UNIVERSITY OF LJUBLJANA
SCHOOL OF ECONOMICS AND BUSINESS

MASTER'S THESIS

THE INTRODUCTION OF SOFTWARE CONTAINERS AT A SELECTED COMPANY

Ljubljana, December 2023 Atif Halitović

AUTHORSHIP STATEMENT

The undersigned Atif Halitović, a student at the University of Ljubljana, School of Economics and Business, (hereafter: SEB LU), author of this written final work of studies with the title The introduction of software containers at a selected company, prepared under supervision of Miro Gradišar, PhD

DECLARE

1. this written final work of studies to be based on the results of my own research;

2. the printed form of this written final work of studies to be identical to its electronic form;

3. the text of this written final work of studies to be language-edited and technically in adherence with the SEB LU's Technical Guidelines for Written Works, which means that I cited and / or quoted works and opinions of other authors in this written final work of studies in accordance with the SEB LU's Technical Guidelines for Written Works;

4. to be aware of the fact that plagiarism (in written or graphical form) is a criminal offence and can be prosecuted in accordance with the Criminal Code of the Republic of Slovenia;

5. to be aware of the consequences a proven plagiarism charge based on the this written final work could have for my status at the SEB LU in accordance with the relevant SEB LU Rules;

6. to have obtained all the necessary permits to use the data and works of other authors which are (in written or graphical form) referred to in this written final work of studies and to have clearly marked them;

7. to have acted in accordance with ethical principles during the preparation of this written final work of studies and to have, where necessary, obtained permission of the Ethics Committee;

8. my consent to use the electronic form of this written final work of studies for the detection of content similarity with other written works, using similarity detection software that is connected with the SEB LU Study Information System;

9. to transfer to the University of Ljubljana free of charge, non-exclusively, geographically and time-wise unlimited the right of saving this written final work of studies in the electronic form, the right of its reproduction, as well as the right of making this written final work of studies available to the public on the World Wide Web via the Repository of the University of Ljubljana;

10. my consent to publication of my personal data that are included in this written final work of studies and in this declaration, when this written final work of studies is published.

11. that I have verified the authenticity of the information derived from the records using artificial intelligence tools.

Ljubljana, _____ Author's signature: _____

(Month in words / Day / Year)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

ACI – Azure Container Instance

AKS – Azure Kubernetes Service

AWS – Amazon Web Services

CBA – Cost-Benefit Analysis

CD – Continuous Delivery

CI – Continuous Integration

CLI – Command Line Interface

CMD – Command-Prompt

CNCF – Cloud Native Computing Foundation

CRM – Customer Relationship Management

CT – Continuous Testing

DevOps – Development Operations

ECS – Elastic Container Service

EKS – Elastic Kubernetes Services

IDC – International Data Corporation

IT – Information Technology

LXC – Linux Containers

NPV – Net Present Value

OS – Operating System

QA – Quality Assurance

ROI – Return on Investment

SDLC – Software Development Life Cycle

VM – Virtual Machine

VS – Visual Studio

1 INTRODUCTION

Automation in technology now plays a significant role in many company activities and has become an integral part of our daily lives. For instance, it helps businesses run more efficiently and increases consumer satisfaction (Najihi et al., 2022). The digitalization of business operations with the goal of automation has potential benefits across all sectors, and the Information Technology (hereafter: IT) industry is especially no exception.

A very complex combination of applications, computer systems and procedures is the foundation of most software companies as one of the biggest parts of the IT sector. The larger the company, the more difficult it is to integrate technologies. Software development life cycle outlines how software products are planned, executed, and packaged for delivery (McCormick, 2021).

The software delivery business has evolved into a very practical profession as a result of constant analysis and modification of our delivery methods. An automated and sophisticated build pipeline that enables developers to enhance feedback at all stages of the software development life cycle has replaced the industry's previous practice of doing software builds on dedicated, individual developer computers. To increase the capability of software delivery in an automated manner, it is important to implement and adopt concept called development operations (hereafter: DevOps) (Azad, 2022).

DevOps concept is based on integration processes and techniques for business software applications to be continuously integrated and delivered, as well as related tooling, in order to reduce IT operating expenses while enhancing software quality, stability, and speed to market. DevOps fundamentally encourages a more in-depth focus on performance and results (Bahadori & Vardanega, 2019).

After successfully following the methodology for developing software products, the software development process in companies continues with software integration and deployment. It simply takes all the things that have been developed and gets them to the right place. It aims to be a system that smoothly distributes all the goods created as a consequence of distinct sprints. DevOps aims to fully automate the procedure and keep track of every single event that occurs during the various stages of the software deployment and integration process. In order to produce software quickly with short delivery cycles, continuous delivery and deployment are the fundamental components of DevOps in a company (Azad, 2022). Once DevOps is properly deployed and integrated inside software companies, the gaps between the operation and development teams are minimized.

Software container technology is a concept that makes these tasks much easier, especially if we consider Continuous Delivery and Continuous Integration as critical components of any software development project today. It is a widely used technology in the cloud computing and IT ecosystems that efficiently separates a single operating system's resources into

separated groups to more fairly balance the competing resource usage demands of isolated groups (Huawei, 2023, p. 295).

Containerization includes combining the program along with all of its dependencies, like the runtime, system utilities, system libraries, and settings of an application, into a compact standalone executable package. Because everything needed to run the program is in the container, it can be transferred from a physical computer to a virtual machine, or on the public or private cloud, basically any environment, without running the danger of becoming tied to the operating system of the actual machine. That is exactly the biggest benefit of the containers, which will be further discussed and compared in the thesis. One of the most popular containers is Docker, which is an open-source platform and containerization-enabling platform for efficiently developing, operating, integrating and deploying applications (Raj & Raman, 2018).

