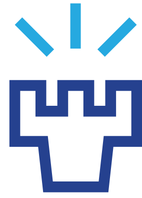


FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Lauri Suutari**
**Joona Holappa**

# FUTURE PROOFING LOVELACE SYSTEM DEVELOPMENT ENVIRONMENT

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
June 2022

**Suutari L., Holappa J. (2022) Future Proofing Lovelace System Development Environment.** University of Oulu, Degree Programme in Computer Science and Engineering, 44 p.

# ABSTRACT

**Software development methods and tools improve continuously to improve the development process. Modern software architecture has paved the way for microservice based architecture. The main point of microservice architecture is to split a system to small independent parts that can be deployed separately without affecting the other parts of the system. With microservices and tools, a system can achieve fault tolerance, scalability and faster release cycle with automation. The use of container technologies has increased and popularized with microservices, because containers simplify the deployment process.**

**In this project, a modern development environment was introduced to help future development of a Virtual Learning Environment. The development environment included a public repository, containers, a container registry, container orchestration, server configuration and automated deployment. After successful implementation the simple mock up system was tested by smoke, load and spike testing methods. Overall the implementation and configuration was successful, however for implementing it for the Lovelace system in University of Oulu's environment, some configuration and tool choices may need to be changed.**

**Keywords: Containers, container registry, container orchestration, Docker, Kubernetes, Ansible, CI/CD, automation**

**Suutari L., Holappa J. (2022) Lovelace järjestelmän modernisointi tulevaisuuden kehitykseen.** Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 44 s.

## TIIVISTELMÄ

**Ohjelmistokehityksen tavat ja työkalut kehittyvät jatkuvasti helpottamaan, sekä parantamaan ohjelmiston kehitysprosesseja. Moderni ohjelmistoarkkitehtuuri on luonut tietä mikropalveluarkkitehtuurille, jonka päätarkoituksena on pilkkoa järjestelmä pieniin lähes itsenäisiin osiin, joita voidaan erikseen kehittää vaikuttamatta järjestelmän muihin osiin. Mikropalveluiden ja muiden työkalujen avulla järjestelmä saavuttaa vikasietoisuutta, skaalautuvuutta sekä nopeamman julkaisusyklin automaation ansiosta. Konttiteknologioiden käyttö on myös yleistynyt mikropalveluiden myötä, jotka helpottaa ohjelmiston toimittamista servereille. Tämän projektin aikana implementoitiin moderni kehitysympäristö helpottamaan jatkokehitystä Lovelace systeemille.**

**Kehitysympäristö sisälsi julkisen säilön, kontin, kontti rekisterin, konttien orkesterointi työkalun, serveri configuroinnin ja automaattisen sijoituksen. Onnistuneen implementaation jälkeen, yksinkertainen järjestelmä testattiin savu, kuorma ja piikki testi metodeilla. Kokonaisuudessaan implementaatio ja configurointi onnistuivat, mutta Lovelace implementaatio Oulun Yliopiston ympäristöön vaatii configurointi muutoksia ja mahdollisesti muutamien työkalujen vaihtamista.**

**Avainsanat: Kontti, kontti arkisto, kontti orkestrointi, mikropalvelu, Docker, Kubernetes, Ansible, CI/CD, automatisointi**

# TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION ... 8
2. LOVELACE ... 9
  2.1. Current Technologies Used in Lovelace ... 9
    2.1.1. Django ... 9
    2.1.2. RabbitMQ ... 9
    2.1.3. Redis ... 10
    2.1.4. Celery ... 10
    2.1.5. PostgreSQL ... 10
    2.1.6. NFS ... 11
    2.1.7. Apache ... 11
    2.1.8. Red Hat Enterprise Linux ... 11
  2.2. Issues with the Current Lovelace Environment ... 11
    2.2.1. Development Challenges ... 11
    2.2.2. Checker ... 12
3. RELATED WORK ... 13
  3.1. Virtual Learning Environments ... 13
  3.2. Microservice Architecture ... 14
  3.3. Automation ... 15
  3.4. Continuous Integration and Continuous Deployment ... 16
4. TECHNOLOGIES ... 17
  4.1. Configuration Tools ... 17
    4.1.1. Jenkins ... 17
    4.1.2. Ansible ... 18
    4.1.3. Chef ... 18
    4.1.4. Puppet ... 19
  4.2. Containers ... 20
    4.2.1. Docker ... 20
    4.2.2. Containerd ... 20
    4.2.3. CRI-O ... 20
  4.3. Container Registries ... 21
    4.3.1. Docker Hub ... 21
    4.3.2. Github Packages ... 21
    4.3.3. Private Registries ... 21
  4.4. Container Orchestration ... 21
    4.4.1. Kubernetes ... 22
    4.4.2. K3s and K3d ... 22
    4.4.3. Docker Swarm ... 22
  4.5. Use Cases ... 23

4.5.1. Netflix ........ 23
4.5.2. Apex Legends ........ 23
4.6. Requirements ........ 23
4.7. Chosen Tools and Technologies ........ 24
4.8. Risk Assessment ........ 25
5. IMPLEMENTATION ........ 26
5.1. Code ........ 26
5.2. Docker ........ 27
5.3. Github and Pipelines ........ 27
5.4. Docker Hub ........ 28
5.5. Ansible ........ 28
5.6. Kubernetes ........ 28
5.7. Networking ........ 28
5.8. Risk Assessment ........ 29
6. EVALUATION ........ 30
6.1. Evaluation Plan ........ 30
6.2. Testing the Cluster for Scalability and Fault Tolerance ........ 30
6.2.1. Input ........ 30
6.2.2. Smoke Testing ........ 30
6.2.3. Load Testing ........ 31
6.2.4. Stress and Spike Testing ........ 31
6.2.5. Desired Output ........ 32
6.2.6. Results ........ 33
6.3. Data Analysis ........ 34
6.3.1. Expected Outcome Vs Reality ........ 34
6.3.2. Major Problems, Shortcomings and Flaws ........ 34
7. DISCUSSION ........ 35
7.1. The Change ........ 35
7.2. Difficulties ........ 35
7.3. Future Work ........ 36
7.4. State of the Art ........ 36
8. CONCLUSION ........ 37
9. REFERENCES ........ 38
10. APPENDICES ........ 41
10.1. Commands ........ 41
10.2. Contributions ........ 42
10.3. Flaskapi ........ 44

# FOREWORD

This Bachelor's thesis was created for University of Oulu's course Applied Computing Project 1, 521041A. We want to thank Mika Oja and Timo Ojala for supervising and inspecting this work. We would also like to thank Miiro Kuosmanen who acted as the project manager and was heavily involved in the development of this project.

Oulu, June 7th, 2022

Lauri Suutari
Joona Holappa

# LIST OF ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| CSE | Computer Science and Engineering |
| VLE | Virtual Learning Environment |
| CI | Continuous Integration |
| CDE | Continuous Delivery |
| CD | Continuous Deployment |
| DevOps | Development and Operations |
| IaaS | Infrastructure as a Service |
| STEM | Science, technology, engineering and mathematics |
| OS | Operating System |
| VUs | Virtual Users |

# 1. INTRODUCTION

New software development technologies and tools are popping up constantly and when choosing the best tools for your development, thorough research must be conducted. High availability, performance and fault-tolerance are some key factors most companies strive to have in their software. Some tools and technologies are solely created to make sure that if one instance of the service fails, the whole service does not crash. Others were created to make the utilization of some components easier for the software developer. Many companies aim to automate procedures to maximize output since most services can be automated for better results [1]. These definitions fall under microservice orientated architecture solutions. Microservices aim to allow developers to deploy code changes independently with shorter intervals, more reliability, fault-tolerance and efficiency. [2]

In its current state, updating Lovelace is difficult due to the current monolithic architecture. This causes current updates to practically revolve around simple bug fixing and the bigger updates have to be done when the course is on hiatus. The benefits of using microservice architecture will help Lovelace overcome some of the current issues.

This thesis focuses on showcasing how different technologies could be used to automate the process of application development. The thesis will provide an alternative way to deploy the applications which are in line with the modern 'agile' mentality. The motivation behind the project was to introduce different state of the art technologies that could be used in the modernization of Lovelace, which is a virtual learning environment.

# 2. LOVELACE

Lovelace is a web-based Virtual Learning Environment used for programming courses such as Elementary Programming, Programmable Web and Computer Systems. It is mainly used by the Faculty of Information Technology and Electrical Engineering in University of Oulu. It is based on Django, which is a web framework that uses Python programming language. In addition to Django, Lovelace also uses technologies such as Apache, Nginx, Redis, Postgres, RabbitMQ, NFS and python modules like Celery. Figure 6 shows the high level architectural design of Lovelace system with mentioned technologies.

