Containerization of Web Application Back-Ends

Master of Science in Technology
Thesis
University of Turku
Department of Computing
Software Engineering
2024
Marko Järvinen

UNIVERSITY OF TURKU
Department of Computing

MARKO JÄRVINEN: Containerization of Web Application Back-Ends

Master of Science in Technology Thesis, 82 p.
Software Engineering
May 2024

Traditionally, web applications and services have been deployed on-premise on physical servers or virtual machines by installing and configuring the necessary software and dependencies manually. This approach can be cumbersome and inefficient, especially when it comes to managing dependencies and scaling applications. As an alternative to traditional deployment methods, utilizing container technologies has emerged as a popular solution for packaging and deploying applications in a consistent and efficient manner. Containers encapsulate an application and its dependencies together with its configuration into lightweight self-contained units that can be independently deployed and managed with relative ease. By running applications in containers, the underlying infrastructure is abstracted from the developer, allowing them to focus on writing code and building applications rather than managing the infrastructure.

This thesis explores and investigates containerization and container technologies as a solution for operational bottlenecks faced by web application back-ends. The thesis begins by providing a background and overview of container technologies, focusing on Docker containers. Then, a literature review is conducted to identify the benefits and drawbacks of containerization in the context of software development and deployment.

After this, a case study is presented on a web application called eShare, which is an information management tool for engineering projects in marine and process industries developed by Cadmatic. The case study involves identifying operational bottlenecks in eShare and proposing a partial containerization solution to address these bottlenecks based on the results of the literature review. Lastly, the feasibility and effectiveness of the proposed containerization solution are evaluated by comparing the containerized solution with the unmodified eShare application.

Keywords: docker, containers, containerization, microservices, web applications

Contents

List of Figures

List of Tables

1 Introduction

In recent years, the rise in popularity of web-based applications has created a demand and need for scalable and flexible ways to deploy said applications. Traditionally, web applications are deployed by installing a web server, its dependencies, and other required components, such as a database, on-premise, on either a physical server machine or a virtual one. However, this approach can be somewhat time-consuming, difficult to scale, and requires a fair amount of manual configuration on behalf of a human administrator. Manual configuration also means that the whole deployment process is prone to errors. A faulty configuration could leave the application vulnerable to breaches in security or otherwise lead to erroneous behavior that could be costly and harmful to the users or the maintainer organization of the application.

Containerization is a popular solution that addresses these issues by offering a flexible, lightweight, and scalable alternative way of deploying web-based applications. Containerization is the process of bundling the software and its dependencies into portable and easily manageable self-contained units called containers. Utilizing them allows for a more user-friendly way of deploying the software by enabling it to be run in practically any environment without the complexity of resolving the proper dependencies by hand. Due to the environment-agnostic nature of containers, their use also makes cloud-based deployments easier and more manageable.

In this thesis, I will explore the idea and concept behind container technologies

and their benefits for web applications. To guide this research, this thesis will attempt to discover answers to the following research questions (*RQ*):

RQ1: What benefits can be gained by utilizing container technologies with an application like eShare? How about the drawbacks?

RQ2: Which functionalities in an application like eShare are most suitable for containerization?

RQ3: What would be a sensible containerization solution for an application like eShare?

In the second chapter, I will introduce the company and its web-based information management software product *eShare*, which is the target of the case study in this thesis. After this, in the third chapter, I will discuss the background and key principles of container technologies and how they differ from other virtualization approaches, such as virtual machines. In order for me to familiarize myself with using containers, the third chapter will also include a practical overview and guide on creating, running, and managing containers using Docker, a popular containerization platform. In the fourth chapter, I will attempt to provide an answer to RQ1 by conducting a literature analysis on the previous research done on the subject and extracting information about the benefits and drawbacks of containerization that other researchers and practitioners have reported. In the fifth chapter, I will take a practical look, in the form of a case study, at how containerization could alleviate some of the operational bottlenecks faced by eShare based on the results of the literature analysis. The fifth chapter will also attempt to answer RQ2 by analyzing which functionalities in eShare are most suitable for containerization and RQ3 by proposing a containerization solution for eShare. In the sixth chapter, I will discuss, analyze, and evaluate the results of the work done on the case study. Lastly, the seventh chapter will conclude the thesis by summarizing what has been

done, discovered and learned during the thesis, and what could be done in the future regarding eShare and containerization.

2 Cadmatic eShare

EShare is a web-based information management software application developed by *Cadmatic*, a company that specializes in developing software applications for computer-aided design (*CAD*) and engineering. EShare is used to manage and present information related to design projects via a 3D model. In industry, eShare's main user base consists of engineers working in small to large-scale design companies, shipyards, or oil, gas, and chemical plants. In addition to engineers, the owners and operators of these properties belong to eShare's target user group.

2.1 EShare as a tool for information management

To avoid confusion, it is important to make a distinction between eShare and other typical CAD applications, such as the ones offered by Cadmatic. Unlike other typical CAD applications, eShare is not used to conduct the design of a project itself. Rather, it is a complementary tool that is best suited to be used alongside a design application capable of producing 3D models on which eShare then operates by managing the various kinds of information related to it.

The amount of data and information associated with even small-scale design projects can usually be very high. This information is often scattered around on different systems and data sources in various formats. Some data might be located in a spreadsheet or a traditional text document while others might only be accessed from a database using a query language such as SQL. The process of manually

traversing these data sources and systems in an attempt to find relevant data related to some point of interest in a design project can be tedious, time-consuming, and prone to human error. EShare attempts to address this issue by reading the data from these external sources enabling the user to access it from a central place and linking it to the relevant points of interest, such as objects in a 3D model of the project.

2.2 EShare as a digital twin platform

Digital twins can be defined as digitalized representations of systems, objects, or items that are physical, concrete, and sensor-enabled. These virtual replicas can then be analyzed and surveyed via the sensor data, in order to better understand the physical counterpart. The concept of digital twins is usually applied in manufacturing, construction, and urban planning, for example. In these fields, the near-real-time linkage between the digital model and the physical object or system can provide essential insights into the object or system being modeled, enabling more educated decision-making. [1]

EShare is essentially a platform for digital twins where one can access the related data and track the status of a design project from an easily accessible central hub. The right panel seen in Figure 2.1 demonstrates what kind of information is displayed for an object in the 3D model. The attributes displayed in the panel originate from various sources. For example, the topmost category *Model* displays attributes from the 3D model itself, the category *Pressure history (Excel)* displays historical time series data defined in an external Microsoft Excel spreadsheet and the *SQL SERVER* category displays attributes located in a SQL database. These attribute data sources are user-defined, meaning that usually, an administrator configures how the data is fetched from an external source and how it is handled in eShare.

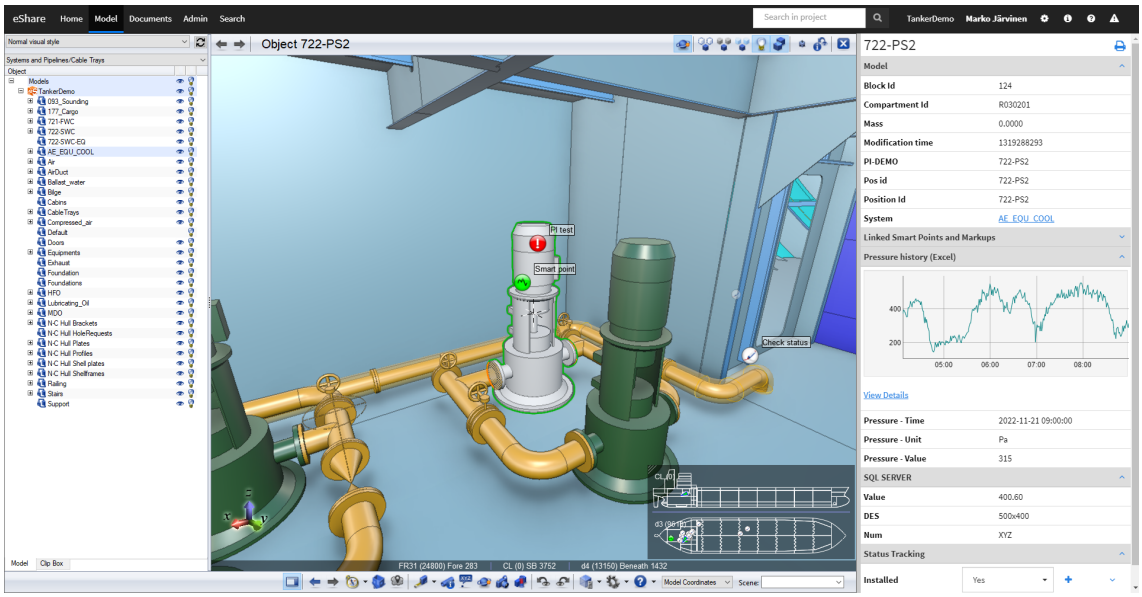

Figure 2.1: View of an object and its associated attributes in a 3D model.

2.3 Use cases

2.3.1 Document-model linkage

Design projects often contain various parts, systems, devices, and gadgets that have technical documents attached to them. Often, these documents are independent of the 3D model and might even be created by an entirely different entity than the designers and engineers of the model, e.g. by a subcontractor. Figuring out which documents are related to which parts of the model can be a challenging and error-prone task to accomplish manually. However, as long as the documents are exposed in a way that eShare may access them, eShare can be used to link the documents to the relevant objects in the model automatically. Common sources for these documents include Cadmatic design applications, file systems, and other external systems accessible via a Restful API. [2]

Once properly configured, eShare may also create a bidirectional linkage between

the documents and objects in the model. This way, the user may both browse a document for related parts, systems, and devices in the model and vice versa, browse the objects in a model and view the related technical documents. [3]

2.3.2 Point cloud visualization

When existing structures are added or removed, it is useful to know how the new arrangement fits the one in the digital 3D model counterpart. This can be achieved with laser-scanned point clouds that are uploaded and configured to eShare. With point clouds, the designer or engineer can examine how the laser scan of a physical space or object corresponds to the 3D model. This way, possible collisions and misalignments between objects of interest can be visualized and recognized. [4]

2.3.3 Map navigation and visualization

In many cases, design projects span large distances both vertically and horizontally. A cruise ship might have a dozen of decks and a process plant might cover a sizeable geographic area. In both cases, navigating and properly visualizing models of such projects can be challenging and disorienting. To help with this, eShare provides functionality for configuring two-dimensional maps that help visualize and navigate the model of the project. For example, floor plans of each deck might be used as maps in the case of a cruise ship and an actual geographical map could be used for a process plant. [5]

Maps help the user visualize the model by providing a two-dimensional representation where the positions of objects and points of interest in relation to each other can be better seen and understood. In addition to this, maps can be used to navigate the model by providing a way to quickly jump to a specific location in the model. This is especially useful in large-scale projects where the user might otherwise have to spend a lot of time traversing the model to find the relevant location.

2.3.4 Data search and export

Design projects are often large and complex, containing thousands of objects and documents. Finding the relevant information from such a large amount of data can be a challenging and laborious task. To inspect and find the relevant information, eShare provides a search functionality that can be used to search for objects, documents, and other points of interest in the project. The user can query and search for relevant information by defining search criteria based on the attributes of the objects, documents, and other points of interest. [6]

Another use case for eShare's search is to export the search results into a file, such as a spreadsheet or a document. This way the information can be easily shared with various stakeholders of the project that do not otherwise have access to eShare, such as subcontractors for example. [6]

2.4 EShare's deployment model

EShare uses a client-server architecture in which the client applications are usually web browsers and native desktop applications. The server on the other hand consists of relational databases and a web server (see Figure 2.2) and as such, it communicates with the client using HTTP. At the time of writing, the most common way to take eShare into use is by installing the server on-premise.

2.4.1 Current limitations

On-premise installation requires an IT infrastructure and resources for maintenance and configuration, which can be inconvenient for some. Better support for cloud environments and the possibility to take the *Software as a Service* (SaaS) model into use is something that Cadmatic aims to move towards with eShare. One objective of this thesis is to explore and research the idea of containerizing parts of eShare so

that the possible transition to a SaaS model could be better facilitated in the future.

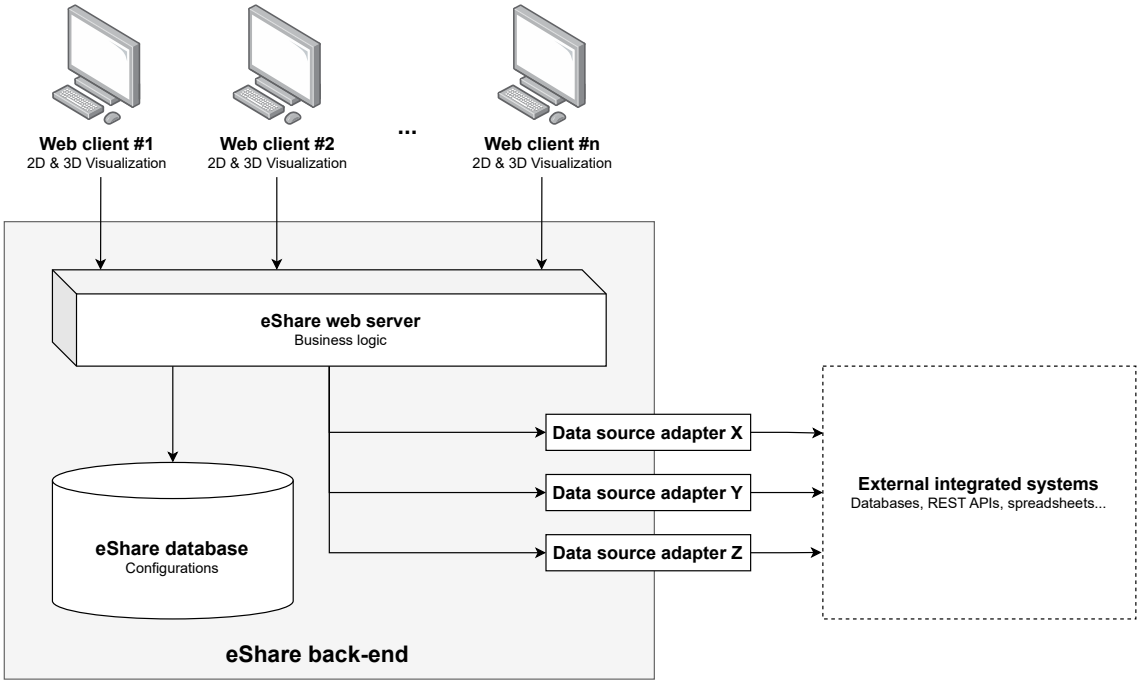

Figure 2.2: A high-level overview of eShare's architecture.

2.4.2 Software as a service

Software as a service (*SaaS*) is a software distribution model in which a software service provider hosts the software in a central location, usually in a cloud environment and makes it available to customers over the internet via a thin client, such as a web browser. SaaS is typically licensed on a monthly or yearly subscription basis.

Compared to traditional on-premise software deployment models, there are no customer-specific production environments, rather, the same central production environment, serves multiple customers at once. This means that the service provider is responsible for maintaining the software, its infrastructure, customer data, and security. [7]

Benefits of SaaS

- **Accessibility:** SaaS applications are accessible from any location with an internet connection. This is especially useful for companies that operate in multiple different locations.
- **Maintenance:** SaaS applications are centrally hosted and maintained by the service provider, which means that the users do not have to worry about maintaining the IT infrastructure themselves.
- **Predictable costs:** SaaS applications are usually licensed on a subscription basis, which means that the users do not have to pay a large upfront cost for the software. This also makes it easier to scale the cost of the software up or down depending on the current demand and desired features.