In larger application deployments, several containers may be deployed as one or more container clusters. Such clusters must be controlled by some kind of container management tool or container orchestrator such as Kubernetes. An open-source system automates the scalability, deployment, and administration of containerized applications. For simple administration and discovery, it divides the containers that make up an application into logical parts. High levels of interoperability, self-healing, automated rollouts and rollbacks, and storage orchestration are all features of Kubernetes. In terms of automatically resolving issues, Kubernetes excels. The fact that containers may crash and restart so quickly means we won't even realize when our containers are crashing (Raj & Raman 2018).

The major problem in a selected company lies in the insufficient utilization of container benefits within its IT operations. I will suggest the improvements for key areas where the company's software development process might be improved with containerization by comparing current and proposed ones. I want to draw attention to the issues and difficulties the business is experiencing and make suggestions for changes to enable a quicker, more effective, and more precise development process. The general public, including other scholars and businesses dealing with the same or comparable problems, will find the results interesting as well.

The purpose of the master’s thesis is to contribute to the understanding of the impact of software containers, exploring both the costs incurred by companies and the potential benefits arising from their integration into development processes of current or future applications. The thesis seeks to clarify how software containers contribute to the activities in the company concerning the software development lifecycle and operations.

The goals of the master’s thesis are to through collecting data from different secondary sources to analyze the possibilities of software containers usage. With that information, we will be able to compare the results to ascertain the reasons for introducing them. Moreover, the goals of the thesis extend to collecting data from primary sources, such as interview with

IT engineer, to learn from him directly about the benefits and challenges of using software containers in certain contexts. However, the last goal is to summarize the main benefits and challenges of software containers and to propose actions on how these challenges may be overcome.

The research question of the master's thesis is: "What are the benefits and costs if we introduce software containers to the selected company?".

For the research in thesis, I used a mix of research methods. This includes qualitative methods, such as review of the existing literature, observation and conducting in-depth interviews with IT engineer, alongside quantitative methods. This includes the analysis of company data that relies on assumptions and is conducted after the implementation of software containers. The integration of these methods allows for an analysis of both the empirical and experiential aspects of software container adoption within the selected company. In addition to this, I conducted Cost-Benefit Analysis. By combining the information from these varied methodologies, the research seeks to form a complete picture of the influence of software containers on IT operations.

This master's thesis is structured in three parts. The main goal of the first part is to provide a solid theoretical foundation for software containers. The thesis will systematically highlight key ideas, current trends, and go thoroughly into the details of the software development life cycle through a thorough literature review, which is the foundation of secondary research. The approaches utilized for software delivery, the specifics involved in its management, and a connection between containerization and cloud services are just a few of the important issues that will be covered in this part.

The focus of the second section is the research methodology used for this study. This section will go into more specifics about research methods, with a focus on an interview with the responsible IT engineer. The two objectives of this interview are to understand the operational modifications that followed the deployment of software containers over time, as well as to obtain insight into the justification for their use. The criteria and logic used in choosing academic materials, such as significant textbooks and important scholarly articles in the field, will be covered in this context.

The final and third section dives more deeply into the findings of the study. IT engineer will provide his perspective on their experiences using software containers in the section titled "Interview Analysis." An overview of the present business operational landscape will be provided in the "Analysis of Present Business Processes," which will be followed by recommendations for how software containers may lead in optimization.

2 LITERATURE REVIEW

2.1 Overview of The Software Development Life Cycle

2.1.1 Evolution of Software Development Methods

Software development is the term used to describe a set of computer system tasks involved in developing, deploying, and maintaining software with the following particular system development methodology. For the term software, we can say that it has been there 50 years. It can be defined as a set of instructions or programs that a computer uses to carry out its tasks (IBM, 2023). The early stages of software development were a chaotic process called "code and fix." The system design was impacted by a number of poor decisions, and the software was developed without much preparation. This worked for simple systems, but as systems grew larger and more complex, it became more difficult to add new features and address problems. Up until a methodology was created, this development technique was used (Mishra, 2013).

The most crucial element of software development is the software development life cycle (hereafter: SDLC) (Mishra, 2013). SDLC is a process that may be used to create software in a methodical way and that will increase the likelihood that the project will be finished within the expected time frame while maintaining the level of software product quality. It includes a number of stages, from early-stage software analysis through final software testing and evaluation. It also includes the models and processes used by development teams to create software systems. These methodologies serve as the framework for organizing and managing the whole development process (Khong et al., 2012).

Using any SDLC model is frequently a question of personal preference totally up to the developer or the company. Each SDLC has its own advantages and disadvantages, and some may perform better than others depending on the circumstances. Choosing which model to use to deliver a certain set of features in a given situation then becomes difficult. One life cycle method may, in theory, match a given set of circumstances while another model may appear to fulfill the needs as well (Mishra, 2013). Despite the fact that software development methods vary, the objective of IT companies is still the same - to deliver a high-quality service at the lowest cost while meeting deadlines (Westfall, 2016, p. 161).

It is crucial to ensure that essential and well-tested processes are consistently applied throughout software development to achieve a high-quality outcome. Well-defined models serve as a clear roadmap for development teams. Following these established models and their associated procedures simplifies the process of confirming that critical stages are included, ultimately raising the overall quality of the project (Westfall, 2016, p. 161).

Mishra (2013) points out that breaking down the development effort into distinct stages, each with its own specific set of tasks and including reviews at each stage transition, these models prove invaluable for project managers and teams. They support activity planning and closely monitoring progress. Any of the models typically involves the following steps:

1. Requirements Definition: The initial step in software development is to understand the problem we are aiming to solve. This entails gathering detailed requirements from customers. Often, the software system we are developing is just a subset of a larger system. Therefore, it is essential to identify requirements that ensure the new software can seamlessly integrate with other components of this broader system. These interface-related requirements are critical for achieving interoperability (Galin, 2018, p 636).