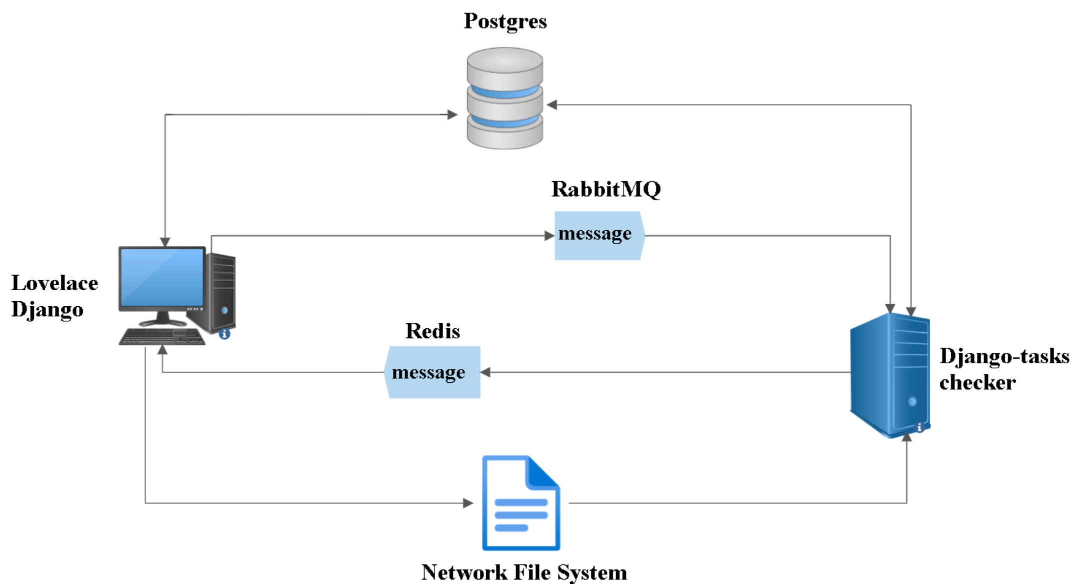


Figure 1. Lovelace system architecture

## 2.1. Current Technologies Used in Lovelace

### *2.1.1. Django*

Django is one of the most popular high-level web frameworks and it focuses on rapid development with clean and reusable templates due to its model-template-view architectural pattern. Lovelace uses Django framework for developing the user interface and backend logic. [3]

### *2.1.2. RabbitMQ*

Modern commonly used open source messaging broker based originally on Advanced Message Queuing Protocol(AMQP). RabbitMQ supports multiple other messaging protocols like Streaming Text Oriented Messaging Protocol(STOMP) and MQ Telemetry Transport(MQTT). Lovelace uses RabbitMQ to queue messages for the Checker. [4]

### *2.1.3. Redis*

Remote Dictionary Server or Redis for short is a fast, open source in-memory data structure store most commonly used as a database, message broker and cache. Low latency and high throughput data access is achieved because all the data handled by Redis is stored in memory. Redis also allows the usage of flexible data structures such as strings, lists and sets, to name a few. Because developers can use a simple command structure, Redis enables developers to write fewer lines of code.

It decreases latency, increases throughput and eases the load on the main PostgreSQL database. This is achieved because Redis stores frequently used objects in memory for fast access. If the accessed object is not found in cache it is queried from the Database. [5]

In Lovelace Redis is used to transmit results of student code checking from Checker to main server and it works as the cache of the system.

### *2.1.4. Celery*

Celery is a open source Python-based application that handles task queues with focus on real-time processing. Task queues are mechanisms that distributes work across threads or machines. This enables multiprocessing of tasks asynchronously (in the background) or synchronously.

Celery uses workers, which are task-based servers on the background to complete long-running tasks so that the main server can operate at optimal speed. Celery workers pick tasks from the queue to process. This is useful because the modern web users have high expectations regarding the loading times of web pages. At worst case scenario long-running tasks could slow down the loading time of the website by minutes. Essentially Celery helps in the horizontal-scaling of the system.

Instagram uses Redis as a cache for user feed, RabbitMQ as the message broker and Celery as a task manager that delivers the posts to the user's view. Django-celery brings the possibility for Celery integration to Django framework. [6]

### *2.1.5. PostgreSQL*

PostgreSQL is a powerful open source database which contains many features. It is a Relational Database Management System (RDMBS), which means it stores the data as tables. PostgreSQL is currently one of the most popular RDMBS's because it is compatible with many of the most popular programming languages, works on popular operating systems and it has good scalability. Lovelace uses PostgreSQL to store data. For example, Apple, IMSB and Instagram use PostgreSQL as the company's databases. [7]

### *2.1.6. NFS*

Network File System (NFS) is used in Lovelace to allow the checker to access to the code files that are to be executed for checking purposes. [8]

### *2.1.7. Apache*

Apache is the dominating web-server for developers, hosting providers and website owners. Web server is responsible for accepting HTTP requests and responding to them with HTTP responses, they have a pivotal role in all the user interactions on web-based applications. Web servers offer access to documents stored at them to the web clients with the help of a web browser. Apache uses TCP/IP protocol to communicate from clients to servers over the network. It handles the process requests and delivers web assets and content via the HTTP protocol. Apache uses PHP as the programming language that creates dynamic web content, but other modules (mod wsgi) allow it to be used it different languages such as Python. Apache is still the most used web-server, but others have started gaining popularity among developers. One of those being Nginx, which was designed to handle high traffic websites. Apache is still the most compatible web-server across all the web software. [9]

### *2.1.8. Red Hat Enterprise Linux*

Red Hat Enterprise Linux (RHEL) is a very popular open source enterprise Linux operating system. One feature of RHEL is to reduce issues in the deployment. RHEL is very reliable and secure. Lovelace uses RHEL the operating system. Yellowdog Updater Modified(YUM) is the most relevant package management tool used in RHEL. YUM allows easy management of packages in Linux via the command line. YUM is dependent on Redhat Pacakage Manager (RPM) which is another popular package management tool used in RHEL. The biggest difference between YUM and RPM is the possibility to control dependency resolution that only YUM can do. Both of them are good tools for RHEL operating systems to install, remove and update packages. Dandified YUM (DNF) is currently the gaining reputation as it resolves the slowness that is caused by the iterative dependency resolution by using a dependency resolver libsolv. [10]

## 2.2. Issues with the Current Lovelace Environment

### *2.2.1. Development Challenges*

The original system named Raippa was created in 2008 and by 2011 it was updated to Raippa 2.0 as a master's thesis project. In 2015 the system's layout was changed, new features were added and it was renamed to Lovelace. The current system uses mostly the same code, software architecture and technologies as back then. The current system is very monolithic, which means it has a large mainframe with no modularity.

Sometimes monolithic applications are better for the system, but the biggest difficulty in the current system comes from the fact that larger updates cannot be done during the active period of courses, and the fixes during those times revolve around small bug fixes and User Interface changes.

In the 14 years after the creation of Lovelace lots of new technologies have emerged that could be used to make the development environment more modular and code changes to the main system easier. The rise of CI/CD mentality in software production has also increased the number of options that could make the overall development environment more modern. This is also why there isn't a proper deployment process or testing pipelines for the current system. The current deployment process is very tedious, since the documentation is outdated, with many of the dependencies not developed since the original launch of Lovelace. Testing the environment is hard, because there are a lot of different scenarios that could happen in a VLE system like Lovelace. Technologies that help address these problems will be introduced in this project to give examples of how certain tasks in the Lovelace environment could be handled differently.

### *2.2.2. Checker*

The current student code checking system, has too many privileges in the overall system. This is due to the Checkers needing to know the architecture of the main server's database to find a list of required files and executable commands. In addition to this it needs these privileges to write the results to the database. This means that the directory that the Checker is in has to be restricted so that the whole environment doesn't get injected in case of an outer threat. The checker processes privileges are lowered in two steps. In the first step the checkers only have the privileges to create temporary checking folders, find the required files and commands for the checking. After the second step the checker can only read the temporary folders and Python libraries and it only has the privileges to write and execute within those folders.

Because Checker has many dependencies, code changes made elsewhere can indirectly affect the Checker as well. Currently there are two checker processes running on the same server. This means that if the server crashes, both of the checkers will crash as well. Furthermore, in the current system there isn't any process to check if the checkers are running, so in case a crash happens the VLE will in effect be unable to work until somebody notices this and informs the system administrator and they have to manually restart the checker server.

# 3. RELATED WORK

This section elaborates some key words and technologies that are related to this thesis. The sections aim at giving a basic understanding of what Lovelace is and what technologies are currently in use. It will start by introducing what a virtual learning environment is and then go into more specifics regarding the architecture of Lovelace.

## 3.1. Virtual Learning Environments

Ever since the start of the Digital Revolution the demand for competent programmers and developers has been on the rise. One of the most challenging part about programming is how to teach and learn it. One of the modern ways of teaching programming is via virtual learning environments.

The amount of data to be processed in these virtual learning environments is increasing constantly. It is vital that the deployment and configuration of the web servers gets automated to minimize the effort needed to get the web servers installed and configured again due to any reason.

A virtual learning environment (VLE) is a collection of integrated tools used to aid in online learning on a web-based platform. With VLE teachers can interactively teach courses via a distance learning format. The need of proper virtual learning environments has risen greatly during the COVID-19 pandemic due to the need of social distancing [11]. That is not to say that there has not already been a virtual learning environment in use, but rather that the need of keeping the virtual learning environment up to date has risen immensely. For example, the commitment from professors on updating the information, and monitoring the students' learning in the platform is required. This added with the fact that one of the most important aspects in regards to developing a teaching tool is to have a clear image of the end users are using the platform for. Different types of online learning environments need a different approach to the architecture. When the VLE is aimed at teaching programming, the developer(s) can expect the end user to be considerably fluent in the technological tools affiliated with the VLE.[12] VLEs also offer the courses to have a larger capacity. The capacity of students allowed on the course is theoretically only limited by the VLE's server capacity. [13]

To ensure that the learning does not suffer when being done in a VLE compared to a physical environment, proper designing of the VLE is crucial. This designing process includes that the facet in charge of the VLE has the proper knowledge on how to build and maintain the VLE. The possibilities that come through the virtualization of courses can play a role in the students' performance [14] and therefore it is of importance for the developers of the VLE to choose the correct tools for it. For example, the deployment of the web server can be done via microservices. Microservices are services that can be deployed solely without needing to deploy the whole web server again. When changing from a monolith architecture to microservice architecture the singular parts need to be decoupled so that the updating, configuring and re-deploying can happen without major issues. [15]