Challenges of SaaS

- **Loss of control:** The other side of the coin of not having to maintain the IT infrastructure is that the users lose control over it. This means that the users have to trust the service provider to maintain the infrastructure, security, and privacy of the software.
- **Internet connection:** SaaS applications heavily rely on internet connectivity. If the internet connection is slow, unstable, or experiences downtime, it can impact user access and productivity.

2.4.3 Migrating towards SaaS

One of the long-term goals for Cadmatic is to move towards a hybrid SaaS/on-premise model with eShare due to the benefits SaaS provides. The on-premise model will most probably be still supported in the future, as a portion of eShare's current

customers prefer the on-premise model due to the high degree of customization and control it provides.

In order to facilitate the migration from a traditional on-premise deployment model to a SaaS model, containerization of the software can be used to help with the transition. Due to software applications being usually large and complex, containerization is often used to make them portable by encapsulating the software, its dependencies, and the runtime environment into consistent manageable units. This way, the software can be run reliably and consistently on different environments, such as on a developer's laptop or in a cloud environment.

As a step towards the SaaS model, the idea of containerizing parts of eShare is explored in the upcoming chapters. More specifically, the focus is on containerizing some of the backend services of eShare which are currently deployed as standalone executables along the eShare server. Furthermore, it is also explored how containerization of these services could perhaps help with scaling their workload, as they sometimes become quite heavy and long-running.

3 Container technologies

3.1 Background

The concept of containers and process isolation is relatively old and has been around for several decades, with early examples such as the *chroot* system call in the Version 7 Unix in 1979. The chroot operation enabled the changing of the apparent root directory of processes and their children [8]. Consequently, processes with modified root directories were incapable of accessing any resources beyond their designated root, thereby creating an isolated environment to a degree. However, the current form of containers has only emerged in the past two decades. The widespread adoption of containerization can be attributed to the launch of *Docker* in 2013, which has since become the de facto standard for container deployment and management.

The primary driver pushing the development of container technologies has been the need for a more efficient and agile method of deploying and managing applications in large-scale environments. To this end, various containerization techniques and technologies have been developed, each building upon previous innovations. For example, in 2000, FreeBSD introduced *Jails*, a tool that extended the traditional chroot concept of process isolation of changing the root directory of a set of processes in order to limit their access to files and resources outside of their root directory. Jails further extended this by also restricting the access to system-users and the networking subsystem that was previously shared between the processes of the host

system [9].

Linux also saw the development of a similar concept in 2001 with *Linux VServer*, which separated the user-space into distinct *Virtual Private Servers* that each functioned as individual servers from the perspective of the processes within. [10] In 2006, Google launched *cgroups* (control groups), which enabled the prioritization, limiting, and accounting of resources used by a group of processes, ensuring that they did not exceed the assigned amount of system resources such as CPU and RAM usage. Cgroups were merged into the Linux kernel in 2008. [10] [11]

The year 2008 also saw the release of the *Linux Containers* project (LXC) which relied on the functionality of cgroups and Linux namespaces. LXC provided environments that closely resembled that of standard stand-alone installations of Linux without the need for a separate kernel. [10] [12]

In 2013, *Docker* was released and it quickly gained popularity due to its ease of use and effectiveness. The current popularity of container technologies can be largely attributed to the success of Docker. Docker built upon all the previous incremental improvements that had been introduced by the older techniques and made it more accessible to developers. For example, at its release and initial stages, Docker utilized LXC as its default execution environment. However, after the release of version 0.9 one year later, Docker replaced it with *libcontainer*, an in-house implementation of an execution environment. [13] Due to its popularity and having the status of the de facto container technology, this thesis will, from this point onwards, only focus on using Docker as the container technology of choice.

3.2 Differences versus virtual machines

Containers and virtual machines are both similar approaches to virtualization. They create an abstraction layer on top of computer hardware and infrastructure which allows singular system resources, such as CPU, memory, and networking to be rep-

resented and divided into multiple virtual resources. The main difference between them is the way they achieve this. Virtual machines virtualize the entire machine, starting from the hardware layers with every virtual machine running a unique guest operating system. Containers, on the other hand, virtualize only the layers above the operating system with every container sharing the operating system kernel of the host machine. This distinction is visualized in Figure 3.1.

The form of virtualization containers use is called *OS-level virtualization*. In OS-level virtualization, each container shares the host operating system kernel and often also other resources, such as libraries and binaries, but they operate on their own separate file system and process space views.

On the other hand, virtual machines use hardware-level virtualization to create isolated environments that can run multiple operating systems on a single physical machine. Each virtual machine has its own guest operating system and virtual hardware. [14]

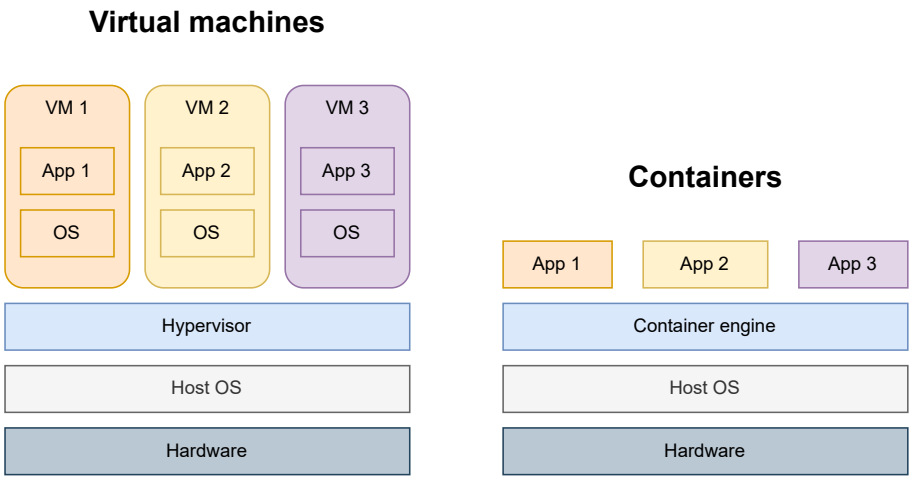

Figure 3.1: The underlying layers of typical virtual machines versus containers.

3.3 Motivations

In recent years, container technologies have gained widespread popularity as a solution to numerous challenges in software development and deployment. The primary motivation behind the widespread adoption of containers is the isolation, modularity, and reproducibility they offer.

Containers encapsulate all the dependencies and configurations necessary to run an application into self-contained units, ensuring consistent and predictable behavior across different environments. This eliminates the concern of the underlying infrastructure from the developer's perspective and enables them to maneuver more easily between the different stages of the development pipeline. For example, moving the application from a developer's laptop to a test environment, and from a test environment into a production environment is smoother and results in consistent behavior. As a result, organizations can adopt more flexible approaches in terms of application deployment and service delivery, as containers can be run on a mixture of environments, such as physical and virtual machines, on cloud services, on-premise servers, or on a developer's laptop, provided the host system has an appropriate container runtime installed and configured.

The loosely isolated environments of containers also offer enhanced security. As containers run in their separate confined environments, the risk of attacks targeting a specific container spreading across the system is reduced. Additionally, containers are minimal in nature, as they only contain the necessary components required to run an application, which results in a minimized attack surface.

Scalability is another significant motivation for the use of containers. Due to their portability and modularity, containers can be quickly and easily added and removed to accommodate changes in demand, enabling applications to scale up and down as required. For reference, Docker containers can be launched in a matter of subseconds [15]. Together with container orchestration techniques, organizations

can optimize their use of containers, only utilizing as many as necessary to provide an optimal user and client experience at a given time.

A further beneficial aspect of containers is their lightweight nature compared to traditional virtualization techniques such as virtual machines. Containers have a considerably smaller footprint and a lower overhead involved in using them. The main reason for this is that individual containers do not require separate personal operating systems like virtual machines. Instead, containers share the operating system of the host system and only include the necessary components to run the application. As seen in Figure 3.1, a typical virtual machine is essentially a copy of an operating system running on top of a hypervisor which in turn runs on top of physical hardware. This means that the application is running on top of all these layers inside a virtual machine which poses some challenges to performance and efficiency. Whereas containers, run directly on top of a container engine and the host operating system.

3.4 Software engineering use cases

For applications that benefit from high levels of portability, such as web applications, containerization is ideal as it enables the application to be easily packaged and deployed via container images. Containers bundle the application and its dependencies into independent units that can be easily and reliably deployed across different environments, such as development, testing, and production environments.

The implementation of *microservices architecture* is another area where containerization is beneficial. Microservices architecture involves breaking down the different concerns and components of a complex software application into smaller, independently deployable components, which can be easily managed using containers. The modular nature of microservices architecture is well-suited for containerization as it enables each component to be deployed and scaled independently, without

affecting the rest of the application.

Containers are also a beneficial tool in managing *Continuous Integration/Continuous Deployment* (CI/CD) processes. A traditional stumbling block in deploying software applications is the differences between the development and production environments. Developers build and ship applications that run on their local machines, only to find out that the application does not work as expected in the testing or production environment [15]. This is often due to the differences in the underlying infrastructure and dependencies between the environments. Automated CI/CD pipelines that utilize containers can help mitigate this issue by ensuring that the application is packaged and deployed consistently across different environments. This enables developers to focus on the application logic and functionality, rather than the underlying infrastructure.

While containerization is beneficial for many applications, it may not be the best choice for all use cases. Applications that require low-level system access, such as kernel-level components, or those that require hardware-level access, such as device drivers, may not be well-suited for containerization. This is because containers are designed to be hardware-agnostic, which means that they do not have direct access to hardware resources. Therefore, applications that require direct access to hardware may not be suitable for containerization.

3.5 Docker

3.5.1 Background

Docker is a software platform that implements the concept of software containers. It was first introduced at the PyCon (*Python Conference*) in 2013 by Solomon Hykes, a software engineer at dotCloud, a Platform-as-a-Service (*PaaS*) company that provided an environment for building and deploying web applications. Initially,

Docker was created as an internal tool for dotCloud to help manage and deploy applications more efficiently. [16] However, the project quickly gained popularity within the developer community, and it was eventually released as an open-source project.

As discussed in Section 3.1, it was not until the inception of Docker in 2013 that container technologies became highly prevalent in the software industry. Docker builds upon the earlier containerization techniques, primarily by providing a developer-friendly interface for creating, managing, and sharing containers. This ease of use and developer-friendliness are some of the primary reasons for Docker's popularity.

3.5.2 Architecture

Docker is a client-server-based application. The Docker client communicates with the server (*Docker daemon*) via a REST API, over UNIX sockets or a network interface. Both the client and server (*daemon*) components can be run on the same system, or they can be connected through a remote connection. The *Docker CLI* and *Docker Compose* are client applications that the users of Docker interact with in order to give commands and instructions to the Docker daemon. Docker daemon, in turn, acts as the server that manages so-called *Docker objects*, such as Docker images, containers, networks, and volumes. The general structure of Docker is illustrated in the Docker documentation (see Figure 3.2). [17]

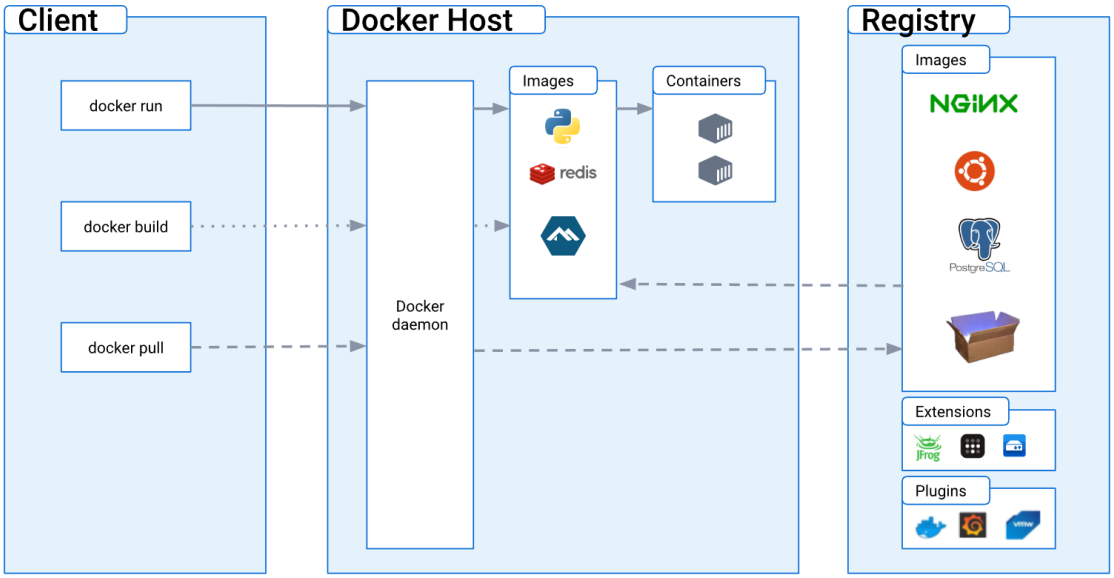

Figure 3.2: The architecture of Docker.

3.5.3 Docker objects

Docker objects are the various components that make up a Docker environment. They are created and managed by Docker Engine, the core component of Docker that allows you to build, run, and manage containers. Docker objects encapsulate different aspects of the Docker ecosystem and provide a way to interact with and manipulate them.

Docker image

Docker images are a core part of running Docker containers. They are immutable snapshots of the instructions for creating containers and are used whenever a container is to be spawned. Images are highly portable and they can be shared between individuals who might want to run containers based on them or build on top of them. Often it is the case that images are not built from scratch, but rather, they

are built on top of existing images with additional custom configuration. For example, one might build an image based on the publicly available image of the Ubuntu operating system and additionally install a web server on top of it along with the needed configuration details.

Docker image configuration (Dockerfile)

A Dockerfile is not, strictly speaking, a Docker object, but it is an essential part of creating a Docker image as it serves as a script for defining the contents and configuration of an image. It is a plain text file that contains a series of instructions that specify how to setup the environment of the container. An instruction could be, for example, a definition of an environment variable or a bash script to execute. The syntax used in Dockerfiles to define instructions is quite straightforward and easy to read, and it supports various configuration options that enable the fine-tuning of the image's contents and runtime environment. Listing 3.2 on page 25 contains an example of a simple Dockerfile which can be used to build an image of a *Node.js* application.

Each instruction in a Dockerfile represents a layer in the image, and the resulting image is a stack of all the layers defined in the Dockerfile. Whenever a Dockerfile is modified and used to build an image, only the layers that have changed are rebuilt. According to the Docker documentation, this is one of the reasons that makes Docker images so lightweight and fast compared to other virtualization technologies [17].

Some of the most commonly used Dockerfile instructions include:

- `FROM`: Specifies the base image that the new image will be based on. Every Dockerfile must start with a `FROM` instruction. To create an image from scratch without a base image, one can use Docker's reserved explicitly empty image *scratch*.

- `RUN`: Used to execute a command inside the container. Typically used for

installing software packages or configuring the environment.

- `COPY`: Copies files from the host machine to the container. For example, configuration files, scripts, or source code.
- `EXPOSE`: Exposes a port from the container to the host machine.
- `CMD`: Specifies the default command to be run when the container is started. Only one `CMD` instruction is allowed for a Dockerfile.