2. Designing: Once the problem is clearly defined, we proceed to the designing phase. This is where we outline our solution approach. The design phase elaborates on what the software will look like and how it will function. It encompasses details like the outputs the software will produce, the inputs it will require, and the underlying processes. Furthermore, considerations on data structures, databases, and the software's architecture are deliberated at this stage (Galin, 2018, p. 638).

3. Coding: Armed with a comprehensive design, developers then delve into coding. This phase translates the design blueprint into functional software code. In other words, it is where the plan is turned into a tangible, operational product (Galin, 2018, p. 638).

4. Testing: The primary purpose of testing is quality assurance. Software can have bugs or deviations from the expected behavior, and testing aims to identify these issues. It is crucial to ensure that the developed software aligns with business requirements and meets user expectations. There are two main approaches to testing: manual and automated. In manual testing, testers act as end-users, navigating the software and verifying its functionalities. On the other hand, automated testing leverages tools to execute predefined tests, minimizing the need for human intervention and repetitive tasks, making the process more efficient (Sekgweleo & Iyamu, 2022).

5. Product Deployment and Maintenance: Deployment is a pivotal phase where the developed software is made available to users. This entails installing and configuring the software to ensure its smooth operation in the intended environment. Traditionally, maintenance was seen as an activity that starts after product deployment (Mens & Demeyer, 2008, p. 1). However, its importance cannot be understated. Maintenance can be categorized into three main types (Galin, 2018, p. 639):

a. Corrective: This focuses on fixing software faults identified either by users or developers during regular operations.
b. Adaptive: Here, the software is tweaked using existing features to accommodate new requirements without major changes.

c. Functionality-improvement: This involves making minor additions to enhance the software's overall performance.

According to Khong et al. (2012), most developers today primarily employ two approaches to the SDLC: traditional development and agile development. These methodologies will be delved into in the next section of this thesis.

2.1.2 Software development life cycle methodologies

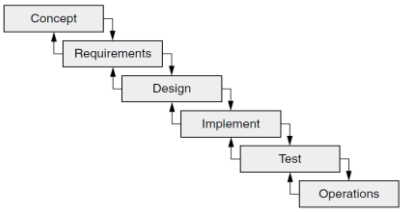
2.1.2.1 Traditional software development

According to Khong et al. (2012), the traditional development methodology in software development relies on a set of preset procedures and ongoing documentation that is created as the work is completed and used to direct subsequent development. The success of a project addressed in this manner depends on having a complete understanding of all requirements before development begins, which makes it challenging to integrate changes as a project moves through its development lifecycle. Nevertheless, this approach also simplifies the task of estimating project expenses, defining a timeline, and distributing resources appropriately.

The most popular model of traditional software development is the waterfall model (Galin 2018, p. 636). The waterfall model has a notable limitation: it struggles with uncertainty and risk. This is coming from its sequential nature, which often means there is minimal feedback from the business side during earlier stages. Consequently, any issues or shortcomings that arise during the planning phase are not easily adjusted later on. This can lead to the delivery of software that does not meet the desired quality standards, highlighting a potential drawback of the waterfall approach in dynamic development environments (Sekgweleo & Iyamu, 2022).

Figure 1 provides a visual representation of the waterfall model. In this depiction, six distinct phases are illustrated. However, it is essential to understand that the actual number of phases can vary based on the specific requirements of the project and the preferences of the organization overseeing it. While many representations of the waterfall model show only a top-to-bottom flow, indicating a linear progression of activities, the example in Figure 1 also incorporates upward arrows. These arrows suggest that some back-and-forth or iterative adjustments are possible within the development activities, highlighting a degree of flexibility in the model (Westfall, 2016, p. 162).

Figure 1: Waterfall model



Source: Westfall (2016, p. 163)

While the waterfall model has faced criticism from many experts who believe it is outdated and not representative of current software development realities, it is important to note that it still has relevance in specific scenarios. Particularly, the waterfall model proves useful for projects where the requirements are clearly defined and unlikely to change during the development process. This structured, sequential approach is especially suitable when transitioning an existing product to a new platform, environment, or language, given that the constraints and expectations are well-understood. Such scenarios underscore the continuing value of the waterfall model in the broader landscape of software development methodologies (Westfall, 2016, p. 163).

2.1.2.2 Agile software development

The growing importance of IT in a turbulent business climate has led to new developments in the end-to-end information technology delivery process. Companies are under increasing pressure to develop software as a key business competence, while also using best practices to ensure timely development without risking business delays. However, the high cost of software development, long supply schedules, and difficulties in implementing requested changes during the development process can lead to lower customer satisfaction. In response to these shortcomings of traditional methodologies, several software developers have proposed radical changes to traditional software development methods (Galin, 2018, p. 653).

In an attempt to address the aforementioned shortcomings of traditional methodologies, a number of software developers proposed changes to traditional software development methods. Since all of these modern methodologies have many characteristics, an organization of their creators known as "the Agile Alliance" was established in February 2001. The alliance was formed with the intention of promoting these approaches (Galin, 2018, p. 654).

Agile development divides the development life cycle into smaller pieces, referred to as "increments" or "iterations," where each of these impacts each of the traditional phases of development, as opposed to a single huge process model that is implemented in traditional SDLC (Khong et al., 2012).

The following four are among the four main agile factors, according to the Agile Manifesto (Agilemanifesto.org, 2001):

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The Agile Alliance (www.agilealliance.com) has also defined the following 12 principles to support the Agile Manifesto (Agilemanifesto.org, 2001):

1. "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and customers/users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity, the art of maximizing the amount of work not done, is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

In essence, any software development method that does not follow the traditional waterfall method and produces components more quickly while using certain approaches to assure quality often qualifies as agile (Bird, 2010, p, 138).