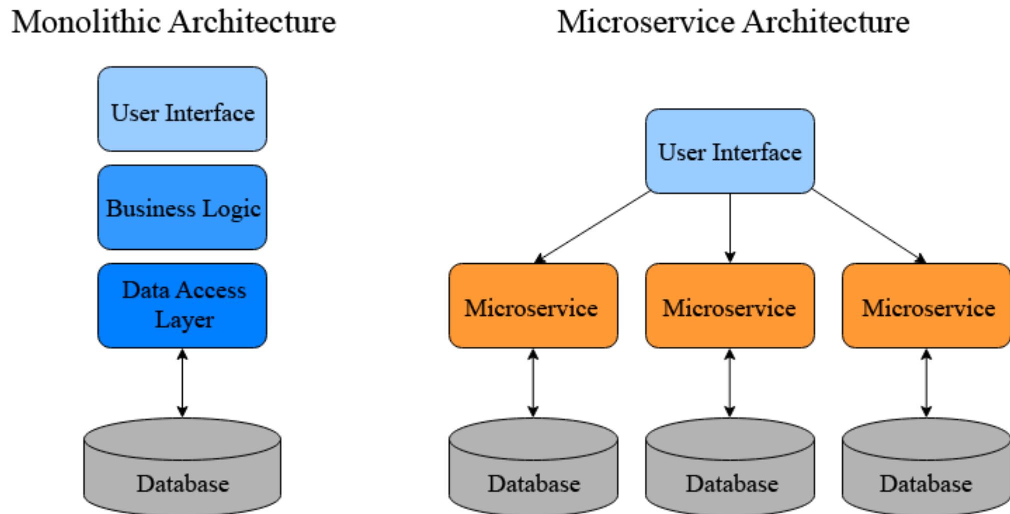


Figure 2. The differences between Monolithic- and Microservice Architectures

### 3.2. Microservice Architecture

Monolithic architecture in software development means that the application runs on a single code base. This means that it is composed of one piece and all the components of the program are interdependent and interconnected to each other. To update an application created in this manner, you may have to rewrite code unrelated to the update or even the whole application. Applications designed with this approach have to be fully deployed every time there is a change to the code. On the other hand we have Microservice Architecture, which focuses on dividing the application into multiple services - microservices. Microservice architecture has many upsides to it such as reduced risk of negative changes within other microservices when deploying new code. These microservices can be tested, developed, scaled and deployed independently, which increases the efficiency and speed of production. As long as there is a common interface for communication between the microservices, the programming language between them can also be different.

Microservices can be used in Virtual Learning Environments to separate different functions of the service into smaller sections. In the STIMEY platform, which was created as an e-learning platform for STEM students, they divided the web-service into microservices such as user profile, activities dashboard, messaging service and community section. The communication between these microservices was handled by calling their respective REST-API(s). The microservices were deployed in Docker containers, which increase failure safety, since the containers were independent to each other. Maintaining an own solution to microservice architecture and container handling is very strenuous. At the time this report was written many open-source alternatives already exist. One such alternative is Kubernetes, which is a platform for managing containerized workloads and services. [16]

Containerization is another modern solution in software development. It allows developers to test and deploy software across environments with ease. This is achieved because containers hold the application and all its the dependencies and configurations

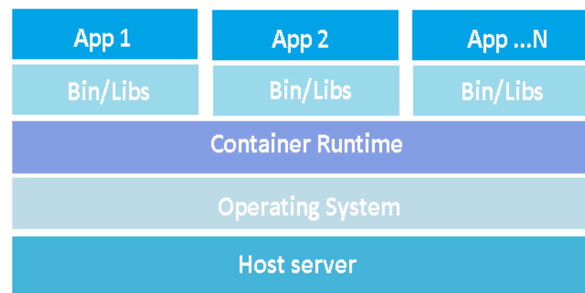


Figure 3. Example of a Container applications and their general structure

in the same image. Containers abstract away differences between different platforms and allow developers to run the software regardless of the differences in operating systems. This means that a containerized application does not require other users to download the required dependencies to execute it. Applications like Docker are used as container hosts. Containers are similar to virtual machines, but instead of holding a complete operating system, they use the host system's features and resources. This allows them to be faster and more lightweight than their virtual machine counterparts. Netflix is one of the biggest companies that has been able to successfully migrate from monolithic based architecture to microservices architecture. The reason Netflix opted for the change was that the amount of data and user information was starting to outgrow the capability of their data centers [1].

### 3.3. Automation

Automation can be defined as the use of technologies to reduce human involvement which can improve the processes and their outcomes. Currently it is viewed as a contributor for improved productivity and economic growth, however majority of the public view automation in an anxious manner due to the fact that it has reduced the amount of jobs in the market. In 2015 Committee for Economic Development of Australia (CEDA) reported that automation could replace nearly 40 percent of the Australia's workforce in the next decade. Automation can be divided into four different aspects of research that are redesigning of processes, deployment and maintenance of automated systems, as a stimulant for innovation and to redefine human skills with automation technologies. It can also be divided into three different classifications that are fixed automation, programmed automation and flexible automation. Fixed automation is focused on technology and machines that replace a task that consists of a fixed sequence of actions such as painting. In contrast programmed automation is designed to accommodate a different sequence of activities such as batch production. Flexible automation learns and adapts to events relevant to it, for example a call center. [17]

Automation can be used to ease the efforts needed by humans. An example of a simple but useful automated process is the customized seat control in vehicles with powered seat controls. When a user opens the car doors with a specific remote key, the computer remembers the last used / preferred setup of the driver with the key and automatically changes the seat positioning accordingly. [18]

### 3.4. Continuous Integration and Continuous Deployment

Continuous software engineering is a combination of Continuous Integration (CI), Continuous Delivery (CDE), and Continuous Deployment (CD). And its purpose is to develop, deploy and acquire feedback in a quick succession from the software and the customer. Continuous integration refers to the practice where a development team routinely merges development work. Continuous Integration in software development practice includes automated software building and testing. Using CI aids software companies by allowing shorter and more frequent release cycles. These factors improve software quality and productivity. Continuous Delivery and Continuous Deployment are very similar in nature, both make sure the application is always passing automated tests to be in a production-ready state. This enables rapid feedback from customers and the users. Continuous Deployment however goes even further and continuously deploys every single change made in the code to the production environment. This increases production speed and efficiency, since code that passes the quality assurance tests is considered approved to production. CD also is a completely automated practice where as CDE requires developers to manually submit and accept the final deployment.[19]

Before Continuous Deployment (CD), it was common that software was traditionally released every couple of months. Now in these rapidly changing markets, it is important for the companies to frequently update their software to satisfy the needs of the customer market. Through CD it is less likely that major bugs get published as less changes to the source code are made at once. It is also easier to recall to the last version that worked and then start debugging. Traditionally if there was a bug that got past the Quality Assurance, the software had to be rolled-back and it would have taken significantly more time to find and correct the issue as more changes had been deployed.[20]

# 4. TECHNOLOGIES

The purpose of this project is to propose and showcase the usage of new technologies in the next version of Lovelace. The main focus of the new technologies introduced will be to improve the development cycle of an application by introducing containers, automatic deployment and configuration of the application. As mentioned before containers were brought to the project to bring portability, lower system requirements and to bring forth a more consistent DevOps environment. A containerization platform simplifies the managing of containers at a larger scale. These container orchestration tools automate the deployment, scaling and networking of containers. A virtual repository needs to be created to store the containers, this is called a container registry. Pipelines are introduced to aid in building and deploying automation as well as bring a CI/CD mentality in to the project. [21]

This chapter goes through the options that were considered regarding each aspect of the project and at the end there is a conclusion on why certain tools were chosen.

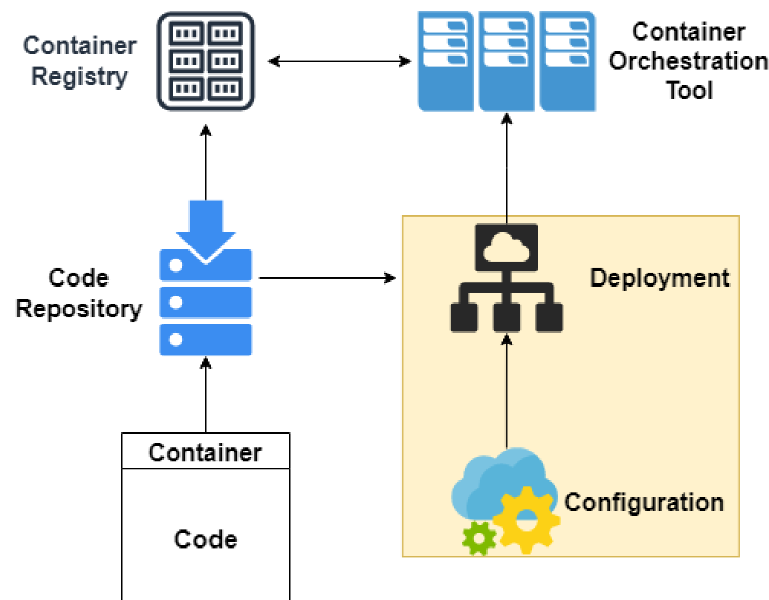


Figure 4. Design plan

## 4.1. Configuration Tools

### *4.1.1. Jenkins*

Jenkins is an open source automation server, which has been used for Continuous Integration. It was created to be an integration environment for software developers to run their applications on. Developers use Jenkins to build and test applications. Jenkins can also be used to deploy code onto shared developer servers for easier collaboration. The main reason for using a Jenkins server for CI/CD is that it provides concurrency and parallelism, which are essential for CI/CD. Concurrency reduces time needed to complete tasks such as pull requests. Jenkins supports extensible plugins and can be integrated with nearly every tool in the CI/CD toolchain. [22]

For DevOps and organisations Jenkins provides the tools to build CI/CD pipelines that enable software developer teams to quality control their code and deliver fixes and updates to their end user through automation. Simplistically, Jenkins realises when the source code has been edited and automatically builds it and runs tests against it. However, running a Jenkins server can be complex and has been categorized as having a steep learning curve.