Docker container

Docker containers are runtime instances of Docker images providing isolated environments to run applications and their dependencies that are defined by the image. Containers are created from images, which act as immutable templates that include all the necessary components and configurations to run an application.

Containers do not persist any data about their state by default, rather they can be configured to store data in volumes, which are separate from the container and can be shared between other containers as well. This ensures that data persists even when a container is destroyed and recreated. By default, containers also provide a high degree of isolation from the host machine and other containers, which can be adjusted via additional configuration.

Docker registries

Docker registries are not vital in terms of operating Docker containers, however, they are essential when building Docker images, as they act as storage and distribution mechanisms for Docker images, allowing users to share and access container images.

As mentioned earlier, to reduce the development workload, images are typically built on top of already existing images. These images are pulled from either public

or private registries. For example, *Docker Hub* is a public registry that Docker uses to look up images by default.

Public registries are used to share and download images publicly whereas private registries are mainly used within organizations to securely store and distribute images within their infrastructure. [17]

Docker volume

Docker volumes are a feature of Docker that provides a way to persist and share data between Docker containers and the host system. They allow you to decouple the data from the container itself, enabling data to survive container restarts, upgrades, and even container removals.

When a container is created, it is isolated from the host system and any other containers. By default, Docker provides a file system within the container, but this file system is ephemeral, meaning that any changes made within the container are lost when the container is stopped or removed.

Docker volumes provide a mechanism to create a persistent storage area that can be shared by one or more containers. Instead of storing data within the container's file system, you can mount a volume to a specific directory within the container. This allows you to read from and write to the volume, and any changes made to the volume are preserved even if the container is stopped or removed. The same volume can be attached to multiple different containers which enables data pipelines and shared storage between containers. [18]

Docker network

Docker networks are an integral element of Docker's networking capabilities. They provide a way to connect Docker containers together and enable communication between them, both within a single Docker host and across multiple Docker hosts.

Upon installation, Docker automatically generates three distinct networks: the *bridge*, *host*, and *none* networks. The currently active Docker networks can be inspected with the `docker network ls` command:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
1dd11129834b   bridge    bridge    local
9a94cb0c01ef   host      host      local
65e7bc68b100   none      null      local
```

Whenever a container is spawned, it is assigned to the `bridge` network by default. Containers connected to the `bridge` network can communicate with each other using IP addresses within the same Docker host. Docker assigns IP addresses to containers on the `bridge` network, and containers can use these addresses to interact with one another.

In contrast, the `host` network is the network of the host machine, meaning that a container connected to it will be directly using the host machine's network infrastructure. Consequently, this affects the isolation of the container by removing the network isolation between the container and the host completely. The `none` network, on the other hand, serves the purpose of disabling a container's networking capabilities.

In addition to the pre-defined networks, Docker permits the creation of custom networks. These customized networks offer enhanced container isolation and segmentation, enabling the establishment of distinct networks for different groups of containers. When a custom network is created, Docker establishes a virtual network that containers can join. Containers affiliated with the same custom network can communicate with one another by utilizing their respective IP addresses.

To override the default behavior and assign a container to a network other than

the `bridge` network, the `--network` option can be utilized when starting a container. [19]

3.6 Usage example

Suppose you had developed a simple Node.js application (see Listing 3.1) that should be run in a container. The first step in accomplishing it, after installing the necessary Docker command-line tools, is to define a Dockerfile which defines the steps needed to create an image. A suitable Dockerfile can be seen on Listing 3.2. After defining the Dockerfile, an image is built which is then used to spawn a running instance of a container.

```
1  // index.js
2
3  const app = require("express")();
4  const { readFile, writeFile, mkdir } = require("fs/promises");
5  const { dirname } = require("path");
6
7  const PORT = process.env.PORT || 8080;
8  const INDEX_PATH = "./data/index.txt";
9
10 app.get("/fibonacci", async (req, res) => {
11   const index = await readIndex();
12   const fib = fibonacci(index);
13   await writeIndex(index + 1);
14   res.json({ fib, index });
15 });
16
17 app.listen(PORT,
18     () => console.log(`App listening on ${PORT}...`));
19
20 function fibonacci(n) {
21   if (n === 0) return 0;
22   if (n === 1) return 1;
23   return fibonacci(n - 1) + fibonacci(n - 2);
24 }
25
26 async function readIndex() {
27   try {
```

```
    const data = await readFile(INDEX_PATH, "utf8");
    return parseInt(data);
  } catch {
    return 0;
  }
}

async function writeIndex(index) {
  await mkdir(dirname(INDEX_PATH), { recursive: true });
  await writeFile(INDEX_PATH, index.toString(), { flag: "w+" });
}
```

Listing 3.1: A minimal JavaScript web application that serves Fibonacci numbers based on the index of the request.

```
# Dockerfile

FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY index.js ./

ENV PORT=8080

EXPOSE 8080

CMD ["node", "index.js"]
```

Listing 3.2: An example of a Dockerfile for building a Docker image.

3.6.1 Dockerfile overview

Base image

While it is possible to start from scratch without a base image, it is usually more convenient to use a base image to build upon, such as the official Ubuntu image for example. However, every Dockerfile must begin with a `FROM` instruction, and in the case of Listing 3.2, the `node:18` image is specified as the base image to build upon. The instruction's argument `node` refers to the publicly available image of the Node.js runtime environment and the tag `18` to the version of it. If the version is omitted, the `FROM` instruction will use the latest version of the image by default, which is the same as specifying `FROM node:latest`.

Working directory

After the `FROM` instruction, the working directory of the application is specified to be the `/app` directory with the `WORKDIR` instruction. Now every subsequent instruction will start from the `/app` directory.

Copying files from host machine

Next, the `package.json` and `package-lock.json` files are copied from the host machine into the working directory of the application. Copying files is achieved with the `COPY` instruction where the first argument is the path of the file in the host machine and the second is the path in the container's working directory.

The `package.json` and `package-lock.json` files are configuration files that specify the dependencies of the application to other external NPM (*Node Package Manager*) modules, among other things. For example, as seen on line 3 of Listing 3.1, the application uses the *Express* framework, which is an external third-party dependency.

Running commands

Now that the information about the dependencies is in the container, they can be installed with the `npm install` command, which can be run inside the container with the `RUN` instruction.

Handling image layers

After installing the dependencies, the source code for the application will be copied over to the working directory with the `COPY` instruction. The reason why it is good practice to copy the source code only after installing the dependencies is the way Docker handles and caches the layers of the Dockerfile. As discussed in Section 3.5.3, each instruction in a Dockerfile corresponds to a layer in the resulting image and whenever an image is built from a Dockerfile, Docker will attempt to cache layers if nothing has changed. Since we can expect that the source code will be modified more often than the `package.json` files, the `RUN npm install` layer will be built only when the `package.json` file is changed. I.e., the NPM modules do not need to be installed every time the source code of the application changes. In more complex projects, with larger amounts of dependencies, this will potentially save a lot of time when building the image.

Environment variables

As seen on line 7 of Listing 3.1, the application is referencing an environment variable called `PORT`. In order for the application to be able to access this environment variable at runtime, it is defined in the Dockerfile with the `ENV` instruction. Note that if the environment variable is to be kept secret and not committed to a version control system along with the Dockerfile, it can also be passed to the container upon spawning it with the `-e` option. For example, `docker run -e PORT=8080 ...`.

Exposing ports

Next, the `EXPOSE` instruction is used to inform Docker that the container will listen on the specified port on runtime. The instruction does not actually publish the port; rather, it serves as a documentation mechanism to indicate which ports should be exposed when running the container. To actually publish the port and enable accessing the application from the host machine, port forwarding rules must be specified when starting the container, as will be described in Section 3.6.3.

Starting the application

Lastly, the `CMD` instruction defines how the container should run the application once it starts. Only one `CMD` instruction is permitted in each Dockerfile. [20]

3.6.2 Building an image

Having defined the set of instructions for building a Docker image in the form of a Dockerfile, the image can be built with the `docker build -t fib-app .` command. The `-t` option is used to name the image while the `.` argument denotes the path of the build's context. In this case, the path refers to the current directory, in which the Dockerfile resides along with the other necessary files.

After successfully building the image, it can be shared via a container registry and used as a base image for other images, or used to spawn containers.

3.6.3 Running a container

Spawning a container using the image can be accomplished with the `docker run` command. However, since the application needs to be accessible from outside the container, port forwarding from the container to the local machine must be implemented using the `-p` option. The `docker run --name fib-server -p 7000:8080 fib-app`

command can be used to spawn a container using the image of the demonstration application. The `--name` option gives the container a name for identification purposes, whereas the `-p` option maps the local machine's port 7000 to the port 8080 of the container. With port forwarding, the application can be accessed outside of the container, with a web browser, for example. Figure 3.3 shows the result of making a request to `localhost:7000/fibonacci` on the host machine with a web browser.

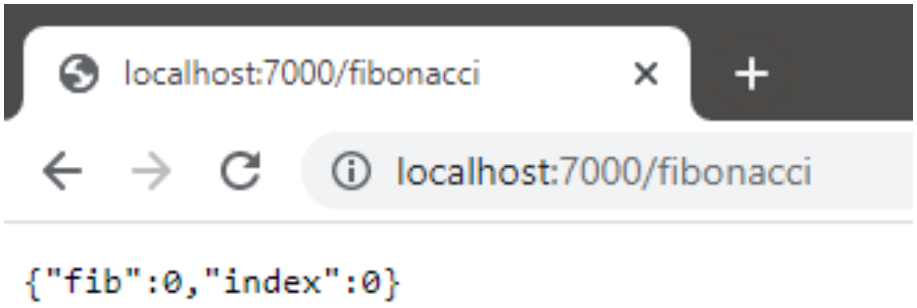

Figure 3.3: The JSON returned from the demo application running inside a container.

3.6.4 Networking

The Fibonacci application uses a text file as a crude and clumsy database to store information about the current value of the Fibonacci index counter. However, in production environments, an actual robust database implementation is usually desired. For this purpose, the Docker Hub contains ready-made images for many of the popular Database Management Systems (DBMS), such as Oracle Database, MySQL, PostgreSQL, and MariaDB.

To keep the concerns of the application business logic and data persistence separate, the sensible approach would be to dedicate a separate container to host a DBMS. A Dockerfile for constructing a container image for hosting the database can be seen on Listing 3.3. It uses the latest version of the MariaDB Relational Database Management System (RDBMS) as the base image. The Dockerfile also

defines the necessary environment variables and creates a table that meets the needs of the Fibonacci application along with a row containing the default values.

```
1  FROM mariadb:latest
2
3  ENV MYSQL_ROOT_PASSWORD=fib-admin
4  ENV MYSQL_DATABASE=fib-db
5
6  RUN echo "USE fib-db;" > /docker-entrypoint-initdb.d/init.sql
7
8  RUN echo "CREATE TABLE IF NOT EXISTS fib_index (" \
9      >> /docker-entrypoint-initdb.d/init.sql
10 RUN echo "id INTEGER PRIMARY KEY," \
11     >> /docker-entrypoint-initdb.d/init.sql
12 RUN echo "current_index INTEGER NOT NULL DEFAULT 0);" \
13     >> /docker-entrypoint-initdb.d/init.sql
14
15 RUN echo "INSERT IGNORE INTO fib_index VALUES (1, 0);" \
16     >> /docker-entrypoint-initdb.d/init.sql
17
18 EXPOSE 3306
```

Listing 3.3: Dockerfile for initializing a MariaDB database with a pre-populated table called *fib_index*.

After introducing a new container to the application, it then must be able to communicate and interface with the container running the Fibonacci application, which can be achieved with the networking capabilities of Docker. As described in Section 3.5.3, by default, Docker allocates containers to the *bridge* network unless instructed otherwise. However, for demonstrative purposes, a distinct network named *fibnet* will be employed. Its creation can be accomplished using the command `docker network create fibnet`. Grouping related containers that necessitate communication within their separate networks is a good practice in terms of isolation, as in general, containers are incapable of communicating beyond the confines of their respective networks.

An updated version of the source code of the Fibonacci application that takes

into account the migration from text-file-based data storage to a database can be seen on Listing 3.4. Notably, on line 9 of the listing, the address of the database container is specified by referencing the container's name. At runtime, the name of the container will be resolved to the corresponding IP address. Analogous to the DNS (*Domain Name System*) in the realm of the Internet, this simplifies the management of multiple containers within a network, as specific IP addresses need not be remembered — container names suffice.

```
1  // index.js
2
3  const app = require("express")();
4  const mysql = require("mysql");
5
6  const PORT = process.env.PORT || 8080;
7
8  const dbConnection = mysql.createConnection({
9    host: "fib-db",
10   user: process.env.DB_USER,
11   password: process.env.DB_PWD,
12   database: process.env.DB_NAME,
13 });
14 dbConnection.connect((error) =>
15   console.log(error ? error : "Database connected")
16 );
17
18 app.get("/fibonacci", async (req, res) => {
19   const index = await readIndex();
20   const fib = fibonacci(index);
21   await writeIndex(index + 1);
22   res.json({ fib, index });
23 });
24
25 app.listen(PORT,
26     () => console.log(`App listening on ${PORT}...`));
27
28 function fibonacci(n) {
29   if (n === 0) return 0;
30   if (n === 1) return 1;
31   return fibonacci(n - 1) + fibonacci(n - 2);
32 }
33
34 function readIndex() {
```