The following software development methodologies incorporate the principles of agile (Galin 2018, p. 654):

- Agile modeling
- SCRUM
- Extreme programming (XP)
- Crystal
- Feature-driven development (FDD)

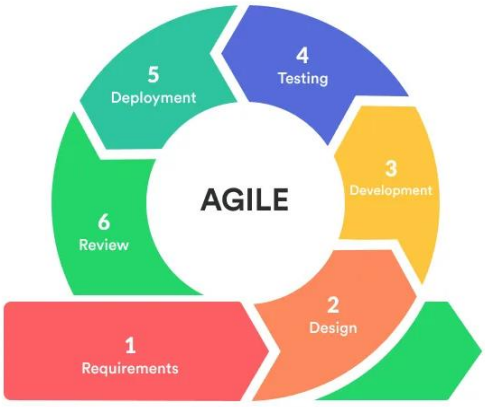
- Dynamic system development method (DSDM)
- Kanban
- Lean software development
- Rapid application development (RAD)

Effective communication between the customer, developer, and product is critical in agile software development. Developers must work closely with customers to understand their needs and create a product that meets their requirements. To ensure that the product meets its specifications, developers must conduct regular reviews and testing. Customers should also be involved in the development process through frequent prototypes and demonstrations of the product to ensure that it meets their intended use and functions properly. This ongoing feedback loop allows for the early detection of problems and ensures the quality of the final product. This regular, effective communication creates a constant feedback loop to ensure that the right product is being developed correctly, to identify issues earlier in the development cycle, and to raise the caliber of the finished product (Galin, 2018, p. 655).

Agile software development methods focus on social solutions and prioritize skills, communication, and community to make projects more effective and agile. Rather than relying on technology-oriented rules, agile methodologies use "light-but-sufficient rules of project behavior" and "human and communication-oriented rules". These solutions are then paired with the appropriate technologies to enable increased efficiencies and facilitate processes such as test automation and continuous deployment (Galin, 2018, p. 656).

Figure 2 offers a graphical depiction of the agile model. This illustration showcases six iterative phases: Requirements, Design, Development, Testing, Deployment, and Review. Each phase is presented in a repeatable cycle, emphasizing the agile model's flexibility and responsiveness. Through this model, it becomes clear how each stage feeds into the next, allowing for continuous improvement and adaptation throughout the development process.

Figure 2: Agile model



Source: Jayathilaka (2020)

2.1.2.3 DevOps integration and its role in Software Development

What is actually the Agile software development methods issue is that they mostly focus on the development side of the software process. In order for a software process to be complete, it requires the combined efforts of both the development and operations teams. Without this collaboration and integration, the process is incomplete (Azad, 2022). The operations teams who are responsible for quality control, packaging, and releasing software products, should work closely with other teams involved in the software development process. In order to improve the process and flow of software delivery, the DevOps movement strives to eliminate bottlenecks between development and operations teams and promote collaboration (Cois et al., 2014).

Software development practices called DevOps combine people, processes, and technology to produce continuous value. Planning and tracking, development, build and testing, delivery, monitoring, and operations make up the method. The unique aspect of DevOps is the collaboration of the development, IT operations, quality engineering, and security teams to improve the speed, quality, and reliability of software releases through a combination of cultural changes, tools, and processes (Jacobs, 2023).

While DevOps is often used in conjunction with agile development methods, it is not inherently tied to the agile methodology. In fact, DevOps can be applied to any development methodology, including agile, traditional, or other approaches. Its focus is on improving the collaboration and integration between development and operations teams, regardless of the specific development methodology being used. It is important to note that DevOps and agile are not mutually exclusive. In fact, when used together, these two methodologies can lead to greater efficiency and more reliable results. DevOps can be seen as an enhancement or extension of the agile methodology, as it adds additional practices and tools to support the integration and collaboration between development and operations teams (Jacobs, 2023).

Agile development at a rapid pace can sometimes lead to gaps between operations and testing teams. This can result in isolated teams, each with different leaders and roles. DevOps bridges these gaps by integrating practices like continuous integration, delivery, testing, deployment, and monitoring. Drawing from the Agile Manifesto's primary principle, continuous delivery emphasizes the importance of continually delivering valuable software to the customer. As Humble and Farley (2010, p. 23) note, the initial software release is merely the start of its delivery journey.

There are many benefits to implementing Continuous Deployment, Continuous Integration/Continuous Delivery and Continuous Testing. These practices allow for errors to be corrected quickly while the context is still fresh in the minds of developers and help to identify and eliminate the underlying causes of these errors. They help to quickly identify and fix errors, automate the testing process, and prioritize test cases (Najihi et al., 2022).

Regarding security, DevOps has gradually expanded to incorporate security measures, leading to the emergence of DevSecOps. While the emphasis on security has been consistent for a long time, its significance has amplified with the rising adoption of DevOps. This shift is largely because product development teams now shoulder greater control over their infrastructure, from provisioning to management. Consequently, these teams must have a thorough understanding of security aspects to ensure they align with the compliance, regulatory, and auditing standards set by both clients and government regulatory bodies. This integration not only enhances the safety of processes and products but also underscores the vital role of security in modern development practices (Jones, 2019).

DevOps teams often rely on several performance metrics, but four stand out as particularly significant (Hall, 2023):

- Time for Code Updates: This refers to the duration from when a code alteration is made to the main branch until it is ready for deployment, after passing all pre-release tests.
- Rate of Post-Production Fixes: This metric calculates the proportion of code updates that need urgent fixes after they have been deployed.
- Deployment Regularity: It is crucial to monitor how often updates are introduced into the final production stage.
- Average Recovery Duration: This measures the average time required to bounce back from minor interruptions or complete system breakdowns, regardless of the root cause.