#### *4.1.2. Ansible*

Ansible is free and open source configuration management tool developed and maintained by Red Hat and written with Python. With Ansible the user can deploy, configure, install software and tools among other things. Ansible uses YAML as its language and depends on python on the deployment computer, but target computers don't require either of them installed, since Ansible manages nodes through SSH. Ansible brings multiple benefits for system management by being declarative, idempotent and agentless. Best practices for Ansible include using inventory, vaults and playbooks when managing systems. [23]

#### *4.1.3. Chef*

Chef is commercial configuration management tool developed and maintained by Progress and written with Ruby and Erlang. As Ruby has a steep learning curve, to use Chef must the developer have a thorough understanding of programming as where as in Ansible is relatively easy to learn due to YAML being human-readable. Chef uses domain specific language for writing "recipes" for system configuration. Chef's servers will run on the client's side and configuring the servers requires more knowledge. Chef is used to automate the supplying of infrastructure. Regarding DevOps, Chef can be used to deploy and run the servers, thus allowing continuous delivery. [24]

The architecture of Chef architecture is built from three building blocks; Chef workstations, Chef servers and Chef nodes. The workstations are the physical locations where the configurations are created. These need to be set up and configured locally in order to start using Chef. The Chef server is the server which has the configuration data, cookbooks and other relevant data stored. The server works as the middle-man between the workstations and nodes. The workstation uses Knife, which is a command-line tool used communicate with the server. It is the responsibility of the server to authenticate communication between the workstations and nodes using public-key cryptography. The Chef nodes are the client servers where the changes are pushed to. Nodes can be for example; virtual servers or containers.

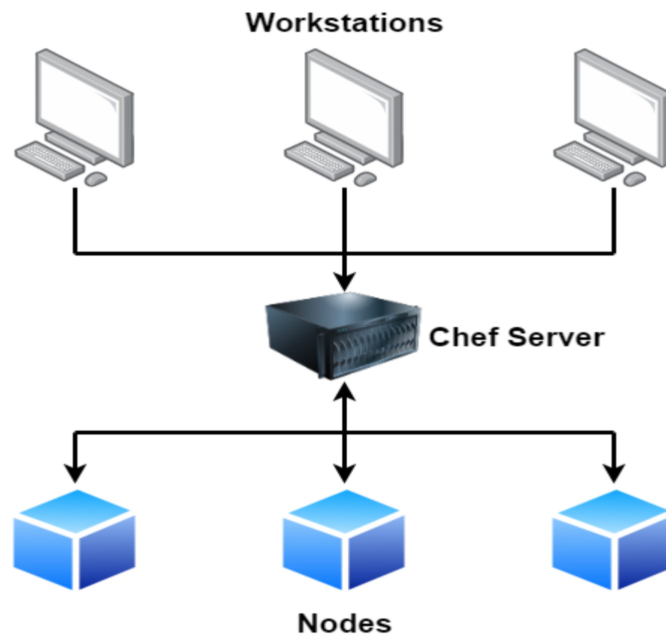


Figure 5. Overview of Chef

### *4.1.4. Puppet*

Puppet is a system management tool designed to be used with Linux and Windows operating systems. It uses a Domain Specific Language and it was created with Ruby. It is mainly used for server management, configuration and deployment of software in conjunction with the organization's infrastructure. Puppet tests the deployed environment to make sure that it is deployed correctly. There are two versions of Puppet, Open Source Puppet and Puppet Enterprise. The first one is the basic version that is freely available from the company's website. The other one is a commercial version that has additional features for more efficient management of nodes. Puppet uses a declarative programming logic which revolves around what to do instead of how to do it. [25]

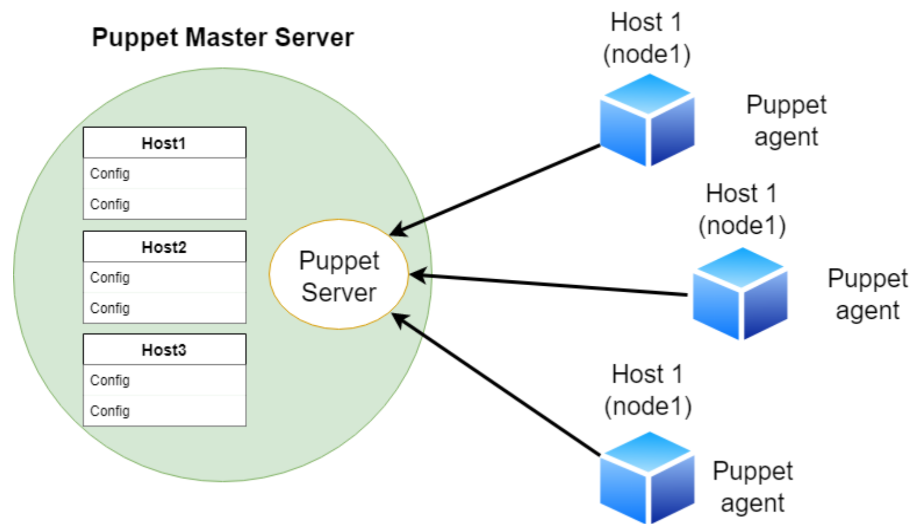


Figure 6. Overview of Puppet

## 4.2. Containers

### *4.2.1. Docker*

Docker is platform as a service product (PaaS) that is used as a containerization platform. It was developed in Go programming language and it uses a client-server architecture where the Docker client communicates with the Docker daemon, called dockerd which listens to the clients API requests. On Windows and Mac OS Docker Desktop is a viable graphical Docker user interface that is beginner friendly and easy to install. On Linux operating system Docker takes advantage of the Linux Kernel's built in isolation features to run containers in singular instances. [26]

### *4.2.2. Containerd*

Containerd is a popular container runtime. Container runtimes are the software which are responsible for running the containers. They create, start, pause and destroy containers. Containerd can also be used pull images from container registries. Containerd was originally developed by Docker but it was later donated to the Cloud Native Computing Foundation (CNCF). Compared to Docker, containerd only has some features of Docker and it is not as easy to learn. Docker also still uses containerd itself to manage and run containers so if Docker is installed, containerd is also installed with it. [27]

### *4.2.3. CRI-O*

CRI-O is also a very popular container runtime. It has the same features as containerd. The key difference is that CRI-O was developed to be a container runtime for Kubernetes. [28]

## 4.3. Container Registries

Container registries are used as part of the DevOps process. They are basically repositories that can be used to store, manage and perform vulnerability analysis for all your containers and container images. They can be used to automatically build and deploy containers. [29]

### *4.3.1. Docker Hub*

Docker Hub is Docker's own container registry and it is the most popular container registry in the world. Docker Hub is a cloud-based registry that also features access to a large catalogue of open source container image repositories. One of the reasons to use Docker Hub is because the open source repositories are reviewed by trusted sources. The reviews include testing out the image repositories to meet quality and security standards. Container registries work as one part in enabling CI/CD. [26]

### *4.3.2. Github Packages*

GitHub is very popular amongst students and because of this, GitHub packages has also become popular as an option for a container registry. The registry does provide public repositories, but anyone who does want to pull one, will need to authenticate with their GitHub's Personal Access Token. [30]

### *4.3.3. Private Registries*

For more private situations, private container registries are a possibly solution. These bring increased security for managing and sharing the containers. Each of the cloud providers have their own products which can be used: Microsoft Azure has Azure Container Registry, Google has Google Cloud container registry, Amazon Web Services has Amazon Elastic Container Registry and Docker Hub also has private registry option. [31, 32, 33]

## 4.4. Container Orchestration

Using containers in production environment can easily become too complicated due to configuration, containers crashing, shutting down etc. Container orchestration tools try to simplify this by introducing configuration files for whole system, auto-scaling, fault-tolerance and automating deployment. The container orchestration tools also help with deploying the application into new environments.

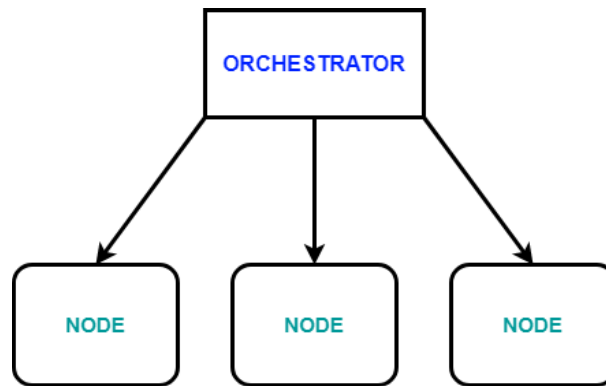

Figure 7. Container Orchestration tool

### *4.4.1. Kubernetes*

Kubernetes is an open source container orchestration tool. Kubernetes can be used to manage, schedule and scale deployments of containers. Kubernetes is fault-tolerant as it has been battle-tested by massive organisations such as Google. It was originally developed by Google. But it is now being maintained and developed by the Cloud Native Computing Foundation. Kubernetes is also one of the most popular container orchestration tools and has a large active community. Kubernetes can be used to create replicas of the application for higher availability and scaling. This information will be typed into the YAML file. Kubernetes has an inbuilt load balancer which distributes traffic and it also has self-healing features meaning that it will restart or create a new container if one of them crashes. [34]

### *4.4.2. K3s and K3d*

K3s is a lightweight distribution of Kubernetes. It was created by Rancher by combining all of Kubernetes features into single binary file. Because it doesn't require much from the host system, it can even be ran in a Raspberry Pi. K3d is same as K3s, but it is ran in a docker container. [35, 36]

### *4.4.3. Docker Swarm*

Docker Swarm is easier to learn than K3s or Kubernetes. It does have the basic container orchestration tool functionalities, but lacks in additional features like auto-scaling. It is more practical for companies that are new to the container world. [26]

