashiCorp Learn*. Retrieved from Terraform Tutorials: https://learn.hashicorp.com/terraform?utm_source=tf_registry&utm_content=sidebar
- Mitevski, K. (n.d.). *Learnk8s*. Retrieved from Provisioning Kubernetes clusters on GCP with Terraform and GKE: <https://learnk8s.io/terraform-gke>
- Delsing, J. (2017). *IoT Automation with Arrowhead Framework*. Boca Raton: CRC Press, Taylor & Francis Group.
- Varga, P., Blomstedt, F., Ferreira, L. L., Eliasson, J., Johansson, M., Delsing, J., & Martínez de Soria, I. (2016). Making system of systems interoperable The core components of the arrowhead framework. *CISTER Research Centre*, 2-13.