Each of these metrics plays a pivotal role in evaluating the effectiveness and efficiency of DevOps practices.

2.2 Continuous Integration and Delivery: Core of Modern Software Practices

In this section we will delve deep into the pivotal components shaping today's software development landscape. We will discuss the complexities of continuous integration, delivery, and deployment. Moreover, we will explore the significance of testing, monitoring, and the emerging role of software containers in optimizing these processes. Each element contributes uniquely to the efficient and responsive practices that characterize modern software development.

2.2.1 Continuous Integration

In the software development, there is a shared understanding that tackling and completing complex tasks more frequently is more beneficial than doing them sporadically. This philosophy emphasizes the value of regular integration, advocating for system updates with every minor change. This practice, known as Continuous Integration (hereafter: CI), represents a transformative approach to software development (Humble & Farley, 2010, p. 90).

Without CI, developers spend excess time coordinating to ensure their code aligns correctly. This often leads to delays and potential errors. The absence of CI can also magnify communication barriers, making it challenging for teams to work harmoniously. As the team and codebase sizes increase, these challenges can intensify. Moreover, a lack of CI can create a disconnect between the development team and other organizational units, with the former seeming like it is operating in isolation. Predicting project timeline additionally becomes difficult since merging new modifications without CI can introduce unforeseen complexities (Rehkopf, 2023).

CI plays a pivotal role not just for elite software teams but for the entire organization. By clarifying the process of software creation and delivery, CI fosters a more efficient planning and execution strategy. It champions scalability by streamlining workflows, which in turn expedites project timelines even as the team and codebase expand. Furthermore, CI enhances feedback mechanisms, allowing teams to rapidly iterate on their ideas and address issues. Through features like pull requests, CI improves team communication, providing a platform for collaborative code reviews and refinements. Despite its advantages, the adoption of CI presents challenges, especially for newcomers. The initial setup can be complex, and there is a learning curve associated with mastering CI related tools. Yet, the enduring benefits of CI workflows and improved collaboration make the initial obstacles worth overcoming. Consequently, to reduce potential challenges many modern software projects gives CI top priority right now (Rehkopf, 2023).

Several DevOps tools have been crafted to bridge the gap between development and operations. Notable among them are CI systems like Jenkins and Travis. When equipped with advanced pipelines, these tools effectively connect various deployment and delivery stages (Erich, 2019).

A plenty of third-party tools, including Codeship, Bitbucket Pipelines, SemaphoreCI, CircleCI, Bamboo, and Teamcity, facilitate CI management and setup. Each of these platforms offers comprehensive setup guides and resources. Jenkins stands out as a seasoned CI tool with a robust track record. As an open-source solution, it benefits from community-driven updates and is primarily tailored for on-premise installations (Rehkopf, 2023). After integrating code changes, the next logical step is to prepare them for release into various environments. This progression leads us to the Continuous Delivery.

2.2.2 Continuous Delivery

Continuous Delivery (hereafter: CD) can be viewed as a natural progression CI. While CI focuses on the frequent merging and testing of code changes, CD extends this by ensuring that these changes are automatically prepared for release to a testing or production environment. Essentially, once the build step in CI is complete, CD takes over, automating the next stages up to the point of deployment (Pittet, 2021).

Organizational processes evolve over time. Initially, they might be manual checklists to be performed by hand. Over time, they can be converted into software scripts to ensure they are consistent. This means that if a task has to be done again, someone can just run the script. The real value of such scripts is their reliability, especially when they are used consistently across different settings. For instance, the methods used to deploy code in a test setting should closely resemble those in the live environment. This consistency helps avoid many common problems (Rehkopf, 2023).

One of the main tenets of CD is automation. Instead of using valuable human time on repetitive tasks, CD emphasizes using scripts and tools. Until a task is automated, it can't truly be considered consistently repeatable. The aim is to automate everything, from testing to releasing new features. Version control is as well fundamental to CD. It allows teams to work on shared projects without stepping on each other's toes. Git, for instance, is a popular tool for this. Beyond just code, everything from configurations and scripts to database structures should be under version control. This ensures changes can be tracked and, if needed, undone (Rehkopf, 2023).

Instead of being a last-minute consideration, quality should be a main focus in CD from the beginning. It is about integrating quality checks at every stage, ensuring that the final product meets the highest standards. When planning new features, teams should also think about ways to monitor and test those features. Tasks that are tricky or time-consuming should be tackled early. The longer they are put off, the more they can drain team energy and resources. Addressing these tasks early can lead to significant time savings in the long run. Frequently revisiting these challenges can also spotlight areas for process improvement (Rehkopf, 2023).

Everyone in the organization should be aiming for the best possible product quality. From product managers to the security team, each member should play their part in the release process. Quality Assurance (hereafter: QA) teams, for instance, should test initial versions as thoroughly as the final product. It is all hands-on deck when planning for a product release. CD is not just about improving processes; it is also about measuring outcomes. Metrics like the time from idea to release, user engagement levels, or the frequency of new feature launches can give insights into CD's impact. CD can help set broader organizational metrics, ultimately influencing overall business success (Rehkopf, 2023). In this situation, Continuous Testing is crucial, ensuring that every change is carefully examined before it becomes a part of the final product.

2.2.3 Continuous Testing

Software testing can be broadly categorized into manual and automated processes. In manual testing, individuals or teams actively engage with the software to ensure it functions as intended. On the other hand, automated testing encompasses a variety of tools with

capabilities ranging from basic code validation to replicating comprehensive manual tests (Maynard, 2023).