## 4.5. Use Cases

### *4.5.1. Netflix*

Some major companies have started to use containerization. For instance, Netflix, a very popular streaming service, changed from a monolith architecture to microservices. This change in architecture allowed Netflix to update or add their platform easier with a lesser risk of the update not working. Netflix decided to break their application into microservices because of their rapidly growing user base, the ease that comes with the possibility of easy rollbacks and the possibility to add new features to the existing application faster. Netflix started moving from the monolith architecture to a cloud-based microservice oriented in 2009. While moving all of their elements to the cloud, Netflix also split the monolithic application into hundreds of smaller services [1] [37]. Netflix wanted to embrace failures so they developed a tool called Chaos Monkey to randomly turn off servers during working hours to learn more on how to maintain the availability, recoverability and fault tolerance of the system. [38]

### *4.5.2. Apex Legends*

Another great example is how Respawn Entertainment and Electronic Arts handled the launch of the free-to-play title Apex Legends. Within the first 72 hours of the game's launch the game had gathered a player base of 10 million people. In video game business the initial days after launch are the most important in keeping players engaged and playing. Ubisoft worked closely with Google Cloud as an component for Multiplay, the game server of Apex Legends. As per Paul Manuel, the Managing Director for Multiplay "After working with Google Cloud on Respawn's Titanfall 2, Google cloud was the logical option for Apex Legends. With its reliable cloud infrastructure and impressive performance during our testing phase, it was clear we made the right choice".[39] With Multiplay and Google Cloud Respawn Entertainment managed to scale their server capability in response to the huge amount of concurrent players ensuring a smooth launch experience. This kind of scaling is not possible without automation. Google cloud can be used to deploy microservices via Kubernetes and its container services.

## 4.6. Requirements

The original requirements in this project were to showcase a way to use Ansible, Kubernetes and containers to modernize Lovelace. This included demonstrating how to use the forementioned technologies to deploy, configure, scale and containerize an application. The requirements were deemed to be the following; be able to deploy a simple application and all it's dependencies with Docker Image and Ansible's playbook features in conjunction with configuring Kubernetes to automatically deploy additional clones of the nodes when needed to. Kubernetes currently only supports Docker, CRI-O and containerd as a container orchestration tool.

### 4.7. Chosen Tools and Technologies

In this project the decision of tools came from the prior experiences of the project members, project requirements and how the tools could be integrated together to solve the task at hand. Another factor that played a role in the decision making of the proper tools was the availability of online documentation, tutorials and research papers. [21]

Docker was chosen as the container platform in this project since it is widely documented and offers more features than containerd. It also has an more user-friendly to new users as the learning curve is not as steep compared to the other tools.

Since Kubernetes fundamentally works together well with Docker containers in the real world scenarios, the only choice to be made was which distribution of it was to be used in the project. Like Docker, Kubernetes also is well documented.

The most difficult decision regarding the Kubernetes distribution was between K3s and K3d. Both of them are good options for working with containers. K3d is better suited for working with smaller environments such as Raspberry Pi, whereas K3s is more production deployment oriented. We opted to use the K3s Kubernetes distribution since it is highly available and light weight on the hardware which suits our testing purposes. K3s is also one of the more easily usable distributions which already had a lot of documentation and research online. Furthermore it supports the usage of a SQLite database, which is the most familiar database engine within our team. Additionally the K3s supports multiple other database engines as well, contradicting the K8s which only supports the etcd. The requirements and support of different databases/OS in Kubernetes distributions can be seen in figure 8. K3s can easily be installed via a WSL Linux distribution on Windows or directly from the terminal on a Linux operating system. As a note from now on in this project K3s will be referred to as Kubernetes for simplicity.

| | K3s | K3d | K8s | MicroK8s | KubeEdge |
|---|---|---|---|---|---|
| RAM/Master | 512MB | 256MB | 2GB | 4GB | - |
| Storage Footprint | 40MB | 150MB | - | 200MB | 266MB |
| Container Runtime | containerd (default), Docker | containerd (default), Docker | containerd (default), Docker (deprecating), CRI-O | conainerd | Docker, containerd, CRI-O, Virlet |
| Default Ingress Controller | Traefik | Traefik | - | Nginx | - |
| Database | SQLite | SQLite | etcd | etcd | etcd |
| OS | Linux distro | Linux distro | Linux distro | Linux, Windows 10 or MacOS(Multipass), LXD (container) | Linux, Windows 10 or MacOS (Multipass) |
| High Availability | 2+ server nodes and 0+ agent nodes | 2+ server nodes and 0+ agent nodes | 3+ control nodes | 3+ nodes | HA CloudCore |
| Suitability for IOT | O | O | X | O | O |

Figure 8. Overview of the system requirements of different Kubernetes distributions

### 4.8. Risk Assessment

| Risk | Likelihood | Impact |
| --- | --- | --- |
| The scale of the project becoming too large for the given time frame | Common | Major |
| Kubernetes configuration complexity | Unlikely | Major |
| Selecting the right tools for the job | Rare | Minor |

As the likelihood of the scope of the project becoming too large was the most common risk, the environment and testing were simplified at first. It is much easier to then move into a more complex environment if the initial goals for the project are met. Furthermore, it makes learning of the tools simpler and makes it easier to focus on tackling the issues that could come as the project moves forward. The goal of the thesis can be met with multiple different tools so there necessarily are no correct tools to be selected, thus keeping the impact of the risk low. As containerization and DevOps are principles that are gaining recognition, the risk of not finding enough scientific research is unlikely.

# 5. IMPLEMENTATION

This chapter of the thesis goes through the implementation process on the tools that were chosen for this project and how they were used together. Implementation plan seen in figure 9 shows the higher level architecture design.

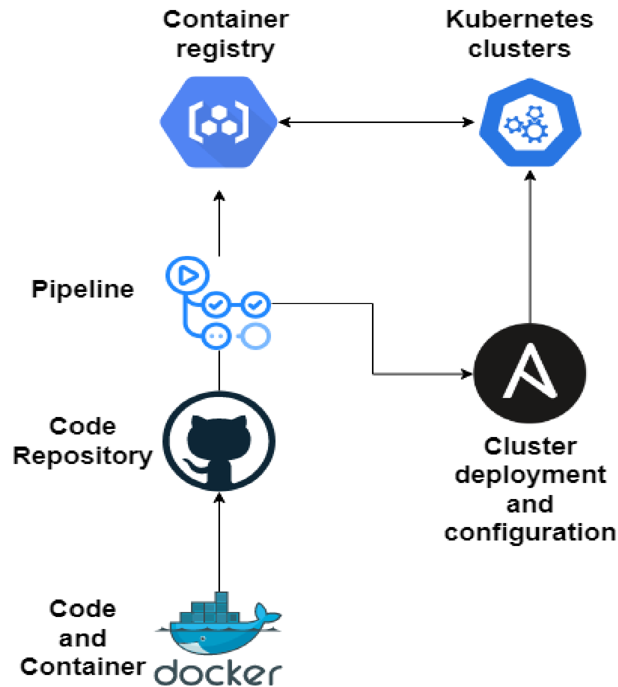


Figure 9. Implementation plan

## 5.1. Code

The implementation of this project started with creating a simple python application that could easily be tested in our testing environment. The application responds to HTTP post and get requests. The code can be found in the appendices of this paper.

After validating the technologies, the team started creating a flask application to resemble the Lovelace system. This meant configuring the application to work with Celery, Redis and RabbitMQ. The point in this was to test and validate that the chosen technologies for containerization and deployment management would work with the other technologies that Lovelace uses. The application used Celery workers to handle user requests, RabbitMQ as the message broker and Redis as the database. The K3d architecture can be seen in fig 10.

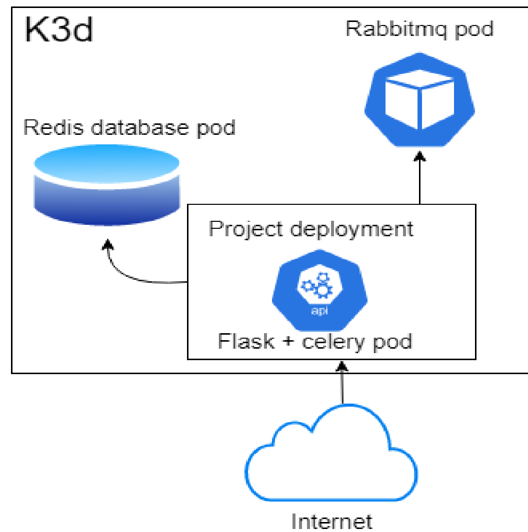


Figure 10. K3d architecture

### 5.2. Docker

A Docker image was created to automatically install all the requirements that the application had.

```
FROM python:3.6-slim

RUN apt-get clean \
    && apt-get -y update

RUN apt-get -y install \
     nginx \
     python3-dev \
     build-essential

WORKDIR /app

COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt --src /usr/local/src

COPY . .

EXPOSE 5000
CMD [ "python", "flaskapi.py" ]
```

### 5.3. Github and Pipelines

The code of the original Lovelace system is hosted in Github, therefore it was decided to use the same environment. The repository contains the simple python code, the

Dockerfile for the container and Github actions which are used to build and deploy the container to Docker Hub when a push happens to the main branch. The Github actions contain checks that test if the new branch is in conflict with the older one. The continuous integration that Github actions supports is an important part of the project.

### 5.4. Docker Hub

A private container registry was created into Docker Hub from which the created docker images could be pulled from. The images that were located in the registry were then automatically pulled by Kubernetes when the chosen image was edited. [26]

### 5.5. Ansible

To create the environment an Ansible playbook was created. The playbook was used to install docker, Kubernetes and other requirements onto the host machine. The playbook was created to ease the effort of the first installation process and apply required configuration for the Kubernetes cluster. The playbook was a yaml file. After running the playbook, the states of the packages that were to be installed were checked to ensure that the execution was successful.