```
35   return new Promise((resolve) => {
36     dbConnection.query(
37       "SELECT current_index FROM fib_index WHERE id = 1",
38       (error, results) => {
39         if (error) return 0;
40         resolve(results[0].current_index);
41       }
42     );
43   });
44 }
45
46 function writeIndex(index) {
47   return new Promise((resolve, reject) => {
48     dbConnection.query(
49       "UPDATE fib_index SET current_index = ? WHERE id = 1",
50       [index],
51       (error, results) =>
52         (error ? reject(error) : resolve(results))
53     );
54   });
55 }
```

Listing 3.4: A minimal JavaScript web application that serves Fibonacci numbers based on an index number that is stored in an external database.

To assign a container to a network other than the default *bridge* network, the `--network` option must be specified during the container's instantiation. For instance, to spawn the container hosting the Fibonacci application and assign it to the *fibnet* network, the following command can be utilized: `docker run --name fib-server --network fibnet fib-app`. Subsequently, any subsequent containers within the same *fibnet* network can communicate with it via HTTP requests by referencing the container name. For instance, after constructing an image based on the Dockerfile provided in Listing 3.3 and assigning a container instantiated from said image to the *fibnet* network, the Fibonacci application can communicate with it by simply referring to its name and a port and vice versa.

3.6.5 Persisting data

In the presented Fibonacci application, the returned Fibonacci number is calculated based on a counter which is incremented during each request made to the `/fibonacci` endpoint. However, as discussed in Section 3.5.3, the internal state of containers is ephemeral. Any file or data a container might write to its file system will not survive container restarts unless written to a Docker volume. Thus, in order to persist the application's database, and subsequently the value of the counter, beyond the lifecycle of the container, a volume must be created and mounted to the container.

A volume named *fib-vol* can be created with the `docker volume create fib-vol` command. The volume can then be attached to the container upon spawning it with the `-v` option. For example, the command `docker run --name fib-db --network fibnet -v fib-vol:/var/lib/mysql fib-db` spawns a database container defined in Listing 3.3 with the `fib-vol` volume mounted to the `/var/lib/mysql` directory of the container, which MariaDB uses by default to store its database data.

3.7 Managing containers with Docker Compose

As the number of containers within a system grows, managing them individually becomes increasingly complex. As seen in the previous usage example of a two-container system, each container requires separate configurations for various aspects such as ports, volumes, and networks. This proliferation of configurations can quickly become challenging to handle, leading to potential errors and difficulties in maintaining consistency across multiple containers. To address these challenges and streamline the management of containerized applications, *Docker Compose* provides a solution.

Docker Compose is a tool that simplifies the deployment and management of multi-container applications. It allows developers to define and orchestrate the con-

figuration of multiple containers within a single configuration file, known as the Compose file. This file serves as a centralized location for specifying the desired state of the application, including container dependencies, network connections, volumes, and environment variables among other things.

With Docker Compose, the definition of containers and their associated configurations can be encapsulated within a single Compose file. This eliminates the need to manually configure each container separately, reducing the complexity and potential for errors.

One of the key capabilities of Docker Compose is its ability to define and manage the relationships between containers. Dependencies between containers, such as one container relying on the availability of another, can be expressed in the Compose file. Docker Compose then ensures that containers are started in the correct order, taking care of inter-container communication and synchronization.

Furthermore, Docker Compose simplifies the management of networks, volumes, and environment variables across containers. It provides a unified interface for defining and configuring these resources, allowing developers to specify network connections, mount volumes, and set environment variables for all containers in a centralized manner. This not only reduces the effort required to manage these aspects but also improves the consistency and reproducibility of the application's environment.

For example, instead of individually starting containers and specifying each of their configurations and parameters by hand, such as environment variables, port forwarding rules, volumes, and networks, we can write a single compose file (see Listing 3.5) using YAML syntax, that takes care of the aforementioned variables. Docker-compose files are comprised of interconnected elements that collectively define the structure and behavior of a multi-container application. Notably, they are composed of services, volumes, and networks. In the case of Listing 3.5, there are

two primary services defined; `fib-db` and `fib-server`, each representing a distinct containerized component. `fib-db` service houses the container running the MariaDB container, whereas `fib-server` is concerned about the container running the main application. The paths to the respective Dockerfile directories (build contexts) are defined in the `build` property of the service. In addition to this, the necessary environment variables (`environment` property), volumes (`volumes` property), port forwarding rules (`ports` property), and networks (`networks` property) are also defined per service basis. However, in order to be able to specify volumes and networks, they must be defined in the `volumes` and `networks` sections respectively.