Many mature software development languages have their own testing tools. Numerous utilities are available to help set up and manage testing suites. Typically, these utilities can be sourced via the project is specific programming language package manager. Beyond just setting up tests, there are tools tailored for test execution and management. Different test runners can be used to extract results from a test suite. It is common to determine the "test coverage" of a project, which refers to the extent of the code that has been tested. To assess this, code coverage tools are utilized (Maynard, 2023).

In the agile software methodology, testing is a shared responsibility that involves the entire team, a contrast to traditional methodologies like the waterfall approach where the responsibility is primarily on the QA team. In the agile framework, everyone plays a role: developers validate user stories, testers ensure these stories adhere to functional specifications, and business stakeholders, along with project managers, assess the software from the user perspective. This inclusive approach to testing ensures swift feedback within the compact timelines typical of agile, as each brief iteration aims to produce software ready for deployment (Najihi et al., 2022).

Continuous Testing (hereafter: CT) sets an ambitious objective: to autonomously execute the complete test suite with every software build. It leans heavily on automation, suggesting every test case should harness this capability. The rationale is that modern technology can replicate and handle any repetitive task, implying a large portion of manual testing can be automated. The software delivery mechanism should be adept at running the full array of tests automatically for every software iteration, making the way for the expedited release of top-tier software. Testing has shifted to the beginning of the development cycle, reflecting actual production conditions and emphasizing assessments in that setting (Najihi et al., 2022). This question drives the transition from merely ensuring that software is always ready for release, to actually releasing it continuously - introducing us to Continuous Deployment.

2.2.4 Continuous Deployment

Altough Continuous Deployment represents a significant advancement in software development practices, it is not a universally applicable solution. It might not be possible for companies with strict compliance requirements or those that must support each version of their software to deploy feature right away. However, its advantages are numerous (Humble & Farley, 2010, p. 299).

Contrary to popular belief that it is high-risk, releasing updates more frequently can actually reduce associated risks. With each minor change being released independently, the risk is limited to that specific update. This systematic and incremental approach makes it easier to troubleshoot issues and ensures minimal disruptions. To uphold this methodology's efficacy,

continuous deployment mandates automation across all stages, from build and deploy to testing and releasing. Such a framework leans heavily on robust automated tests and system checks executed in environments that mirror production settings. Even if real-time releases are not always achievable, preparing for such a setup is advantageous (Humble & Farley, 2010, p. 300).

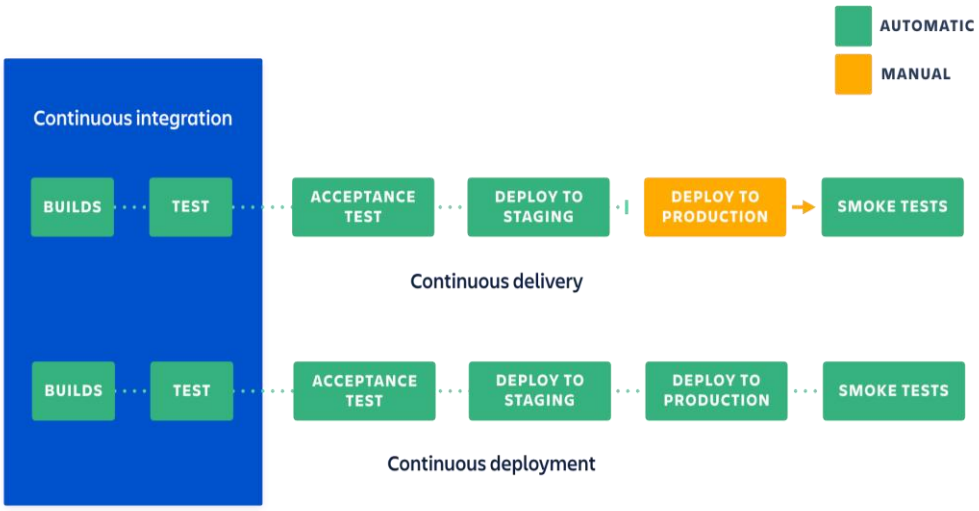
Broadly, software deployment involves:

- Preparing the environment, encompassing hardware, software, infrastructure, and external services.
- Integrating the desired application version.
- Configuring the application, integrating necessary data or states.

Modern tools and practices have made deployment smoother than ever. Today, automating deployment straight from version control systems is feasible, as is the automatic configuration of applications. With emerging technologies like Puppet and virtualization tools, even environment provisioning is becoming largely automated (Humble & Farley, 2010, p. 59).

Figure 3 shows the interconnections between Continuous Integration, Continuous Delivery, and Continuous Deployment. Continuous Integration lays the groundwork. Building upon it, Continuous Delivery and deployment follow. While they share foundational principles, they differ in execution. Continuous Deployment pushes the envelope of continuous delivery by ensuring automated, hands-off releases, underscoring the emphasis on automation and fluidity in modern software development. To answer this, Continuous Monitoring provides the lens, offering a view of the software's performance post-deployment.

Figure 3: Continuous Integration, Continuous Delivery and Continuous Deployment



Source: Pittet (2021)

2.2.5 Continuous Monitoring

Continuous Monitoring is a crucial aspect for anyone in the IT world, whether you are a developer, site reliability engineer, IT operations specialist, program manager, or a DevOps engineer. As applications shift from traditional on-premises setups to more complex, hybrid, or microservices architectures, it's essential to upgrade your skills and learn the best practices for effective monitoring, especially in hybrid or public cloud environments.