### 5.6. Kubernetes

To begin the testing of Kubernetes the team had setup a Docker Hub repository at maso77/lovelaceregister that contained the Docker image, Ansible playbook and all the other required files. The testing was done by first pulling the Docker Image from the repository, then verifying its version with docker tag and push. After confirming the version it was deployed and scaled manually via Kubernetes commands.

When the scaling was verified the next part was to configure Kubernetes with the Ansible playbook to automatically deploy and scale a defined amounts of nodes from the application. Kubernetes successfully scaled the application to three different worker nodes as seen below. The worker nodes could then each start working on reacting to different tasks/requests in the queue in a First in, First out methodology.

```
$ kubectl get deployments
NAME                 READY   UP-TO-DATE  Available   AGE
simplepython-dep     3/3     3           3           68s
```

### 5.7. Networking

Firstly, a service yaml was created containing information to listen to TCP port 5000. The service forwards traffic to the pods which then forwards it to the nodes. Above the service there is an Ingress resource, which routes all incoming traffic to the service.

Ingress works as the load balancer in this thesis work. The benefits from using Ingress for networking comes when there are more than one service in the Kubernetes cluster. All the traffic can be routed with one Ingress resource to all the services without needing to use multiple load balancers.

## 5.8. Risk Assessment

| Risk | Likelihood | Impact |
|---|---|---|
| The scale of the project becoming too large for the given time frame | Unlikely | Major |
| Kubernetes configuration complexity | Unlikely | Major |
| Selecting the right tools for the job | Rare | Minor |

As the project progressed the risk of the scale of the project becoming too large became less likely to happen, since the initial focus was on keeping it simple and adding to it if there was time. The choice of tools was focused around experience and the available scientific research and online documentation so the other risks were kept identical to the proposal made in the previous part of the project.

# 6. EVALUATION

This chapter of the thesis includes how the project was evaluated. It is divided into sections on which each element of the project is evaluated. The main goal of the evaluation is to examine the results of some of the most likely reasons for failure.

## 6.1. Evaluation Plan

To test the implementation introduced in this thesis, it was decided to be evaluated with K6. K6 is an open source testing tool that allows the user to test their system with different testing methodologies which will be introduced in this section. The plan is to get data on how the Kubernetes clusters react to different kinds of traffic and if it can recover from failures/crashes.

## 6.2. Testing the Cluster for Scalability and Fault Tolerance

### *6.2.1. Input*

The load testing of the Kubernetes cluster is to be done using the K6 testing tool which is specifically designed to create a user-friendly load testing experience. There are two versions of K6, cloud and the open source version. For our project the open source version was chosen, because it is free to use and has plenty of documentation, while the cloud version is still quite new. K6 testing tool enables us to test the environment in different ways such as Smoke Test, Load Test, Soak Test and Stress Test (or Spike Test). With these different testing methodologies the project can be tested in different aspects. [40] These aspects include testing the environment in minimal loads, testing the environments performance in terms of requests per second (RPS), seeing how the environment reacts to extreme conditions and finally seeing the reliability of the system's performance in an extended time period. In this project the application will be evaluated with Smoke-, Load-, Spike- and Stress Tests. All the tests come with thresholds to automatically notify the tester if the HTTP request failure rate is over 1 percent or the HTTP request duration takes over 200 ms over 95 percent of the time. The next subsections will quickly introduce these testing methodologies.

### *6.2.2. Smoke Testing*

Smoke testing is beneficial to verify that the scripts are running correctly without errors and that the system doesn't malfunction under minimal loads. In the case of smoke test failing, it can be concluded that either the system or the script needs to be fixed before continuing on to the other tests. Smoke tests only test the system with 1 to 2 Virtual Users (VUs). [41]

Figure 11. Example of a Smoke Test

### *6.2.3. Load Testing*

Load testing is a controlled testing methodology that assesses how much traffic the tested system can withstand under normal and high traffic conditions. For example, if a website normally has 100 concurrent users using it and in at peak hours 1000 users, then load testing can be used to determine how the system behaves under this load. In this case it will steadily increase VU's until the set peak and then stay at that level of traffic for a while before scaling back down. The standard metric that load tests use is Requests Per Second or the number of concurrent users. [42]

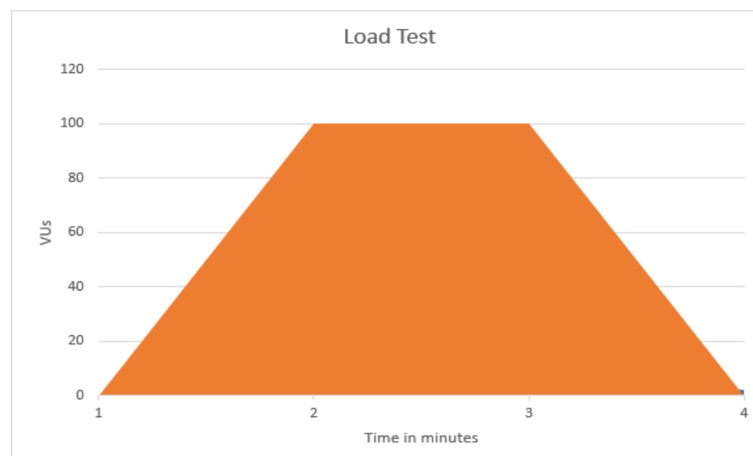


Figure 12. Example of a Load Test

### *6.2.4. Stress and Spike Testing*

Contrary to the nature of load testing, stress and spike tests will measure the stability level of the system when it is under heavy traffic in an unpredictable, chaotic way. It's main purpose is to determine the failure points of the system and how the system recovers from failures. Stress testing can be used for example to test how a web application behaves under extremely high number of concurrent users and HTTP

connections. These results can be observed to assess does the system save its state, can it recover the last stable state and is a failure in the system a basis for compromised security. Stress testing and spike testing will be concluded in the following way: ramp up the number of VUs to 10, then suddenly spike it up to specified amount and then level it back down to 10 before going back to 0 VUs. After each ramp up or level down the number of VUs stays the same for a couple of minutes. [43]

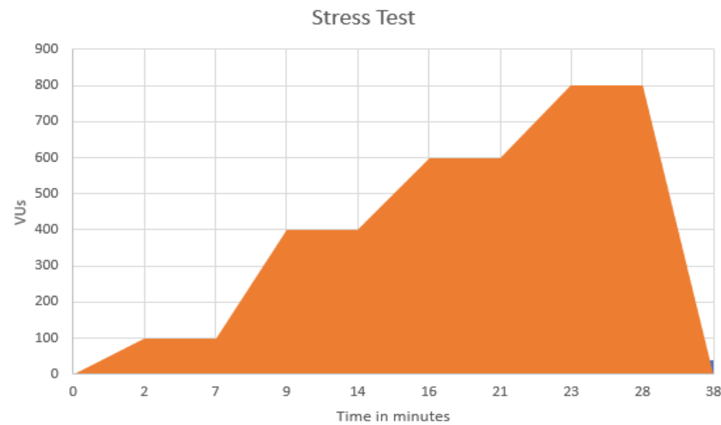


Figure 13. Example of a Stress Test

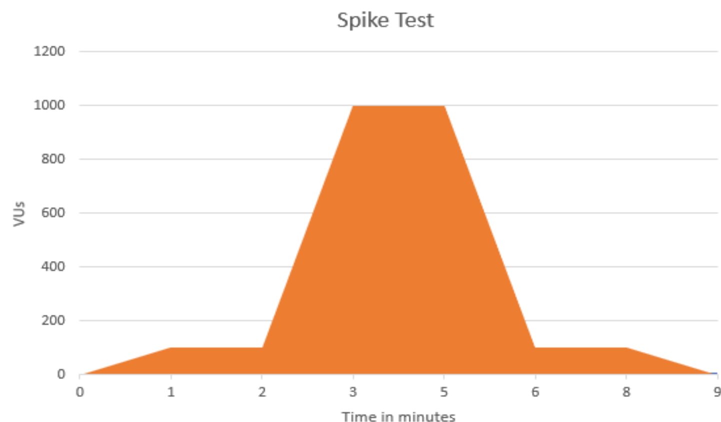


Figure 14. Example of a Spike Test

### *6.2.5. Desired Output*

The expected result of the different tests will be that under low and medium traffic Kubernetes wont react and scale the system. The expectation is that Kubernetes will start scaling the system when experience high-to-extreme size of traffic. Smoke testing will be concluded in the same way as shown in figure 11. It is expected that smoke and load tests will conclude smoothly without any issues since the number of concurrent users is low and the system should in theory easily withstand this. The expectation regarding stress and spike tests is that under a 1000 concurrent users there will be a noticeable spike in the system's resource usage and that when nearing the expected

limits it will start to work noticeably slower. In higher traffic situations Kubernetes is expected to automatically scale the system. In extreme conditions the system is expected to crash and this is done purposefully to test the current limits of the server. During fault tolerance testing the expectation is that Kubernetes correctly recovers from pod failure in short time and starts re-creating the crashed pods.

| | VUS | HTTP Request Fail % | HTTP Request Duration | Test Duration | No of Pods | Crashed [Y/N] |
|---|---|---|---|---|---|---|
| Smoke test | 1 | <1% | <100 ms | 1min | 1 | N |
| | | | | | | |
| Load test | 10 | <1% | <100 ms | 10min | 1 | N |
| Load test | 100 | <1% | <200ms | 10min | 2 | N |
| | | | | | | |
| Stress test | 10 | <1% | <200ms | 10min | 1 | N |
| Stress test | 100 | <1% | <200ms | 10min | 2 | N |
| Stress test | 200 | <1% | <200ms | 10min | 2 | N |
| Stress test | 400 | <1% | <200ms | 10min | 3 | N |
| Stress test | 600 | <1% | <200ms | 10min | 3 | N |
| Stress test | 800 | <1% | <200ms | 10min | 4 | N |
| | | | | | | |
| Spike test | 100 | <1% | <200ms | 1min | 2 | N |
| Spike test | 200 | <1% | <200ms | 1min | 3 | N |
| Spike test | 1000 | <2% | <200ms | 1min | 4 | Y |