```
1  # docker-compose.yml
2  version: "3.8"
3  services:
4    fib-db:
5      build:
6        context: ./fib-db
7      environment:
8        - MYSQL_ROOT_PASSWORD=fib-admin
9        - MYSQL_DATABASE=fib-db
10     volumes:
11       - fib-vol:/var/lib/mysql fib-db
12     ports:
13       - 3306:3306
14     networks:
15       - fib-net
16   fib-server:
17     build:
18       context: ./fib-server
19     ports:
20       - 7000:8080
21     networks:
22       - fib-net
23 volumes:
24   fib-vol:
25 networks:
26   fib-net:
```

Listing 3.5: Docker-compose file that configures a multi-container application.

4 Web application containerization

4.1 Research methodology: literature analysis

The successful implementation of containerizing a web application relies on an understanding of what advantages and improvements containerization can bring to the application and its deployment process. It is also important to understand and acknowledge the drawbacks and challenges that containerization might bring along.

The purpose of this chapter is to provide an answer to the first research question (*RQ1*) outlined in Chapter 1: *What benefits can be gained by utilizing container technologies with an application like eShare? How about the drawbacks?*. To accomplish this, a literature analysis delving into existing literature on the topic of implementing containerization and migration to containerized solutions is conducted in the following sections. The literature analysis aims to uncover the benefits and drawbacks of containerization that others have experienced when implementing containerization and migrating to containerized solutions. The findings of the literature analysis are then used to answer RQ1.

4.1.1 Source of literature

To gather relevant literature, an approach relying on academic databases and repositories was employed. Specifically, Google Scholar, IEEE Xplore, and UTU Volter (digital publications of the University of Turku Library) were leveraged to access a

variety of academic literature on the topic.

A set of keywords and phrases revolving around the topic of applying and using containerization technologies were used to cast a wide net for relevant literature when searching the databases. The results of these searches were then filtered based on the criteria described in the next section. The most promising results were selected for a more detailed inspection.

4.1.2 Criteria for literature selection

Selecting the most pertinent literature from the vast amount of literature available on the topic was a challenging task due to the sheer volume of the literature and the broadness of the topic. However, it is crucial in order to ensure the validity and reliability of the findings. The following criteria were used to determine the suitability of the literature search results for the analysis:

Relevance: The literature had to directly relate to the topic of implementing containerization and indicate that it might contain information relevant to answering RQ1. This was evaluated based on the title and abstract of the publication. This is a somewhat subjective criterion, and as such, it was evaluated based on my personal judgment.

Publication date: The age of the literature was a key consideration. Preference was given to recent publications, preferably from 2015 onwards, to ensure the information remains current and relevant in the fast-evolving field of web application development.

Citation count: The number of citations a publication has received was also considered, as it is a good indicator of the relevance and impact of the publication. Publications with under 10 citations were discarded and not considered for the

analysis whereas anything over this threshold was considered for further inspection.

4.1.3 Search process

The search process was conducted in the following manner. First, multiple broad searches were conducted using different sets of keywords and phrases, as mentioned in the previous section. Then, for each search, the first 40 results (sorted by relevance) were skimmed and analyzed to determine whether they matched the criteria described in the previous section or not. The results that matched the criteria were then selected for a more detailed inspection, which consisted of reading the introductions and conclusions of the publications and skimming through the rest of the publication to determine its relevance and suitability for the analysis (i.e., whether it contained information relevant to answering RQ1). The most promising publications were then selected for analysis.

The first broad search was conducted using the search term *docker adoption OR containers adoption OR docker lessons learned OR containers lessons learned*. This yielded a total of 19,200 results in Google Scholar, 8,771 results in Volter, and 340 results in IEEE Xplore. From these results, a total of 10 matched the described criteria and were selected for further inspection.

The second broad search was conducted using the search term *docker experience report OR containers experience report OR case study docker OR case study containers*. This yielded a total of 25,300 results in Google Scholar, 10,096 results in Volter, and 914 results in IEEE Xplore. From these results, a total of 7 matched the described criteria with 3 of them being duplicates from the previous search. The remaining 4 results were selected for further inspection.

The third and final broad search was conducted using the search term *(docker OR containers) AND (challenges OR drawbacks OR best practices)*. This yielded a

total of 705,000 results in Google Scholar, 9,339 results in Volter, and 1,902 results in IEEE Xplore. From these results, a total of 8 matched the described criteria with 3 of them being duplicates from the previous searches. The remaining 5 results were selected for further inspection.

After the initial broad searches, a total of 19 publications matched the described criteria and were selected for further inspection where they were read more thoroughly to determine their relevance and suitability for the analysis. And if they were deemed relevant and suitable, they were selected for the literature analysis and the relevant information was extracted from them. Upon further inspection, 13 out of the 19 publications were deemed relevant and suitable for the analysis. The remaining publications were discarded. The findings of these publications are discussed and presented in the next section.

4.2 Overview of the findings in the literature

The findings of the literature analysis are presented in Table 4.1. Each publication is labeled as P1, P2, P3, etc. The benefits and drawbacks of containerization reported in the publications are listed in the benefits and drawbacks columns, respectively. The severity of each of the benefits and drawbacks is categorized as either *major* (3), *minor* (2), or *neutral* (1). The severity is determined based on my personal assessment of how significantly it would affect an application like eShare. In addition to the table, a graph visualizing the number of times each benefit and drawback was reported in the literature is presented in Figure 4.1. The benefits and drawbacks are discussed in more detail in the following sections.

Publication	Summary	Benefits	Drawbacks
P1 [21]	Focuses on Kubernetes in a private cloud for microservices. It evaluates the availability through experiments in pod and node failure scenarios.	**Performance** (2): Containers are lightweight and restart fast. **Availability** (1): Failure of one containerized microservice does not affect other services. **Scalability** (3): Independent evolution of each service.	**Complexity** (2): Challenging deployment and configuration in private cloud Kubernetes clusters. **Availability** (1): Significant service outages are possible with default Kubernetes configuration.
P2 [22]	Discusses replicable performance experiments using Raspberry Pi and Docker. Focuses on micro- and macrobenchmark experiments, performance fluctuations, and experimental setup.	**Performance** (2): Efficient compared to virtual machines and quick start and stop times. **Reproducibility** (2): Easy to create and configure reproducible environments.	**Performance** (2): I/O processes can be resource intensive. Unexplained high variances in response times also reported.
P3 [23]	Discusses the measurement of Docker's performance, focusing on CPU and disk I/O overheads, and highlights the difficulties in measuring and monitoring container performance.	**Performance** (1): Lower overhead compared to VMs. **Abstraction** (1): A higher level of process lifecycle management. **Portability** (2): As a microservice a container can be executed on any platform supporting containers.	**Monitoring** (1): Not many stable and dedicated tools for monitoring container performance, and the available ones can give different or incomplete results. **Performance** (2): 10% to 30% overhead in CPU and disk I/O.
P4 [24]	Discusses the use of Docker in deploying and testing astronomy software, highlighting Docker's advantages in software container management and its rapid development within the open-source community.	**Reproducibility** (2): Easy to describe and manage software containers. **Performance** (2): More system resources are available for applications due to lower overhead compared to VMs.	**Complexity** (2): Managing and understanding container ecosystems can be complex. **Security** (1): Potential security risks with privileged access in containers.

Continued on next page

Table 4.1 – *Continued from previous page*

Publication	Summary	Benefits	Drawbacks
P5 [25]	Evaluates Docker containers' impact on genomic pipeline performance.	**Portability** (2): Docker facilitates easy distribution and execution across different computing platforms. **Reproducibility** (2): Containers provide isolated environments, ensuring predictable system configurations. **Performance** (2): Only a negligible overhead for medium/long-running tasks, which are common in genomic pipelines.	**Performance** (1): Noticeable overhead for pipelines with very short tasks due to container instantiation. **Complexity** (2): Managing the workflow of multistage pipelines with Docker can be complex. **Flexibility** (2): Platform limitation due to native Docker installation depending on the Linux kernel.
P6 [26]	Discusses the transition from monolithic enterprise architectures to microservices in three Brazilian government institutions. It explores the motivations, benefits, and challenges faced during this two-year migration process, validated by a survey with 13 practitioners involved in the migration.	**Agility** (1): Microservices reduce deployment risks and speed up development. **Flexibility** (1): Containers enable the use of different programming languages, frameworks, and tools than the rest of the system is using. **Scalability** (3): Containers facilitate the scaling of individual services based on demand.	**Complexity** (2): Adoption of microservices architecture demands an understanding of new techniques and tools. **Effort** (2): Requires significant changes in development process and architecture of legacy systems.
P7 [27]	Explores container technologies with a focus on security, discussing the emergence of containers as a lightweight alternative to VMs, their role in microservice architecture, and the major concerns and challenges in container security.	**Performance** (2): Efficient to start and stop, better resource utilization than VMs as a shared OS kernel reduces duplicate OS copies. **Scalability** (2): Effective for employing microservices architecture.	**Isolation** (2): Some resources are not namespace-aware, risking isolation. **Security** (3): Vulnerable to resource drainage attacks if not properly managed.

Continued on next page

Table 4.1 – *Continued from previous page*

Publication	Summary	Benefits	Drawbacks
P8 [28]	Evaluates Docker and Singularity containers for scientific workloads in cloud environments, focusing on performance, RDMA communication, and resource management in parallel workloads.	**Performance** (2): Near-native performance for scientific workloads. **Flexibility** (2): Docker offers fine-grained resource allocation.	**Security** (2): Docker containers have been reported to have root escalation vulnerabilities.
P9 [29]	Discusses the transition from a monolithic architecture to microservices in Danske Bank's FX Core system, focusing on scalability improvements and system rearchitecture.	**Scalability** (2): Improved scalability due to microservices representing single business capabilities that can be scaled independently. **Isolation** (2): Services are independently deployable, reducing the impact of bugs and errors. **Availability** (1): Self-healing services with Docker Swarm, enabling service discovery and load balancing. **Flexibility** (1): Technology independence; reduced reliance on specific technologies and platforms.	**Complexity** (2): Challenging deployment and management due to the distributed nature of microservices. Due to this, also a high dependency on infrastructure management and automation tools.
P10 [30]	Argues for a paradigm shift in software engineering towards service agent orientation, facilitated by the CAOPLE programming language and CIDE DevOps environment. It focuses on cloud computing's demands on scalability, performance, and reliability, addressing these challenges through microservices, container technology, and DevOps.	**Scalability** (2): Microservices and containers enable scaling individual components. **Availability** (2): Enhanced fault tolerance with microservices architecture.	**Complexity** (2): Increased complexity in deployment and monitoring of multi-container systems. **Performance** (2): Challenges with network latency due to intensive inter-service communication.

Continued on next page

Table 4.1 – *Continued from previous page*

Publication	**Summary**	**Benefits**	**Drawbacks**
P11 [31]	Examines the integration of Docker containers with VMs in cloud computing, focusing on performance, isolation, and system management. Empirical studies were conducted on platforms like KVM, XEN, and Hyper-V to evaluate performance overhead and energy consumption.	**Performance** (2): Containers can be created and destroyed much faster than VMs. **Scalability** (3): Fast startup times allow on-demand scaling and utilization. **Flexibility** (2): Containers are adaptable to different cloud infrastructures.	**Performance** (1): Additional virtualization layer can lead to performance overhead compared to just VMs or containers alone.
P12 [32]	Examines Open5GCore's performance on KVM and Docker using Open5GMTC for IoT/M2M traffic. Studies virtualization's impact on network services and compares the two environments.	**Performance** (2): Comparable performance to physical machines.	**Performance** (1): Noticeable memory overhead when utilizing Docker. **Complexity** (1): Docker requires a complex setup for Open5GCore.
P13 [33]	Examines Docker for high-performance computing applications, using AutoDock3 as a test case. Compares Docker containers with VMs in terms of efficiency and memory management.	**Performance** (2): Quick start-up times. **Scalability** (2): Due to fast start-up times, containers are suitable for applications requiring real-time launching of resources.	**Performance** (1): Challenges in handling extremely high memory allocations.

Table 4.1: Findings of the literature analysis. The severity of each benefit and drawback is categorized as either *major* (3), *minor* (2), or *neutral* (1).

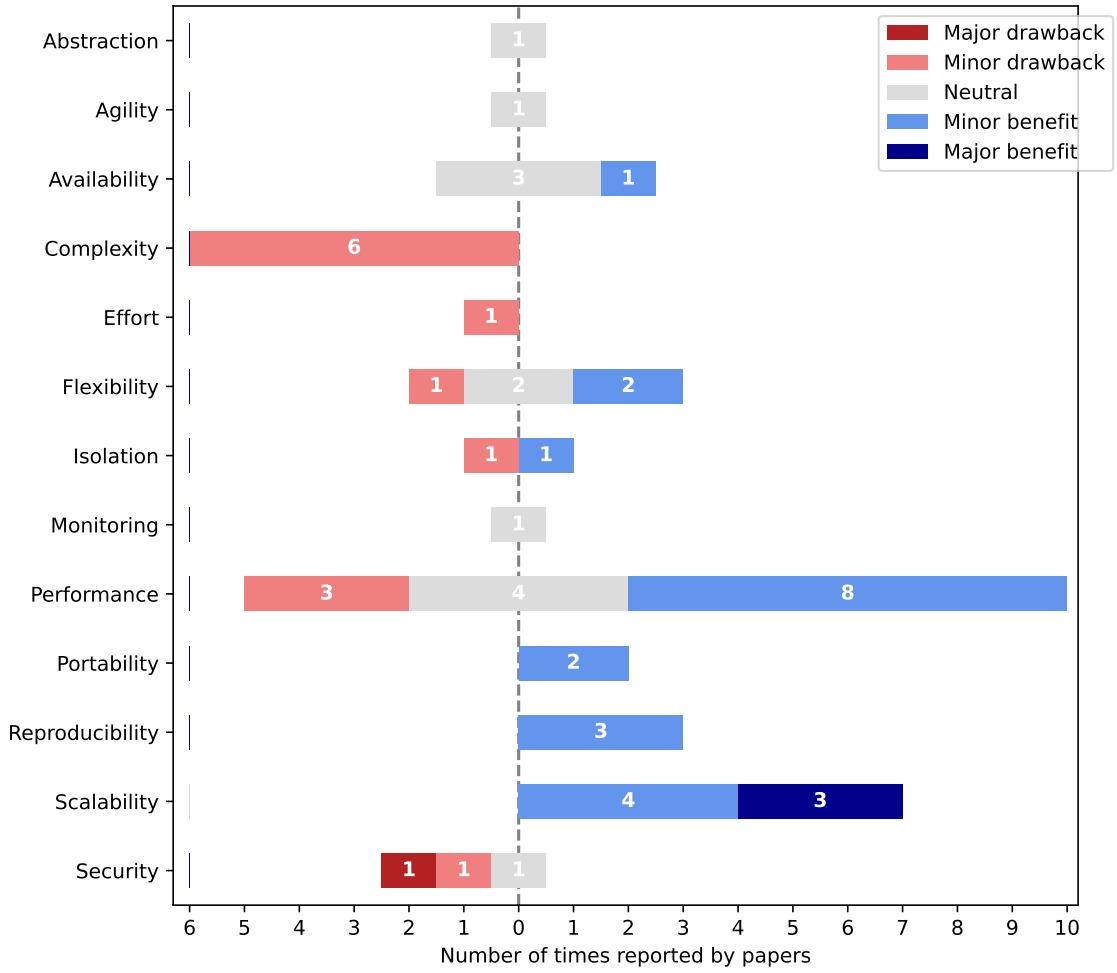

Figure 4.1: A graph visualizing the number of times each of the benefits and drawbacks was reported in the literature.

4.3 Insights from the literature

A recurring theme across the reviewed literature is the emphasis on several key benefits of containerization. Notably, performance improvements and scalability are frequently cited advantages, as visualized in Figure 4.1. It is also notable that the significance of scalability is highlighted by the fact that it was rated as a major benefit three times, while the other benefits were only rated as either minor or neutral.