Azure Monitor is a powerful tool that can significantly help in continuous monitoring across various stages of your workflow. It's designed to work smoothly with development and testing tools like Visual Studio (hereafter: VS) and Visual Studio Code. This integration is not just limited to the development phase; Azure Monitor also plays a key role in release and work item management when used alongside Azure DevOps during deployment and operational phases (Bagaria, 2018).

Azure Monitor is not limited to using just by Azure services. It extends its capabilities to other IT Service Management (ITSM) and Security Information and Event Management (SIEM) tools. This flexibility allows you to track issues and manage incidents effectively within your existing IT processes, making it a versatile and integral part of any tech professional's toolkit (Bagaria, 2018).

2.2.6 Integration with Software Containers

Software containers have emerged as game-changers in the realm of CI, CD and CT. They provide a standardized environment, assuring that software behaves consistently across every phase of the development process. Whether it is on an individual developer's workstation or in the expansive production setting, the software behaves uniformly. Such predictability has addressed the age-old dilemma developers often faced, encapsulated in the phrase, "it works on my machine." In doing so, containers have greatly simplified both the integration and deployment phases (Battina, 2021).

The adoption of widely used DevOps tools and methodologies has significantly enhanced collaboration among developers, testers, and operations teams. This collaborative synergy not only streamlines the software delivery process but also allows for ongoing testing and continuous monitoring. This is evident across various stages, from development and integration to distribution. Containers are very common and widely used component in DevOps toolchain. Essentially, containers serve as lightweight, flexible platforms designed for easy software packaging and deployment. However, as beneficial as they are, there is a caveat: assessing the security of these containers can present challenges. Ensuring that they leverage secure libraries and are set up with optimal security configurations is paramount (Battina, 2021).

So, using methods like Continuous Integration, Continuous Delivery, Continuous Testing, and Continuous Deployment is crucial. The focus on testing, monitoring, and software containers ensures good and consistent software. These methods are key in modern software creation and create the way for the future. They greatly improve efficiency, reliability, and growth, making them central to the software field today. As we have seen these methods shape our present, it is intriguing to delve into the journey of a key player that has transformed our software practices - software containers.

2.3 The Evolution of Software Containers

2.3.1 Containerization: From Inception to Growth

Contrary to some modern examples, containerization is not a entirely new concept. IBM originally came up with the idea in 1979, which eventually grow into containerization technology. With its incorporation into the UNIX operating system V7, the chroot system call was introduced. This technology allowed for process isolation, designating specific user and kernel resource permissions. Such a foundational step laid the groundwork for the encapsulation techniques we now associate with today's container models (Bentaleb et al., 2022).

In 2013, the introduction of Docker caused a revolutionary shift in the containerization field. IT operations were quickly affected by this concept. Thanks to Docker, long-standing challenges like software portability have been effectively addressed. Docker's platform, being open-source, revolutionized the way we think about software deployment. One of its standout features is its capacity to host a diverse range of microservices. This flexibility translates into enhanced scalability for essential software applications, making it a pivotal factor in the growing interest in containerization (Raj & Raman, 2018).

The combination of microservices and Docker containers went to incredibly high value in a new era for both software developers and system administrators. Docker containers, celebrated for their lightweight design, when paired with the platform's standardized packaging, contribute immensely to streamlining and expediting software deployment. As described by Raj & Raman (2018), a container is more than just a package; it is a comprehensive unit hosting the software, relevant configuration files, dependencies, and binaries, ensuring that the software remains operational across varied environments.

2.3.2 Drivers Behind the Rise of Containerization

Containers have become a game-changer in modern software for several reasons. Firstly, they help cut costs. This is mainly because they tackle deployment issues that come up due to problems with dependencies. This means that both DevOps teams and production units can work more smoothly. Major companies like Microsoft, Amazon Web Services

(hereafter: AWS), Google, and IBM have backed this approach, hinting at a trend: Docker containers are quickly becoming the go-to method for deploying server-based software (de la Torre et al., 2023, p. 335).

This popularity is not just for the new, smaller software pieces, known as microservices. Even the bigger, traditional software built on platforms like the .NET Framework sees benefits, especially when using a special type called Windows Containers. For services based in the cloud, which can sometimes have interruptions, containers offer a sturdy solution. They can help handle hiccups like slow system responses or temporary network problems. Tools called orchestrators, such as Azure Kubernetes Service (hereafter: AKS) and Azure Service Fabric, play a big role too. They help manage the complex needs of containerized apps, showing just how vital they are in the growing world of container solutions (de la Torre et al., 2023, p. 336).

2.3.3 Key Milestones in Container Development

The journey of container development can be charted through several pivotal milestones:

1970s - The Inception of IBM Virtualization: In the 1970s, IBM's introduction of virtualization to their mainframes marked the basic beginnings of containerization.

2000s - The Era of FreeBSD Jails and Solaris Zones: During the early 2000s, the FreeBSD jail emerged as a pioneering technology. This mechanism facilitated OS-level virtualization, enabling administrators to segment a FreeBSD-oriented computer into multiple independent subsystems termed as 'jails' (Raj et al., 2015, p. 39). Concurrently, the Solaris Containers, inclusive of Solaris Zones, became recognized. Specifically designed for the x86 and SPARC systems, a Solaris Container combines system resource controls with the partitioning capabilities of zones. Such zones function as isolated virtual servers within a singular OS instance (Raj et al., 2015, p. 39). These innovations set the stage for the advanced containerization methodologies we recognize today.

2013 - The Rise of Docker: Docker surfaced in 2013 as a revolutionary shift in containerization. The main advantage of Docker lies in its ability to encapsulate applications ensuring universal portability, encapsulated in the vision of “create once, run everywhere.” Docker’s open-source platform accelerates the processes spanning from development to enhancement of containerized workloads (Raj & Raman, 2018). Docker effectively popularized container technology, making it accessible to a large developer community and creating an important movement in favor of containerization.