### *6.2.6. Results*

The results of the testing process was monitored via the terminal window and a platform called Lens. Lens is a Kubernetes platform that allows easy monitoring of the Kubernetes clusters to show the user what is going on behind the scenes in a graphical interface. The results were saved in a text file after each execution. Fig 15 shows what kind of data comes out of testing.

| | VUS | HTTP Request Fail % | HTTP Request Duration | Test Duration | No of Pods | Crashed [Y/N] |
|---|---|---|---|---|---|---|
| Smoke test | 1 | 0.00% | 39.42ms | 1min | 1 | N |
| | | | | | | |
| Load test | 10 | 0.09% | 40.24ms | 10min | 4 | N |
| Load test | 100 | 0.04% | 40.63ms | 10min | 4 | N |
| | | | | | | |
| Stress test | 10 | 0.07% | 39.17ms | 10min | 4 | N |
| Stress test | 100 | 0.05% | 40.87ms | 10min | 4 | N |
| Stress test | 200 | 0.02% | 44.19ms | 10min | 4 | N |
| Stress test | 400 | 0.01% | 43.85ms | 10min | 4 | N |
| Stress test | 600 | 0.01% | 171.32ms | 10min | 4 | N |
| Stress test | 800 | 0.80% | 328.99ms | 10min | 4 | N |
| | | | | | | |
| Spike test | 100 | 0.92% | 45.75ms | 1min | 4 | N |
| Spike test | 200 | 0.92% | 184.33ms | 1min | 4 | N |
| Spike test | 400 | 0.00% | 1.98s | 1min | 4 | N |

The fault tolerance of the system was observed during and after the tests. As the team had to wait for the number of pods to downscale back to one. When manually deleting the pods, Kubernetes would automatically create new ones to combat the deletion.

```
  data_received..................: 1.1 MB 18 kB/s
  data_sent......................: 595 kB 9.9 kB/s
  http_req_blocked...............: avg=1.1ms    min=932ns    med=2.69µs  max=94.64ms  p(90)=5.16µs   p(95)=14.55µs
  http_req_connecting............: avg=1.09ms   min=0s       med=0s      max=94.56ms  p(90)=0s       p(95)=0s
✗ http_req_duration..............: avg=182.96ms min=33.76ms med=38.1ms  max=709.97ms p(90)=592.52ms p(95)=604.87ms
    { expected_response:true }...: avg=184.33ms min=34.55ms med=38.17ms max=709.97ms p(90)=592.68ms p(95)=604.99ms
✓ http_req_failed................: 0.92%  ✓ 60        ✗ 6410
  http_req_receiving.............: avg=74.89µs  min=12.82µs med=68.4µs  max=413.34µs p(90)=120.91µs p(95)=139.11µs
  http_req_sending...............: avg=19.32µs  min=5.9µs   med=12.36µs max=414.38µs p(90)=35.03µs  p(95)=47.34µs
  http_req_tls_handshaking.......: avg=0s       min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
  http_req_waiting...............: avg=182.87ms min=33.7ms  med=38.01ms max=709.85ms p(90)=592.41ms p(95)=604.79ms
  http_reqs......................: 6470   107.172386/s
  iteration_duration.............: avg=1.36s    min=1.06s   med=1.07s   max=2.32s    p(90)=2.19s    p(95)=2.21s
  iterations.....................: 3235   53.586193/s
  vus............................: 1      min=1       max=200
  vus_max........................: 200    min=200     max=200
```

Figure 15. Data from testing

## 6.3. Data Analysis

### *6.3.1. Expected Outcome Vs Reality*

The expected outcomes of the executed tests were that the pods would scale without errors as the number of virtual users increases. It was expected that during load testing, stress testing and spike testing; the amount of pods would scale up to a maximum of four, the HTTP request failure percentage would stay below 1 percent, and that the HTTP request duration would stay under 200ms. After concluding the tests, the outcomes matched our expectations. The HTTP request failure percentage and duration were below expected, and the number of pods scaled up to the expected levels. The unexpected result from testing was that the number of pods always scaled to four when there was over 1 VU accessing the server. The overall results from the tests were better than what was expected, for example the HTTP request duration and failure rate stayed under their respective thresholds. The system didn't crash even at 800 concurrent VU's. When manually deleting pods to simulate crashing the HTTP request failed percentage rose to 3.3% with 20 VU's.

### *6.3.2. Major Problems, Shortcomings and Flaws*

During the testing phase of this project it was observed that the largest shortcoming of this environment was the actual hardware of the server. The system had 1 Gigabyte of RAM memory which quickly was used up during the tests. This caused the aforementioned problem regarding the pods. In some tests the maximum HTTP request duration did break the threshold, but the average durations were well below 200ms for the load and stress tests. Spike testing the system managed to increase the avg HTTP request duration up to almost 2 seconds as can be observed from the results table.

# 7. DISCUSSION

## 7.1. The Change

The original plan was to introduce microservice architecture, containers and Ansible into the current Lovelace system. As time progressed the goal of the project changed. The team noticed that it wasn't viable time-wise to introduce these new technologies directly into the current system as it would have to be revamped completely and so the goal changed to showcasing how these technologies can be used to automatise the deployment and configuration of an application. This project works as an option for the future proofing of Lovelace in its next version.

## 7.2. Difficulties

The first problem in the project came forward very early on as the installation documentation for the Lovelace system was severely outdated. Some python libraries could not be installed anymore as the versions were so old. The team spent a lot of time trying to install correct versions of the python libraries, before concluding the change of the goal discussed before. After the change the project halted for a duration of time while the team was figuring out the new direction of the project.

While creating the networking aspect for this thesis work, the team came across some problems regarding traffic routing for Kubernetes cluster, correct service and pods.

During the evaluation of the system, the team encountered a problem regarding the server's hardware. The amount of VUs required to actually put pressure on the system had to be extremely large (over 1000), before the CPU usage rose noticeably. This wasn't a very realistic amount of concurrent users when looking at it from a system like Lovelace's perspective as it will not have this many users accessing it at the same time in the near future. When the system was stress tested the bottleneck of it was found to be the original memory allocation, the rest of the system's usage stayed low. For instance, when the Kubernetes was scaling the system, the pods would take up all the RAM of the system in a way the team was cyber-attacking their own system.. This caused the whole server to crash and it would have to be restarted. The server was updated to have a larger amount of memory and the configuration regarding scaling of pods was changed. Even after the hardware update the system would scale its number of pods even when it was handling traffic from 10 VU's.

After the technologies had been validated, the next problem was installing and configuring a flask application with Celery, RabbitMQ and Redis. Installing Redis proved to be a problem, since it didn't get installed and thus the Celery workers couldn't use it as a database in the application. The answer for this problem was that the deployment and configuration of Redis had failed and it was quickly fixed after the discovery.

### 7.3. Future Work

Before actually implementing this project into any system, the system should be configured to support these technologies and this project needs to be further tested. The next step would be to apply these technologies, methods and tools on to the actual Lovelace environment. This project merely works as a proof of concept to give an idea on how the technologies could be integrated into Lovelace. This would start by choosing the tools that can differ from the ones in this project depending on factors such as Lovelace's administrators strong suites. After the choice, Lovelace can be split into microservices and then added into the chosen cloud environment. The scaling of Kubernetes needs to be configured correctly for the demands of the actual environment, in this project this was deemed to be 10% of the system's hardware.

There are three options for the implementation of this project into Lovelace. The options are listed in the order of how much effort/changes the implementation would take from lowest to highest effort. First is Lift&Shift which like the name implies is simply copy pasting Lovelace system into containers with minimal changes. The second one is Refactoring, this approach is very similar to the first one except the source code would be further optimized for containerization. The third one, Rearchitechturing is the most time and effort consuming, in this approach the whole system would be overhauled to better fit the DevOps mentality and containers.

### 7.4. State of the Art

Similar technologies and methodology have been used in the modern development environments. One example was a creation of a Framework that also deployed containers using Kubernetes. The framework used Kube-API as the frontend server. In contrast to this project the Kubernetes clusters were managed with Rancher and Jenkins was used as the automation server for CI/CD and pipelines. Docker images were the main part of the framework that were used to deploy instructions and configurations to execute the application. The evaluation of the project was done by testing multiple scenarios with the main focus being around the number of pods and containers and analyzing the benchmark results during the deployment process. [44]

Another example was an application that was tested in two different virtualization environments, Docker and VM. The purpose of the testing process was to determine which of these environments had the better performance. The research that was done while creating the project was in line with this project. The project proposed that VMs were the worse environment, since they had to contain the whole OS, its dependencies and a hypervisor. The project also used Kubernetes and Docker to create a multi-containerized application. The testing was done by running a Fibonacci sequence in both environments and seeing how they performed against each other. Docker managed to reach an average computation time of 52 seconds and the VM had an average of 96 seconds. In this example the containerized application outperformed its VM equivalent by almost 46%. [45]

# 8. CONCLUSION

The conclusion made from all the observations regarding the system was that technologies such as Ansible and Docker were seen as positive options for the next version of Lovelace. With Ansible and Docker the next version's deployment and configuration could be automated and the code changes could be implemented without worrying about the whole system having to be taken down. Kubernetes could be used to automatically create new code checker pods to combat the amount of students posting their assignments on deadline days. However, Kubernetes is possibly overly complicated solution considering the level of traffic that a system like Lovelace has. Kubernetes would be ideal in a system that needs to scale to cater to a larger number of simultaneous users. One estimation of when a Kubernetes would be beneficial for a system was that the system was expected to handle over a thousand concurrent users.