However, it is also important to acknowledge the drawbacks and challenges that containerization can introduce. Complexity, performance overhead, and security risks are the most frequent drawbacks reported. It is worth noting that only one drawback was rated as major, while the other drawbacks were either minor or neutral.

The following sections discuss the benefits and drawbacks of containerization that were reported in the literature in more detail.

4.3.1 Relevancy of the findings

Since the field of containerization is still relatively new and rapidly evolving, it is important to keep in mind that the relevancy of some of the findings in the literature may have changed since the publications were published. As of writing this, the publication dates of some of the publications date as far as 8 years ago. For example, some security-related aspects are likely patched or otherwise addressed in modern versions of Docker and other container technologies while new security issues may have emerged.

However, certain aspects regarding performance, scalability, and complexity are likely to remain relevant for the foreseeable future as they are inherent to the nature of containerization. For example, the distributed nature of microservices architecture and container orchestration techniques will likely continue to introduce complexity and require additional expertise and resources to manage effectively. Similarly, the lightweight nature of containers will likely continue to offer performance-related benefits compared to traditional virtual machines.

4.3.2 Prevalent benefits of containerization

The literature consistently highlights several key benefits of containerization, each with varying levels of impact and significance.

Performance : A common observation is the lightweight nature of containers leading to faster start and stop times compared to traditional virtual machines. This aspect is crucial for web applications like eShare that demand quick responsiveness. Out of the 13 publications, 9 reported some level of performance-related improvements as a benefit of containerization.

Scalability : The ability to scale services independently is a highly valued feature in microservice architectures. Containers enable this by allowing individual components of an application to be scaled based on demand, which is particularly relevant for fluctuating workloads in web applications. However, it is worth noting that increased scalability was most of the time attributed to microservices architecture rather than containerization itself. Scalability was reported as a benefit in 7 out of the 13 publications.

Flexibility : Being able to use different programming languages and technologies for different components of an application is another frequently cited benefit. Also noted is the ability to deploy containerized applications across different computing platforms, which is particularly relevant for cloud-based applications. Flexibility was reported as a benefit in 4 out of the 13 publications.

Reproducibility : Another somewhat common observation is the benefit of reproducibility. Containers were reported to provide an easy way to create and configure reproducible environments, which is invaluable for ensuring consistent deployment across different environments. Reproducibility was reported as a benefit in 3 out of the 13 publications.

Availability : The utilization of microservices architecture and container orchestration techniques, such as Docker Swarm and Kubernetes, was noted to enhance the availability of applications. Orchestration has been reported to improve fault tolerance and enable self-healing services, which are critical for ensur-

ing high availability. Availability was reported as a benefit in 3 out of the 13 publications.

4.3.3 Recognized drawbacks and challenges

While containerization offers significant advantages, it also introduces certain drawbacks that require careful consideration.

Complexity : A frequently cited challenge is the complexity involved in deploying and managing containerized systems. This seems to be mainly due to the distributed nature of applications utilizing microservices architecture and container orchestration techniques. This complexity can introduce a steep learning curve and may require additional expertise or resources to manage effectively. Complexity was reported as a drawback in 6 out of the 13 publications.

Performance : Although containers were generally reported to have a lower performance overhead and be more efficient than traditional virtual machines, many of the publications noted that containers can still introduce a certain level of performance overhead compared to running the application natively on the host operating system. However, the performance-related drawbacks were generally rated with a low level of significance, while the performance-related benefits were rated with a higher level of significance. Out of the 13 publications, 6 reported performance-related drawbacks.

Security : Out of the 13 publications, 3 pointed out potential security risks associated with containerization, such as isolation issues and vulnerability to certain types of attacks, such as resource drainage attacks. This is a critical aspect to consider, especially for applications dealing with sensitive information.

4.3.4 Implications for eShare-like applications

Applications like eShare, monolithic in nature and mostly deployed on-premise, stand at a technological crossroads where the adoption of container technologies can significantly shape its future architecture, performance, and scalability. The literature analysis provided a perspective on the potential implications of containerization for eShare-like applications. The findings suggest that while containerization can offer substantial improvements in performance and scalability, it is imperative to approach its implementation with a strategy to manage the increased complexity and address potential security issues.

Performance and scalability

The inherent lightweight nature of containers can lead to notable performance improvements for eShare and applications similar to it. Reduced start and stop times are particularly advantageous for a web-based application, ensuring rapid responsiveness that is crucial for user experience and satisfaction. Furthermore, the scalability aspect of container technologies allows for more efficient resource allocation and service scaling based on demand, aligning with eShare's desired operational dynamics of handling varying workloads effectively.

Complexity in management

While container technologies promise enhanced performance and scalability, they also introduce a layer of complexity in the deployment and management of services. Transition to a containerized environment may require a robust strategy to address the steep learning curve and the necessity for specialized expertise. This complexity is not just operational but also conceptual, demanding a comprehensive understanding of container orchestration techniques and microservices architecture.

Security considerations

Security remains an important concern, especially for a data-centric platform like eShare. Containerization, while isolating applications, presents challenges such as ensuring the security of containers, managing resource consumption, and protecting against potential breaches. Addressing these concerns necessitates a proactive approach, incorporating best practices in container security and utilizing up-to-date tools and technologies.

Conclusion

The conducted literature analysis provided insights into the potential benefits and drawbacks of containerization for web applications like eShare. To answer RQ1 (*What benefits can be gained by utilizing container technologies with an application like eShare? How about the drawbacks?*), the discovered benefits and drawbacks are listed in detail in Table 4.1 and visualized as a graph in Figure 4.1.

While the adoption of container technologies offers significant advantages in terms of performance and scalability, they also introduce complexity and overhead compared to deploying applications natively on the host operating system. The findings suggest that the adoption of container technologies for eShare should be approached with a strategy that addresses the increased complexity.

The next chapter will delve into the practical aspects of containerizing eShare, providing a plan for its implementation and migration to a containerized environment based on the insights gained from the literature analysis.

5 Case study: eShare

5.1 Introduction

In the landscape of modern software development, containerization has become a popular approach to software deployment and delivery. The benefits and drawbacks of containerization have been discussed in the previous chapter, and this chapter will build upon that discussion by exploring the application of containerization in the context of a real-world software system in the form of a case study. The case study is centered on eShare, a web-based information management software product that was described in more detail in Chapter 2.

This chapter will begin with an introduction to the case study, outlining the objectives and motivation behind the study. This will be followed by a technical overview of eShare's architecture and the potential challenges and bottlenecks that may be encountered in the process of containerization. The chapter will then continue by presenting a containerization solution for eShare, including the plan and implementation of the solution. The chapter will conclude with remarks on the implementation and suggestions for future work. Section 5.4.2 also acts as the answer to RQ3 (*What would be a sensible containerization solution for an application like eShare?*) presented in the introduction.

5.1.1 Objectives

The primary objective of this study is twofold: firstly, to undertake and evaluate the feasibility of partial containerization of eShare in an attempt to profit from the benefits of containerization that were identified in the previous chapter (Chapter 4), and secondly, to set the stage for a future transition towards cloud-based deployment and Software as a Service (SaaS) model, which Cadmatic is exploring and pursuing as a potential future business model for eShare. The latter is a long-term goal that is not within the scope of this study but is nonetheless an important consideration in the context of the case study.

Central to these objectives is the evaluation of the containerized solution in the context of eShare, specifically assessing its performance and outlining the benefits and limitations inherent to the process of containerization.

5.1.2 Current operational bottlenecks

Currently, eShare encounters occasional operational bottlenecks, primarily due to certain computationally intensive server-side tasks. These tasks, such as *point cloud processing*, *document link-injection*, *document conversions*, and *model publishing* suffer from sluggish performance that sometimes negatively affects the user experience. This problem can be of course mitigated by scaling the infrastructure vertically (increasing the power of the existing system), but this is not a sensible solution in terms of cost-effectiveness, especially in cloud environments, where eShare is planning to move in the future, as eShare does not demand a high level of hardware resources most of the time. But when it occasionally does, the current infrastructure is not well-suited to handle these spikes in demand.

5.2 Background

The advent of containerization has significantly impacted the landscape of web application development and deployment, offering a robust solution to many longstanding challenges faced by developers and system administrators alike. This section delves into the architectural overview of eShare, the challenges and bottlenecks that might affect the containerization process, and how containerization emerges as a promising solution to these issues.

5.2.1 Overview of eShare's architecture

As previously described in Chapter 2, eShare is a monolithic client-server web application that acts as a central platform for sharing and viewing engineering data and information, such as 3D models, point clouds, and documents. The application is designed to be used in the context of small to large-scale engineering projects, such as shipbuilding, where the management and sharing of large and complex data sets are essential in order to track the status and progress of the project and to ensure that all stakeholders have access to the relevant up-to-date information.

As such, most of the engineering data eShare operates on is not stored or managed by eShare itself, but rather accessed and processed from external systems, such as databases, document management systems, and spreadsheet files. The data is accessed and processed by the eShare server, which then serves it to the client applications, such as web browsers and native desktop applications, which are used by the end-users.

Server

The eShare server is a monolithic application, written in C# and running on the .NET platform. The server is responsible for handling the business logic, data processing, and communication with external systems. The server is also responsible

for serving the client applications with the data they request. For accessing external systems, eShare provides a set of configurable *Data Source Adapters* for interfacing with different types of data sources, such as databases, REST APIs, and spreadsheet files. The server also performs periodical background-tasks, such as indexing documents and processing point clouds, which are often computationally intensive and can be a bottleneck for the system. The server is hosted on IIS (*Internet Information Services*) and uses Microsoft SQL Server as the database management system. The high-level overview of the server-side architecture of eShare is visualized in Figure 5.1.

Since most of the heavy tasks are performed by the server, the containerization efforts will be focused on the components and services running on the server.

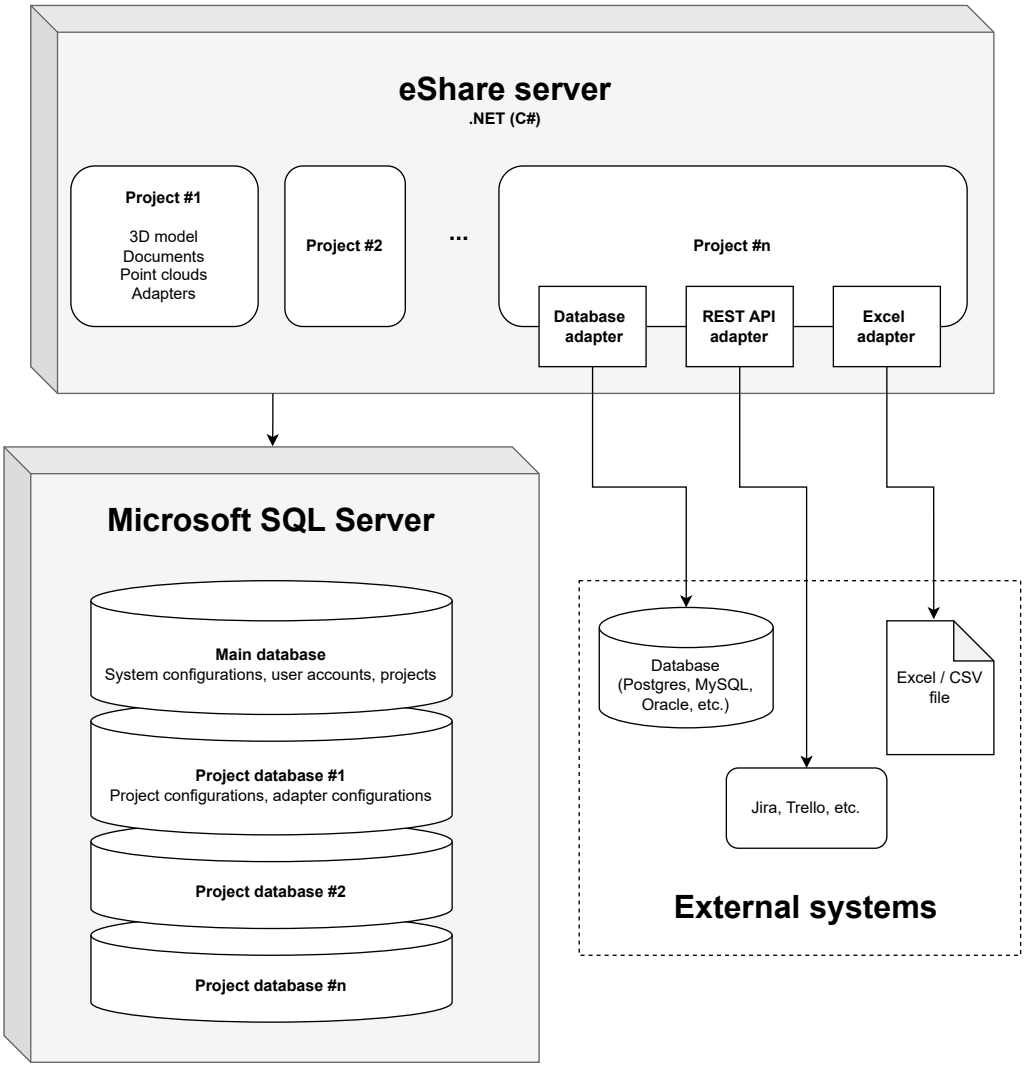

Figure 5.1: High-level overview of eShare's server-side architecture.

Client

The eShare client application is a web-based single-page application written in TypeScript and Angular. The client application is responsible for rendering the data and providing the user interface for interacting with the data. The 3D visualization of models and point clouds is handled by Cadmatic's proprietary visualization technology written in C++ and compiled to Web Assembly using Emscripten. The client-server communication is handled by a REST API provided by the eShare

server.

Deployment

Deploying eShare requires a Windows environment, as it is hosted on IIS (*Internet Information Services*) and uses Microsoft SQL Server as the database management system. Currently, eShare is most often deployed on-premise, on a per-customer basis. Cloud-deployment is also possible by using a Windows-based virtual machine but, as of yet, it is not very common among eShare's customers.

5.2.2 Potential challenges in containerization

Challenges in the current architecture

The transition to containers raises some questions about potential challenges that might surface during the containerization process. Particularly, since Docker is dependent on the Linux kernel, the fact that eShare server is a Windows-based application might pose some challenges. However, the .NET platform, which eShare is built on, is nowadays mostly cross-platform, thanks to the introduction of .NET Core. This means that the amount of Windows dependencies should be minimal or even zero, which should make the transition to containers easier.

Another possible challenge is the handling of large files and their transfer to and from containers. For instance, some of the point cloud files might exceed 100 gigabytes in size. If the container, to which the point cloud file is assigned to, operates on the same host as eShare server, the challenge is only a matter of assigning an appropriate volume to mount to the container. However, if the container is on a completely different host, the problem might be more complex to address properly.

Challenges discovered in literature

In addition to the challenges specific to the existing architecture of eShare, the literature analysis in Chapter 4 also highlighted some potential challenges that might arise in the process of containerization. Most notably, the complexity of managing and orchestrating containers, and the security implications of containerization, are two aspects that need to be considered. However, this case study will focus on containerizing only a part of the eShare server, and thus the challenges related to the complexity of managing and orchestrating containers should be fairly limited.

5.2.3 Containers as a solution

As was discovered in the previous chapter (Chapter 4), containerization together with microservices architecture offers a promising solution to the challenges and bottlenecks described in Section 5.1.2. For example, increased performance and scalability were identified as the most cited benefits of containerization in the literature analysis, and these are precisely the issues that eShare is currently facing regarding certain server-side tasks and services. By encapsulating these server-side tasks, services, and responsibilities into containers in a microservice, the application can achieve more fine-tuned scaling capabilities and streamline the transition between on-premise and cloud deployments. The use of containers and microservices architecture not only offers a potential solution to the immediate operational bottlenecks but also sets a foundation for a more cost-effective, maintainable, and scalable system architecture in the future.

5.3 Plan for a proposed containerization solution

5.3.1 Selecting the target for containerization

In many of the cases in the literature analysis conducted in Chapter 4, the reported performance and scalability benefits of containerization were achieved by utilizing microservices architecture. Thus, after careful evaluation of the findings in the literature analysis and discussion with some of the developers working with eShare, it was decided that the best approach to address the performance bottlenecks through containerization would be to build independently scalable microservices that encapsulate the computationally intensive tasks.

As a starting point, the target for containerization was chosen to be the server-side tasks related to reading, writing, and converting documents from one format to another. These tasks are currently implemented as independent stand-alone executables and are relatively decoupled from the rest of the system, and thus should be fairly trivial to extract from eShare and containerize.

Currently, these document processing tasks are triggered whenever a user opens a document for viewing in eShare. Depending on the server hardware and the complexity of the document, these tasks can take several seconds to complete which often severely affects the user experience negatively. Solving this problem with the performance of the document processing tasks would significantly improve the user experience of eShare.

5.3.2 Document processing microservice

The initial plan is to build a microservice that encapsulates these tasks into a containerized application. The microservice would then be deployed as an independent service, which would be responsible for processing documents. Whenever document processing is requested, the eShare server would send a message to the document

processing microservice, which would process the document and as a result, return the relevant data to the eShare server.

The plan is also that, in a production environment, the document processing microservice would be deployed on a separate more powerful infrastructure reducing the load a single eShare server instance would have to handle. This approach would also allow the document processing microservice to serve multiple instances of eShare at the same time, which would improve the cost-effectiveness of the system as a whole, as the idle time of the more powerful and expensive document processing microservice hardware would be minimized.

5.3.3 Event-driven communication

After discussing the details of the potential containerization solution with some of the eShare developers, it was decided that an event-driven communication mechanism would be used for communication between the eShare server and the document processing microservice. Event-driven communication is a pattern where the sender of a message (event) does not need to know the receiver of the message. Instead, the sender publishes the message to a message bus, and the receiver subscribes to the message bus to receive the message. This pattern is well-suited for the proposed containerization solution, as it reduces coupling and allows for a distinct separation between the eShare server and the document processing microservice (and any future microservices), which is ideal in a distributed environment.

Whenever a document processing is requested, eShare server can publish a message to a designated message bus, and the document processing microservice can, in turn, subscribe to this bus to receive the message and process the document. This is especially useful when the document processing microservice is scaled horizontally to multiple instances, as the messages can be distributed among the instances by the message broker, which reduces the complexity of managing multiple microservice

instances.

The message bus technology chosen for this purpose is Apache Kafka, which is a distributed event streaming platform that is well-suited for this kind of event-driven communication. Kafka is a popular choice for this kind of communication, as it is highly fault-tolerant and designed for high throughput. [34]

5.3.4 Overview of architecture

The architecture of the proposed containerization solution is visualized in Figure 5.2. The proposed solution revolves around the Kafka message broker, which is used for event-driven communication between the eShare server, the document processing microservice, and any other potential future microservices. Instances of eShare servers utilize the microservices by publishing messages to specific Kafka topics, which are then consumed (subscribed) by the relevant microservices. Conversely, the microservices publish messages back to separate Kafka topics, which are consumed by the eShare server to retrieve the results of the processing. The flow of a document processing request beginning from the eShare client application and ending with the document being served to the client is visualized as a sequence diagram in Figure 5.3.

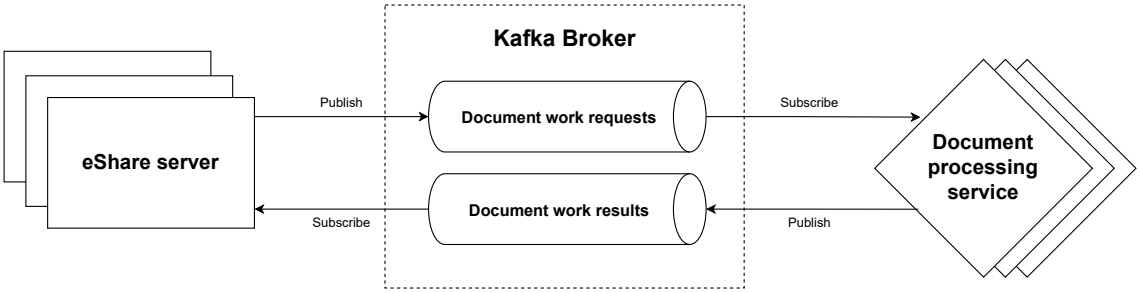

Figure 5.2: High-level overview of the proposed containerization solution for document processing in eShare. *Document work requests* and *Document work results* are Kafka topics to which eShare server and document processing microservice publish and subscribe messages.

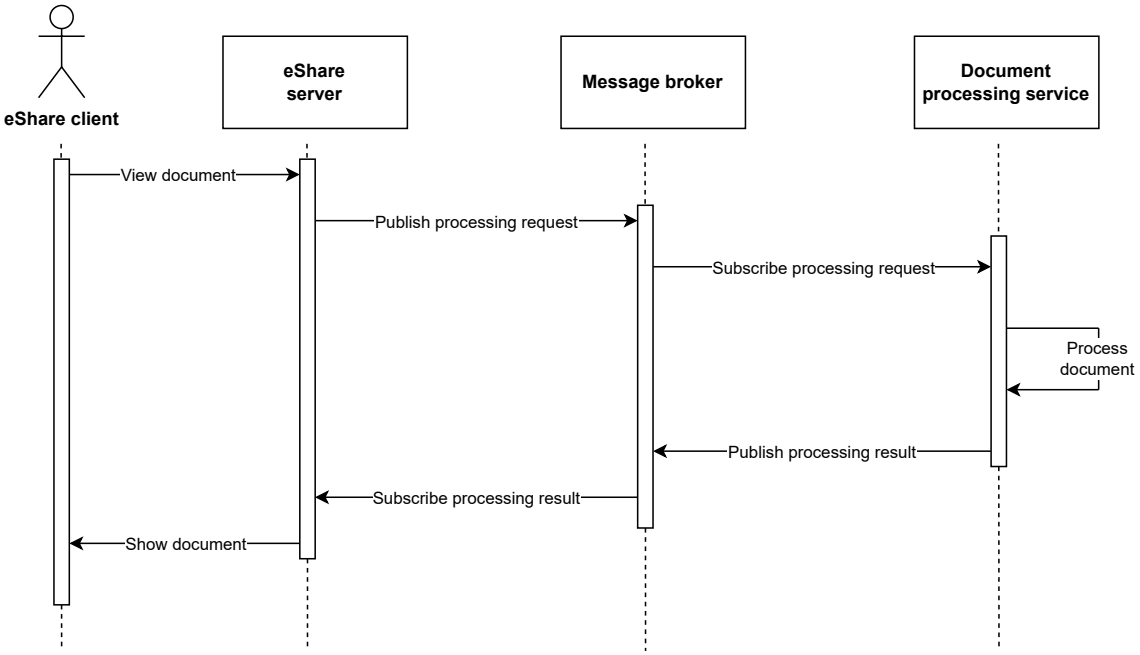

Figure 5.3: Sequence diagram illustrating the flow of a document processing request in the proposed containerization solution.

5.4 Implementation

5.4.1 Containerization process

Document processing microservice

The first step in the containerization process was to implement the document processing microservice, which encapsulates the document processing tasks into a containerized application. The document processing microservice was implemented as a .NET application, which is the same platform that eShare server is built on. This choice was made due to the familiarity of the developers with the platform and the ease of integration with the existing eShare codebase.

A significant development was the creation of an abstract base class designed to streamline the development of Kafka-driven microservices. This abstraction facilitates the development and deployment of microservices by providing a common framework and set of functionalities essential for microservice operation within the eShare ecosystem. The document processing microservice was then implemented by extending this abstract base class and providing the required functionality for processing documents. This is illustrated in the class diagram in Figure 5.4.

This approach abstracts the details of the Kafka communication from the microservice developer, allowing them to focus on the actual document processing logic. The abstract base class handles the Kafka communication, including message serialization and deserialization, and provides a simple interface for the microservice developer to implement the actual processing logic. This also mitigates the risk of vendor lock-in, as the underlying message bus implementation can be easily swapped out for another one if needed.

Modifications to eShare

The next step was to modify and refactor eShare server's document processing capabilities to support the new event-driven approach. This involved implementing event-driven handling of document processing requests, where eShare server publishes messages to a message bus whenever a document processing is requested and subscribes to another message bus to receive the results of the processing. This system is designed to be agnostic to the actual processing of the documents, which allows for easier microservices communication and integration in a distributed environment.

This setup being agnostic to the document processing implementation also enables eShare to operate in two modes based on a configuration parameter. If, for whatever reason, the document processing microservice is not desired, eShare can be configured to process documents locally, as it did before the containerization process. This is achieved by creating two different message bus implementations in eShare that conform to the same interface, but operate differently:

- A message bus that communicates with a Kafka broker, which is used for microservice processing.
- A message bus that is implemented using C# code and operates locally, in a similar manner as before the containerization process.

The appropriate message bus implementation to use is then determined at runtime based on the server configuration. This way, the usage of the document processing microservice is something that can be opted-in by the administrator of that particular eShare server instance, rather than being forced upon them.

Encountered challenges

One of the main challenges encountered during the implementation process was the lack of support for the eShare's PDF library on Linux. The PDF library used by eShare is only supported on Windows and macOS, which meant that the document processing microservice had to be run in a Windows container. This is not ideal for a cloud-native environment, as it requires the host OS to be Windows, which limits the flexibility of the solution. However, for the purposes of this thesis and case study, using Windows containers was deemed acceptable.

The optimal path forward involves transitioning to Linux containers and adopting a cross-platform PDF library, which would enable a more cloud-native deployment. The use of Windows containers is seen as a temporary solution to demonstrate the feasibility of the containerization process and to provide a proof of concept for future development.

5.4.2 Containerization solution

Microservice application architecture

Building the document processing microservice involved creating a generic abstract class, `KafkaWorkerService<TRequest, TResponse>`, which serves as the foundation for Kafka-driven microservices within the eShare ecosystem. This abstract class handles the Kafka communication and provides a framework for implementing the actual processing logic. The microservice developer is required to extend this class and implement a `HandleMessage` method that receives the incoming message of type `TRequest` as input and returns a response of type `TResponse` as output. The input and output data are passed as JSON messages via Kafka, which allows for easy serialization and deserialization of the data.

The `KafkaWorkerService` abstract class itself is essentially an ASP.NET hosted

service [35] that runs in the background and listens for messages on a Kafka topic. When a message is received, the `HandleMessage` method is called, and the microservice processes the message and returns a response.

The document processing microservice was implemented by creating a `DocumentWorkerService` class that extends the `KafkaWorkerService` class and provides an implementation of the `HandleMessage` method. The class diagram in Figure 5.4 illustrates the relationship between the `KafkaWorkerService` abstract class, the `DocumentWorkerService` class, and any potential future microservices that might be implemented using the same framework.

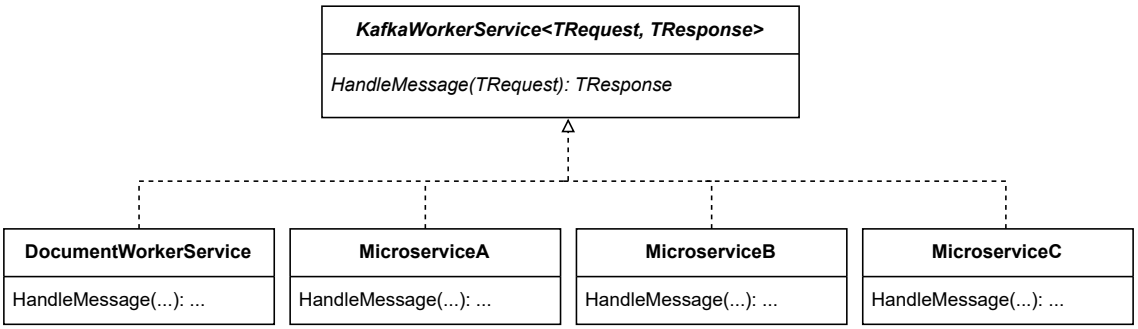

Figure 5.4: Class diagram illustrating the relationship between the KafkaWorkerService abstract class and the microservice implementations.

Kafka is not well-suited for transferring large binary files, such as PDF documents, due to its message size limitations and performance implications. Thus, a reference to the document file path is passed via Kafka, and the document itself is transferred using a shared filesystem – not via the Kafka pipeline. In the future, a separate object storage service, such as *Amazon S3* or *Azure Blob Storage*, could perhaps be used for transferring and sharing the actual document files for a more robust and scalable solution. However, implementing such a solution was deemed outside the scope of this case study but is an important consideration for future development.

Changes to eShare

To accommodate the new document processing microservice, eShare server was modified to support the new event-driven communication model. In this new design, eShare server publishes messages to a message bus whenever a document processing is requested and subscribes to another message bus to receive the results of the processing. There are currently two message bus implementations in eShare: one for local document processing and one for microservice processing. The appropriate implementation is determined at runtime based on the server configuration, as described earlier in Section 5.4.1.

As illustrated in the class diagram in Figure 5.5, both of the message bus implementations are based on the same `IMessageBus` interface, which provides a common interface for publishing and subscribing to messages. The local message bus implementation, `MessageBus`, is a simple in-memory message bus that is used for processing documents locally, while the Kafka message bus implementation, `KafkaMessageBus`, is used for microservice processing. The Kafka message bus implementation uses the Confluent Kafka .NET client library for communicating with the Kafka broker and an internal JSON library for message serialization and deserialization.

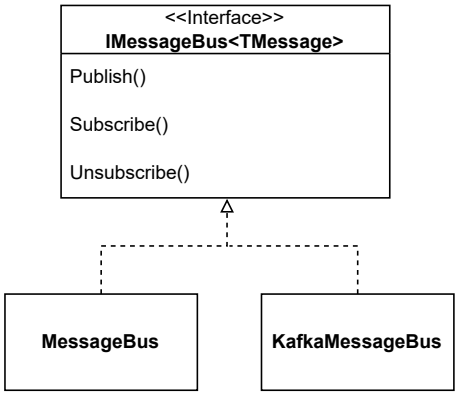

Figure 5.5: Class diagram of the `IMessageBus` interface and the `KafkaMessageBus` and `MessageBus` implementations.

Docker

Containerizing the document processing microservice involved creating a Dockerfile that specifies the container image's configuration and dependencies. The Dockerfile for the document processing microservice is based on the official `mcr.microsoft.com/dotnet/aspnet:8.0-windowsservercore-ltsc2019` and `mcr.microsoft.com/dotnet/sdk:8.0-windowsservercore-ltsc2019` images, which provide the base images for running and building .NET applications on Windows. The .NET SDK image is used for building the application, whereas the .NET ASP.NET image is used for running the application. The Dockerfile used is shown in Listing 5.1.

Since the resulting image is a Windows container, it is worth noting that the host OS must also be Windows and Docker must be configured to use Windows containers in order to run the container. This is a limitation of the current implementation, as it restricts the deployment options for the microservice. However, as mentioned earlier, this is seen as a temporary solution, and the plan is to transition to Linux containers in the future.