With this project the team managed to implement the forementioned technologies in a way that a Flask based application could handle thousands of concurrent users with autoscaling. Even if one of the instances of the application crashed another one would instantly be created to handle the traffic. The project can be used as a basis for the next version of Lovelace development environment with minimal changes as well as for implementing Kubernetes into a development environment.

# 9. REFERENCES

[1] Izrailevsky Y., Vlaovic S. & Meshenberg R. (2016), Completing the netflix cloud migration. URL: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration.

[2] Jamshidi P., Pahl C., Mendonça N., Lewis J. & Tilkov S. (2018) Microservices: The journey so far and challenges ahead. IEEE Software 35, pp. 24–30.

[3] Django. URL: https://www.djangoproject.com/.

[4] Rabbitmq. URL: https://www.rabbitmq.com/.

[5] Redis. URL: https://redis.io/.

[6] Celery. URL: https://docs.celeryq.dev/en/stable/getting-started/introduction.html.

[7] Postgres. URL: https://www.postgresql.org/docs/.

[8] Nfs. URL: http://nfs.sourceforge.net/.

[9] Apache. URL: https://www.apache.org/.

[10] Rhel. URL: https://access.redhat.com/documentation/.

[11] Torres Martín C., Acal C., El Homrani M. & Mingorance Estrada C. (2021) Impact on the virtual learning environment due to covid-19. Sustainability 13. URL: https://www.mdpi.com/2071-1050/13/2/582.

[12] Guerrero C. & Angarita A. (2014) Virtual learning enviroment to support object oriented programming learning. Revista Colombiana de Computación , pp. 6–8.

[13] David C. (2007) Web.based learning; pros, cons and controversies. Clinical Medicine 7, pp. 37–42.

[14] Alves P., Miranda L. & Morais C. (2017) The influence of virtual learning environments in students' performance. Universal Journal of Educational Research , pp. 517–527.

[15] Newman S. (2019) From monolith to microservices. O'Reailly Media Inc, 253 p.

[16] Bauer D., Penz B., Mäkiö J. & Assaad M. (2018) Improvement of an existing microservices architecture for an e-learning platform in stem education.

[17] Bowles M. (2017) Automation Skills: Background Report.

[18] (2004), Methods and apparatuses for configuration automation. United States patent 7715790B1.

[19] Shahin M., Ali Babar M. & Zhu L. (2017) Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access pp. 2-3.

[20] Gerard Claps G., Berntsson Svensson R. & Aybüke A. (2015) On the journey to continuous deployment: Technical and social challenges along the way. Information and Software Technology 57, pp. 21–31.

[21] Sharma M., Junior E., Vasilev B., Litmaath M. & Santana R. (2020) The simple framework for deploying containerized grid services. EPJ Web of Conferences 245, pp. 1–7.

[22] Jenkins. URL: https://www.jenkins.io/.

[23] Ansible Documentation. URL: https://docs.ansible.com/.

[24] Configuration Management System Software - Chef Infra | Chef. URL: https://www.chef.io/products/chef-infra.

[25] Webteam P., Powerful infrastructure automation and delivery | Puppet. URL: https://www.puppet.com/.

[26] Docker. URL: https://www.docker.com/.

[27] containerd. URL: https://containerd.io/.

[28] cri-o. URL: https://cri-o.io/.

[29] Mohd Mydin M.N., Ismail B., Rajendar K., Ahmad H. & Khalid M. (2021) An operational view into docker registry with scalability, access control and image assessment.

[30] Introduction to GitHub Packages. URL: http://ghdocs-prod.azurewebsites.net:80/en/packages/learn-github-packages/introduction-to-github-packages.

[31] Azure Container Registry | Microsoft Azure. URL: https://azure.microsoft.com/en-us/services/container-registry/.

[32] Container Registry. URL: https://cloud.google.com/container-registry.

[33] Fully Managed Container Registry – Amazon Elastic Container Registry – Amazon Web Services. URL: https://aws.amazon.com/ecr/.

[34] Kubernetes. URL: https://kubernetes.io/.

[35] K3s: Lightweight Kubernetes. URL: https://k3s.io/.

[36] k3d. URL: https://k3d.io/v5.4.1/.

[37] Evans J. (2016), Mastering chaos - a netflix guide to microservices. URL: https://www.infoq.com/presentations/netflix-chaos-microservices/.

[38] Lenka R., Padhi S. & Nayak K. (2018) Fault injection techniques - a brief review. pp. 832–837.

[39] Rayan (2019) How google cloud helped multiplay power a record-breaking apex legends launch .

[40] Khan R. & Amjad M. (2016) Performance testing (load) of web applications based on test case management. Perspectives in Science 8.

[41] Cannavacciuolo C. & Mariani L. (2022) Smoke testing of cloud systems.

[42] Menascé D. (2002) Load testing of web sites. Internet Computing, IEEE 6, pp. 70– 74.

[43] Briand L., Labiche Y. & Shousha M. (2005) Performance stress testing of real-time systems using genetic algorithms , pp. 1–3.

[44] Abhishek M., Rao D. & Subrahmanyam K. (2022) Framework to deploy containers using kubernetes and ci/cd pipeline. International Journal of Advanced Computer Science and Applications 13.

[45] Alowolodu O. (2021) A multi-containerized application using docker containers and kubernetes clusters. International Journal of Computer Applications 183, pp. 55–60.

# 10. APPENDICES

## 10.1. Commands

Creating a SSH key:

- chmod 400 actionsvmkey.pem
- ssh -i actionsvmkey.pem stpuser@12.345.67.890

K3s and kubectl:

- kubectl get pods
- kubectl get service
- k3s cluster list
- kubectl port-forward app 3003:3000
- kubectl apply -f test.yaml
- kubectl get ing

K3d:

- k3d cluster delete
- k3d cluster create –port 8082:30080@agent:0 -p 8081:80@loadbalancer –agents 2

Ansible commands:

- ansible all –ask-pass -m ping
- ansible-playbook –ask-pass playbook.yaml
- docker ps
- docker images
- docker ps -a
- docker system prune
- docker image rm ed
- docker pull maso77/lovelaceregister

Scaling with kubectl

- sudo curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
- sudo kubectl create deployment hellolove –image=maso77/lovelaceregister:latest
- kubectl scale deployment mockdeployment –replicas=4
- kubectl autoscale rs flaskapi-dep –min=1 –max=4 –cpu-percent=10
- kubectl describe deployment mockdeployment
- kubectl config view –minify –raw
- kubectl apply -f bb.yaml
- sudo systemctl start docker
- sudo systemctl enable docker
- kubectl apply -f manifests/

- kubectl apply -f filename
- kubectl describe pod
- kubectl logs -f resourcename
- kubectl get rs/ing/pod/service
- kubectl delete rs/ing/pod/service appname

Testing:

- docker run –rm -i grafana/k6 run - <spiketest.js –insecure-skip-tls-verify (replace spiketest.js with the test you want to run)

## 10.2. Contributions

| | Stage 1 | |
|---|---|---|
| Student | Hours | Contributions |
| Miiro Kuosmanen | 49 | Researching Docker, Containers, Ansible, Kubernetes, writing, communicating with supervisor |
| Lauri Suutari | 51.5 | "Researching Docker, Containers, Ansible, Kubernetes, writing, communicating with supervisor" |
| Joona Holappa | 42 | Researching Docker, Containers, Ansible, Kubernetes, writing, communicating with supervisor |
| | Stage 2 | |
| Miiro Kuosmanen | 52 | Research and installation of the environment. Implementing Docker, Ansible and Kubernetes. Managing the system and communicating with supervisor. Implementing the configurations, writing |
| Lauri Suutari | 47 | Research and installation of the environment. Implementing Docker, Ansible and Kubernetes. Implementing the configurations. writing |
| Joona Holappa | 52 | Research and installation of the environment. Implementing Docker, Ansible and Kubernetes. Implementing the configurations, writing |

| | Stage 3 | |
|---|---|---|
| Miiro Kuosmanen | 42 | Configuring and networking the system for autoscaling. Running tests and evaluating results. Researching articles for citations and rewriting the text. Researching and implementing K3d with django, celery, redis, rabbitmq. |
| Lauri Suutari | 46 | Configuring the system for autoscaling and implementing the evaluation tests. Running tests and evaluating results. Researching articles for citations and rewriting the text. Researching and implementing K3d with django, celery, redis, rabbitmq. |
| Joona Holappa | 45 | Configuring and networking the system for autoscaling. Managing the Kubernetes platform. Running tests and evaluating results. Researching articles for citations and rewriting the text. Researching and implementing K3d with flask, celery, redis, rabbitmq. |
| | Stage 4 | |
| Miiro Kuosmanen | 23 | Communicating with the supervisor, working on the final presentation, implementing the last version of the prototype for Flaskapi with redis, rabbitmq and celery. |
| Lauri Suutari | 35 | Working on the final presentation, implementing the last version of the prototype for Flaskapi with redis, rabbitmq and celery. Finishing the writing part of the thesis. Communicating with the supervisor. |
| Joona Holappa | 31 | Working on the final presentation, implementing the last version of the prototype for Flaskapi with redis, rabbitmq and celery. Finishing the writing part of the thesis. |
| | Total | |
| Miiro Kuosmanen | 166.5 | |
| Lauri Suutari | 179.5 | |
| Joona Holappa | 173 | |

### 10.3. Flaskapi

```
from flask import Flask, redirect, url_for, request
app = Flask(__name__)


@app.route('/',methods = ['POST', 'GET'])
def login():
   if request.method == 'POST':
      return "Post request"
   if request.method == 'GET':
      return "Get request"


if __name__ == '__main__':
   app.run(host="0.0.0.0", port=5000)
```