```
1 FROM mcr.microsoft.com/dotnet/aspnet:8.0-windowsservercore-
    ltsc2019 AS base
2 WORKDIR /app
3
4 FROM mcr.microsoft.com/dotnet/sdk:8.0-windowsservercore-ltsc2019
    AS build
5 WORKDIR /src
6
7 # Copy PdfWorkerService project
8 COPY ["./DocumentWorkerService", "./DocumentWorkerService"]
9
10 # Copy dependencies
11 COPY ["./KafkaWorkerService", "./KafkaWorkerService"]
12 COPY ["./Dto", "./Dto"]
13 COPY ["./Json", "./Json"]
14 COPY ["./VersionInfo.cs", "./VersionInfo.cs"]
15
16 # Restore and build
17 WORKDIR "/src/DocumentWorkerService"
```

```
18 RUN dotnet restore "./DocumentWorkerService.csproj"
19 RUN dotnet build "./DocumentWorkerService.csproj" -c Release -o /
     app/build /p:IsDockerBuild=true
20
21 # Publish
22 FROM build AS publish
23 RUN dotnet publish "./DocumentWorkerService.csproj" -c Release -o
      /app/publish /p:UseAppHost=false /p:IsDockerBuild=true
24
25 FROM base AS final
26 WORKDIR /app
27 COPY --from=publish /app/publish .
28 ENTRYPOINT ["dotnet", "DocumentWorkerService.dll"]
```

Listing 5.1: Dockerfile for the document processing microservice.

5.4.3 Remarks and future work

The implementation of the containerized document processing microservice was successful and operational for the most part. The microservice can process documents in a containerized environment and communicate with the eShare server via a Kafka message broker. However, several areas could and should be improved upon before the solution can be considered production-ready:

Kafka authentication and authorization : The current implementation relies on the eShare server instances directly connecting to the Kafka broker using a single shared API key. Before moving to a production environment, it would be essential to take into use a more robust and granular authentication and authorization mechanism for Kafka. It should be ensured that only authorized parties can publish and subscribe to the relevant Kafka topics and that sensitive data is protected. This could perhaps be achieved by a more granular API key provisioning or by routing the Kafka traffic through a secure gateway instead of directly connecting to the broker.

Cross-platform compatibility : The current implementation is limited to Win-

dows containers due to the dependency on the Pdfium.Net SDK for PDF processing. Migrating to a cross-platform library for PDF manipulation would enable the use of Linux containers, which are more suitable for cloud environments.

Auto-scaling and orchestration : The current implementation does not support auto-scaling of the document processing microservice. Implementing automatic scaling of the microservice based on the load would be beneficial for handling varying workloads efficiently and reducing costs when the microservice is underutilized. This kind of functionality could be, for example, achieved with Kubernetes [36] or Azure Container Apps [37].

Object storage service : Currently, the document files are transferred to the container via a shared filesystem, which might not be the easiest to configure and maintain. Using an object storage service instead, such as Amazon S3 or Azure Blob Storage could perhaps be a subject for further research and development.

6 Evaluation of eShare's containerization solution

This chapter presents the evaluation of the proposed containerization solution for eShare which was described in Chapter 5. This chapter will also answer RQ2 (*Which functionalities in an application like eShare are most suitable for containerization?*) and further elaborate and expand on the answer of RQ1 (*What benefits can be gained by utilizing container technologies with an application like eShare? How about the drawbacks?*).

The evaluation is done by comparing the proposed solution with the current unmodified solution of eShare. The comparison is done against the criteria that were discovered and identified as being potentially beneficial or disadvantageous for eShare and applications like it in Chapter 4. The results of the evaluation are then analyzed and conclusions based on them are presented.

6.1 Evaluation criteria

The most prevalent benefits and drawbacks associated with containerization identified in Chapter 4 are used as criteria to evaluate the proposed containerization solution for eShare. The benefits and drawbacks are as follows:

- **Overhead**: The amount of overhead introduced by the new proposed solu-

tion is evaluated based on the time taken to process documents in the new architecture compared to the unmodified architecture.

- **Scalability and performance**: The scalability and performance of the proposed solution is evaluated based on the time taken to open a document in the eShare client application when the number of users is increased.

- **Complexity**: The complexity of the solution is evaluated based on my experience and subjective opinion on how complex the solution is to implement and maintain.

6.2 Test setup

The evaluation tests were conducted on a Cadmatic company-issued developer laptop running Windows 10. The laptop has an Intel Core i7-9850H CPU, 32 GB of RAM, and a 1 TB SSD. The tests used a debug build of the eShare server application (version 2024T1) whereas the document processing microservice was built as a release build that was run in a Docker container. The Kafka message broker used in the tests was running in the cloud using the Confluent Cloud service.

It is important to note that the results of the tests are not generalizable to all systems and environments. The results would, most probably, vary between different systems and environments. For example, running the tests on a cluster of powerful servers in the cloud would probably yield different results compared to running the tests on a developer laptop. It is also important to note that the test environment was not a controlled environment. Other applications and background processes running on the test machine could have affected the available resources and thus the results of the tests. The tests are not meant to be exhaustive but rather to give a general idea of the impact of the containerization solution on eShare.

6.3 Overhead

To evaluate the overhead of the proposed containerization solution, the time taken to process documents in eShare as a single user was measured using three different configurations:

- **Config 1**: The current unmodified architecture of eShare.
- **Config 2**: The new proposed local message bus architecture without the use of document processing microservice.
- **Config 3**: The new proposed remote Kafka message bus architecture with a single document processing microservice instance.

The purpose of Config 2 is to evaluate the overhead introduced solely by the new event-driven architecture without the additional overhead introduced by the document processing microservice, whereas the purpose of Config 3 is to evaluate the overhead introduced by the new event-driven architecture together with the document processing microservice. Logically, the assumption is that the overhead introduced by Config 3 should be higher than the overhead introduced by Config 2 since running the document processing microservice in a container should introduce additional overhead.

The measurements were done by processing a set of 9 different documents (D1-D9) in the eShare server application and measuring the time it took to process each document. The measurements were repeated 20 times for each document via an automated script. The median time taken to process each document, the standard deviation of the measurements, and the overhead introduced by the new configurations compared to the current unmodified configuration are presented in Table 6.1. In addition to this, the summary of the measurements is presented as box plots in Figure 6.1.

The overhead introduced by the new event-driven message bus architecture (Config 2) and the new event-driven message bus architecture together with microservice processing (Config 3) was calculated by comparing the median time taken to process a document with the current unmodified architecture (Config 1).

Document	Size (KB)	Config 1		Config 2			Config 3		
		Median (ms)	SD	Median (ms)	SD	Overhead	Median (ms)	SD	Overhead
D1	4	1345	96	1374	192	2%	1901	173	41%
D2	7	1063	95	1087	68	2%	1722	109	61%
D3	20	1132	74	1159	98	2%	1818	133	60%
D4	43	1133	67	1180	133	4%	1756	77	54%
D5	49	1112	68	1126	90	1%	1737	133	56%
D6	1544	1923	418	2079	104	8%	2540	109	32%
D7	1652	1735	213	2951	411	70%	3224	234	85%
D8	2062	7915	289	14941	1122	88%	11075	1133	39%
D9	5423	9644	889	17343	2590	79%	13578	1848	40%

Table 6.1: Table displaying the median time taken to process a document with the different configurations, the standard deviation of the measurements, and the overhead introduced by the new configurations compared to the current unmodified configuration. The overhead is reported as the percentage increase of the median time compared to the unmodified configuration (Config 1).

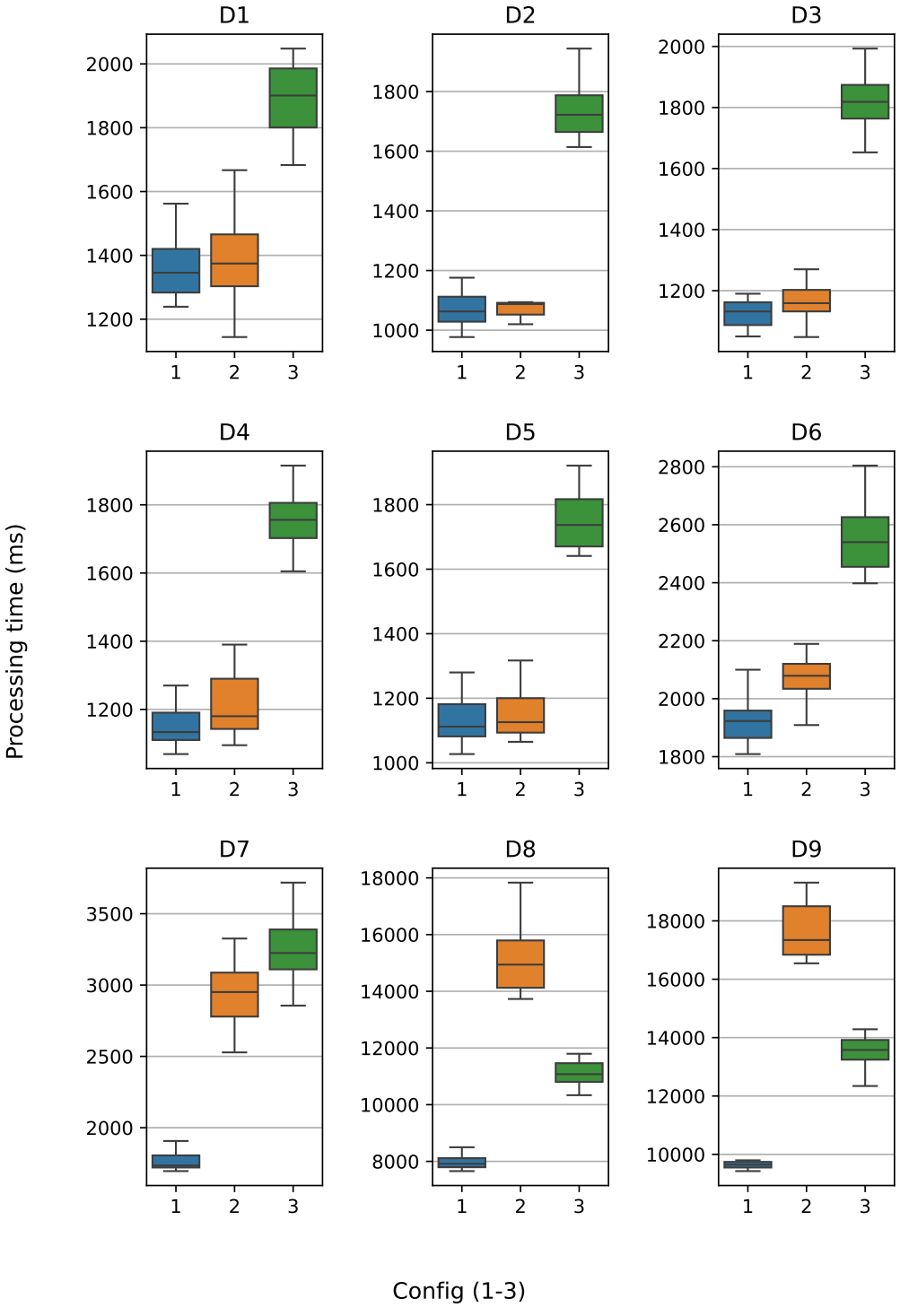

Figure 6.1: Box plots visualizing the overhead measurements of the different configurations when processing various documents (D1-D9). Outliers are omitted from the plots for clarity.

The results seen in Table 6.1 and Figure 6.1 indicate that the event-driven message bus architecture (Config 2) introduces around 0-4% overhead compared to the current unmodified architecture of eShare with small documents (D1-D5). However, with larger documents (D6-D9), the overhead introduced by the event-driven architecture is 70-80%. This is a significant increase compared to the overhead introduced by the event-driven architecture with smaller documents and the reason for this is not immediately clear.

The overhead of Config 3, on the other hand, is around 30-60% compared to the current unmodified architecture of eShare. This is, for the most part, expected. However, the overhead introduced by Config 3 is smaller for D8 and D9 compared to Config 2 which is interesting.

6.4 Scalability and performance

Due to time constraints, the scalability and performance tests were not implemented. Instead, this section only focuses on describing how the tests and evaluation would be done if time allowed.

Similarly to the overhead tests, the scalability and performance tests would involve measuring the time taken to process documents in eShare with different configurations. The difference would be that concurrent document processing requests would be introduced to simulate a higher number of users. The following two configurations would be used to perform the tests:

- **Config 1**: The current unmodified architecture of eShare.
- **Config 2**: EShare with multiple instances of the proposed containerized document processing microservice. Ideally, the document processing microservice would be deployed in the cloud on a more powerful infrastructure than the one used in Config 1 to simulate a realistic production environment.

This time, instead of measuring the overhead introduced by the new configuration, the scalability and performance tests would focus on measuring the time taken to process documents when the number of users is increased. The assumption and expected result would be that Config 1 would be bottlenecked by the inferior hardware and lack of parallel processing capabilities compared to Config 2.

To further evaluate the scalability and performance of the proposed solution, the number of concurrent document processing requests and microservice instances could be varied to gain more insight into the scalability and performance of the containerized solution.

6.5 Complexity

The complexity of a given software architecture and configuration can be difficult to measure objectively. In this case, the complexity of the proposed containerization solution was evaluated based on how I perceived the complexity of the solution and how complex it was to build and implement the proposed containerization solution that was presented in Chapter 5.

In terms of system architecture, sources of complexity in the containerized solution include the need to manage multiple separate components and services such as the Kafka broker, the document processing microservice, and the eShare servers. This also introduces complexity in terms of security, monitoring, and logging. The increased number of components and services in the system not only increases the attack surface but also makes it more complex to manage security policies across multiple components. Monitoring and logging also become more complex with multiple components, requiring a robust logging and monitoring solution that can aggregate logs from multiple sources and provide a coherent view of the system. In terms of development and maintenance, the containerized solution can be more complex and time-consuming to develop and maintain. For example, mocking and simulating a

microservice can be intricate and require additional effort compared to developing and testing a monolithic application.

On the other hand, the containerized solution also simplifies certain aspects of the system. For example, due to the isolated nature of containers and microservices, each microservice can be developed, deployed, and scaled independently. This simplifies individual service lifecycles and makes it easier for a developer to understand, work, and focus on a single service. Managing errors and faults is also simplified due to the isolated nature of microservices. A failure in a microservice is less likely to affect other parts of the system, simplifying fault recovery processes.

6.6 Evaluation

The overhead evaluation tests revealed that the event-driven message bus architecture (Config 2) seems to introduce around 0-4% overhead compared to the current unmodified architecture (Config 1) of eShare with small documents while with larger documents (D7, D8, D9), the overhead was around 70-80%. This radical increase in overhead with larger documents is unexpected and the reason for this is not immediately clear. It is possible that there are flaws in the implementation of the solution or that the test setup is otherwise flawed.

Since Config 1 and Config 2 only differ in the way how document processing requests are dispatched, the overhead introduced by the event-driven architecture should be similar regardless of the document size. In Config 1, whenever a document is processed, the executable responsible for processing the document is called directly from the eShare server application. In Config 2, the document is processed by sending a document processing request to an in-memory message bus that has a listener attached to it which then calls the same executable that is called directly in Config 1. And as previously mentioned, the binary document files are not sent over the message bus but only the metadata of the document. Instead, a shared

file system is used to transfer the files. This would suggest that the reason for the increased overhead with larger documents would be caused by a flaw in the way how the implementation of the in-memory message bus or the listener that processes the document processing requests handles the input and output of the document processing executable. Further investigation and a more granular approach to testing and benchmarking are needed to determine the exact cause of the increased overhead with larger documents. However, due to time constraints, this is unfortunately not possible.

On the other hand, Config 3 of the overhead tests revealed that the remote Kafka message bus together with the document processing microservice introduces around 30-60% overhead compared to Config 1. This is expected and seems reasonable since the Kafka broker used in the tests is running in the cloud and the document processing microservice is running in a Docker container which introduces additional overhead compared to processing documents directly in the eShare server application. However, the overhead with D8 and D9 is smaller compared to Config 2 which is interesting. This is likely related to the unreasonably high overhead introduced by Config 2 with larger documents and to uncover the reason for this, further investigation is needed.

As for the complexity of the proposed containerized solution, it is clear that the new architecture introduces some complexity to the system while also simplifying certain aspects of it. The increased complexity is mainly due to the need to manage multiple components and services such as the Kafka broker, the document processing microservice, and the eShare servers. The increased number of components and services in the system makes it harder to secure, monitor, and log the system as a whole. However, the isolated nature of containers and microservices also simplifies certain aspects of the system. Each microservice is highly independent and decoupled from other parts of the system, making it easier to develop, deploy, and scale

individual services.

Overall, the proposed containerization solution for eShare seems to introduce some overhead compared to the current unmodified architecture of eShare. Apart from the anomalies with the overhead introduced by the event-driven architecture with larger documents, the overhead introduced by the new architecture seems reasonable and expected. The increased complexity of the new architecture is a trade-off for the benefits it brings in terms of scalability, performance, and maintainability. The increased complexity should be something that can be managed and tolerated with proper tools, processes, and practices given the benefits of the new architecture. Due to time constraints, the scalability and performance tests were not implemented and thus the scalability and performance of the proposed solution remain untested. Further investigation and testing are needed to evaluate the scalability and performance of the proposed containerization solution for eShare.

In terms of RQ1, the benefits of utilizing container technologies with an application like eShare are mainly related to scalability, performance, and maintainability. The scalability and performance aspects remain untested but the proposed containerization solution should, in theory, improve the scalability and performance of eShare. The maintainability of the system, on the other hand, is something that will only be revealed over time as the system is developed and maintained. However, in theory, the isolated nature of containers and microservices should simplify the development and maintenance of the system. The drawbacks of utilizing container technologies with an application like eShare are mainly related to the increased complexity of the system and the overhead introduced by the additional layer of virtualization of Docker containers.

In terms of RQ2, the functionalities in an application like eShare that are most suitable for containerization are functionalities that are decoupled and easy to isolate from the rest of the system. For example, the document processing functionality in

eShare was a good candidate for containerization since it is a separate and highly independent part of the system.

7 Summary and conclusion

The purpose of this thesis was to explore and investigate containerization and container technologies in the context of web applications. Utilizing container technologies as a solution for various challenges in software development and deployment has been a popular topic in recent years. Containers encapsulate an application, its dependencies, and its configuration into self-contained units that can be easily deployed and managed while also ensuring that the application runs consistently and predictably across different environments. As a result, running applications in containers abstracts the underlying infrastructure and enables developers to focus on writing code and building applications rather than managing the infrastructure.

The thesis began with an introduction to the topic and presented the research questions that guided the research. The research questions were as follows:

RQ1: What benefits can be gained by utilizing container technologies with an application like eShare? How about the drawbacks?

RQ2: Which functionalities in an application like eShare are most suitable for containerization?

RQ3: What would be a sensible containerization solution for an application like eShare?

Then in the next chapter, the thesis introduced the eShare application, which was the target of the case study in this thesis. This chapter provided an overview

of the features and use cases of eShare and its role as an information management tool for engineering projects in marine and process industries.

The third chapter provided a background and overview of container technologies, focusing on Docker containers. The chapter also provided a practical overview and guide on how to configure, operate, and manage Docker containers. The chapter introduced the basic concepts and terminology of containerization, such as images, containers, registries, and volumes.

The fourth chapter provided an answer for RQ1 by conducting a literature review and analysis of the benefits and drawbacks of containerization in the context of software development and deployment. The literature review extracted benefits and drawbacks other researchers and practitioners have reported in academic papers and articles. The chapter then presented and discussed the benefits and drawbacks of containerization and how they would affect web applications like eShare. The answer for RQ1 was that the most prevalent benefits of containerization were discovered to be performance and scalability, while the prevalent drawbacks were increased complexity and performance overhead.

After this, the fifth chapter continued by building on the results of the literature review and presented a case study on eShare to answer RQ3. The chapter presented an overview of the server-side architecture of eShare and identified the current operational bottlenecks in the application. The chapter then presented a plan and implementation of a partial containerization solution of eShare, focusing on extracting eShare's document processing functionality into a separate microservice running in a Docker container. The chapter concluded by presenting remarks and limitations of the containerization solution and ideas for future development.

Finally, to answer RQ2, the sixth chapter evaluated the proposed containerization solution of eShare against the benefits and drawbacks of containerization reported in the literature review. The evaluation involved comparing the container-

ized document processing microservice with the non-containerized monolithic eShare application in terms of overhead, performance, scalability and complexity. The overhead evaluation tests revealed, as expected, that the containerized microservice had a higher overhead compared to the non-containerized monolithic application. However, due to time constraints, the performance and scalability evaluations were not fully conducted. The evaluation regarding complexity was based on my personal experience and observations during the implementation of the containerization solution and the conclusion was that while the containerization solution added some overall complexity, it also reduced complexity in some areas. The answer to RQ2 was that the most suitable functionalities for containerization in monolithic web applications like eShare are those that have distinct and clearly defined responsibilities in the system and can be easily extracted into separate independent microservices.

In conclusion, the thesis has provided an overview of container technologies and their benefits and drawbacks in the context of software development and deployment. The case study on eShare has demonstrated how containerization can be applied to web applications to potentially improve performance, scalability, and maintainability. The thesis has also identified areas for future research and development, such as conducting more comprehensive performance and scalability evaluations. Overall, the thesis has contributed to the understanding of containerization and its potential applications in web application development.

References

[1] A. Parrott and L. Warshaw, "Industry 4.0 and the digital twin", *Deloitte Insights*, [Online]. Available: https://www2.deloitte.com/us/en/insights/focus/industry-4-0/digital-twin-technology-smart-factory.html (visited on 01/14/2023).

[2] Cadmatic, "Project administration", *CADMATIC Product Documentation*, [Online]. Available: https://docs.cadmatic.com/eshare/Content/Administration/ProjectAdministration.htm (visited on 08/28/2023).

[3] Cadmatic, "Documents view", *CADMATIC Product Documentation*, [Online]. Available: https://docs.cadmatic.com/eshare/Content/Navigation/Introduction_Documents.htm (visited on 01/16/2023).

[4] Cadmatic, "Point clouds", *CADMATIC Product Documentation*, [Online]. Available: https://docs.cadmatic.com/eshare/Content/3DViewer/Point_Clouds.htm (visited on 01/16/2023).

[5] Cadmatic, "Map", *CADMATIC Product Documentation*, [Online]. Available: https://docs.cadmatic.com/eshare/Content/Administration/ProjectAdministration_Map.htm (visited on 01/16/2023).

[6] Cadmatic, "Search view", *CADMATIC Product Documentation*, [Online]. Available: https://docs.cadmatic.com/eshare/Content/Navigation/Introduction_Search.htm (visited on 01/16/2023).

[7] Oracle, *What is SaaS (Software as a Service)?*, [Online]. Available: https://www.oracle.com/applications/what-is-saas/ (visited on 09/11/2023).

[8] OpenBSD, *chroot(2) - OpenBSD manual pages*. [Online]. Available: https://man.openbsd.org/chroot.2 (visited on 03/08/2023).

[9] S. C. Delgado, *FreeBSD Handbook*, ch. 16. Jails. [Online]. Available: https://docs.freebsd.org/en/books/handbook/jails/ (visited on 03/08/2023).

[10] T. Hildred, "The history of containers", *Red Hat*, Aug. 2015. [Online]. Available: https://www.redhat.com/en/blog/history-containers (visited on 03/08/2023).

[11] Linux man-pages, *cgroups(7) — Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 03/08/2023).

[12] Linux Containers, "Lxc - linux containers", *GitHub repository*, [Online]. Available: https://github.com/lxc/lxc (visited on 03/08/2023).

[13] K. Chandrakant, "Evolution of docker from linux containers", *Baeldung on Linux*, Nov. 2022. [Online]. Available: https://www.baeldung.com/linux/docker-containers-evolution (visited on 03/08/2023).

[14] IBM Cloud Team, "Containers vs. virtual machines (vms): What's the difference?", *IBM*, Apr. 2021. [Online]. Available: https://www.ibm.com/cloud/blog/containers-vs-vms (visited on 03/14/2023).

[15] C. Anderson, "Docker [software engineering]", eng, *IEEE software*, vol. 32, no. 3, pp. 102–c3, 2015, ISSN: 0740-7459.

[16] S. Hykes, "The future of linux containers", *dotcloudtv*, Mar. 2013. [Online]. Available: https://www.youtube.com/watch?v=wW9CAH9nSLs (visited on 04/26/2023).

[17] Docker docs, "Docker overview", *Docker docs*, [Online]. Available: `https://docs.docker.com/get-started/overview/` (visited on 05/03/2023).

[18] Docker docs, "Volumes", *Docker docs*, [Online]. Available: `https://docs.docker.com/storage/volumes/` (visited on 05/25/2023).

[19] Docker docs, "Networking overview", *Docker docs*, [Online]. Available: `https://docs.docker.com/network/` (visited on 05/25/2023).

[20] Docker docs, "Dockerfile reference", *Docker docs*, [Online]. Available: `https://docs.docker.com/engine/reference/builder/` (visited on 05/17/2023).

[21] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned", in *2018 IEEE 11th international conference on cloud computing (CLOUD)*, IEEE, 2018, pp. 970–973.

[22] H. Knoche and H. Eichelberger, "Using the raspberry pi and docker for replicable performance experiments: Experience paper", in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 305–316.

[23] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!", in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 11–16.

[24] D. Morris, S. Voutsinas, N. C. Hambly, and R. G. Mann, "Use of docker for deployment and testing of astronomy software", *Astronomy and computing*, vol. 20, pp. 105–119, 2017.

[25] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, "The impact of docker containers on the performance of genomic pipelines", *PeerJ*, vol. 3, e1273, 2015.

[26] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three brazilian government institutions", in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, 2018, pp. 32–41.

[27] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead", *IEEE access*, vol. 7, pp. 52 976–52 996, 2019.

[28] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud", in *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018, pp. 1–8.

[29] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain", *Ieee Software*, vol. 35, no. 3, pp. 50–55, 2018.

[30] H. Zhu and I. Bayley, "If docker is the answer, what is the question?", in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, pp. 152–163.

[31] I. Mavridis and H. Karatza, "Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing", *Future Generation Computer Systems*, vol. 94, pp. 674–696, 2019.

[32] H.-C. Chang, B.-J. Qiu, C.-H. Chiu, J.-C. Chen, F. J. Lin, D. De La Bastida, and B.-S. P. Lin, "Performance evaluation of open5gcore over kvm and docker by using open5gmtc", in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2018, pp. 1–6.

[33] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?", in *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, IEEE, 2015, pp. 507–510.

[34] Apache Software Foundation, "Use cases", *Apache Kafka Use Cases*, [Online]. Available: `https://kafka.apache.org/uses` (visited on 03/18/2024).

[35] Microsoft learn, "Background tasks with hosted services in asp.net core", *Microsoft learn*, [Online]. Available: `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-8.0&tabs=visual-studio` (visited on 04/13/2024).

[36] Kubernetes, "Horizontal pod autoscaling", *Kubernetes*, [Online]. Available: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/` (visited on 04/13/2024).

[37] Microsoft Azure, "Azure container apps", *Microsoft Azure*, [Online]. Available: `https://azure.microsoft.com/en-us/products/container-apps` (visited on 04/13/2024).