2015 - The Advent of Kubernetes: Originated by Google and subsequently open-sourced in 2014, Kubernetes (often abbreviated as k8s) emerged as a powerful orchestration platform. Its primary objective centers around automating the deployment of services encapsulated within multiple containers. This guarantees that these services are planned and run efficiently

on large scale. Kubernetes' popularity raised up, especially for managing scientific workloads, attributed to its deployment capabilities, flexibility, portability, and reproducibility (Bentaleb et al., 2022).

2016 - Present - The Phase of Refinement & Diversification: Container technology has witnessed considerable attention from renowned cloud service providers in recent years. For instance, AWS introduced Elastic Compute Cloud and, by 2017, began facilitating Elastic Container Service (hereafter: ECS) to establish Kubernetes clusters. The Google Cloud Platform offers tools such as Google Compute Engine for large-scale virtual machine operations, supplemented with Google Cloud Storage for expansive data object storage. Microsoft Azure brings forth Hyper-V containers, wherein each container operates within a specialized virtual machine. In parallel, Red Hat is OpenShift, rooted in Kubernetes, extends a variety of tools catering to container infrastructure—ranging from deployment to monitoring. Its offerings encapsulate security and cluster management features, making it a pivotal entity in the realm of container development (Bentaleb et al., 2022).

2.4 Major Players in the Container Ecosystem

Containers have been evolving since the 1970s. This journey has not been a series of sudden changes but rather a continuous development over decades. The rise of container technology is not a recent idea out of the blue. It is the result of many years of hard work, now reaching a broad audience. The growth of containers owes much to early thinkers who recognized their potential and the leaders who encouraged their widespread use.

2.4.1 Origins and Innovation: Tracing the Roots of Containerization

The containerization narrative began long before its recent surge in popularity. The foundational seed was planted in 1979 with the introduction of the chroot system call in the Unix V7 operating system. This marked a significant milestone in achieving process isolation, a key principle underlying containerization. Over time, multiple iterations and enhancements built upon this foundational idea (Osnat, 2020).

By the turn of the millennium, around 2000, an innovative solution known as FreeBSD Jails emerged from a hosting provider aiming to segregate its services from those of its clients. This separation was crucial for ensuring robust security and streamlining administrative tasks. Essentially, FreeBSD Jails provided administrators with the capability to segment a FreeBSD computer system into several discrete subsystems, aptly termed "jails." Each "jail" could then be assigned its own IP address and customized configurations, further solidifying the partition (Osnat, 2020).

As the 2000s progressed, Linux took significant strides in container technology. A particularly notable achievement was the introduction of Linux Containers (hereafter: LXC)

in 2008, representing a comprehensive container management solution. Preceding LXC, other pivotal developments included the Linux VServer in 2001, which enhanced the Linux Kernel, and Solaris' introduction in 2004. Solaris harnessed innovative features such as snapshots and cloning, courtesy of ZFS (Osnat, 2020).

Further expanding the container landscape was Google's 2006 launch of Process Containers, aimed at managing resource usage across processes. This initiative underwent rebranding as "Control Groups (cgroups)" a year later and was eventually integrated into the Linux kernel 2.6.24. Both LXC and cgroups highlighted the continuous evolution and refinement of container technology, with LXC utilizing both to function seamlessly on a unified Linux kernel (Osnat, 2020).

In the subsequent decade, Docker burst onto the scene in 2013, building upon the LXC foundation. Docker transcended being just another container solution; it presented an expansive ecosystem dedicated to container management, orchestration, and various other functionalities. While Docker's initial iterations were based on LXC, the platform eventually transitioned to its proprietary libcontainer library, underscoring its commitment to innovation and adaptability (Osnat, 2020). The journey from chroot to Docker represents a complex web of innovation, with each development contributing a vital piece to the containerization puzzle we are familiar with today.

2.4.2 Visionaries: Predicting the Rise of Containers

Leaders who predict and shape the direction of these breakthroughs are at the core of transformative technical developments. Seeing how quick decisions and a thriving market helped firms like Docker to flourish is fascinating. The birth of Docker's container technology is attributed to Solomon Hykes and his colleagues. Initially designed as an in-house tool for dotCloud platform services, the team made a game-changing decision in 2013 to make this technology open-source under the Docker brand. This move carved Docker's name into the annals of technological advancements (Bhartiya, 2017).

Using observations from a 2013 dotScale conference, Hykes explained the reasons behind the development of Docker. He made an important analogy between the difficulties the shipping industry experienced before the 1950s and the complicated nature of shipping software. He predicted that container technology will bring about a similar revolution in software deployment, similar to how containerization had revolutionized the transportation sector (Utekar, 2022).

Hykes openly commented on the fortunate timing of Docker's appearance: "Docker merely seized an existing opportunity, furnishing the right tools at the opportune moment. This evolution was inevitable, irrespective of Docker's involvement," he observed. Emphasizing the magnitude of this transformation, he added, "We are merely riding a colossal wave that is larger than Docker or any single entity, irrespective of its size. This wave represents a

profound shift across the entire technological realm". Reflecting on the historical challenges, distributed systems posed considerable obstacles for developers, especially when applications had to be built and deployed across a vast network of computers. With containers, Hykes did not just aim for an expert-only solution. Instead, he aspired to create a tool accessible and beneficial to the entire developer community (Bhartiya, 2017).

According to Burns (2018) in his blog post "The History of Kubernetes & Community Behind It": "Docker's emergence dramatically influenced the cloud landscape. Its lightweight container runtime simplified the packaging, distribution, and maintenance of applications, heralding a new cloud-native methodology. Indeed, without Docker's transformative impact on cloud development, Kubernetes might never have seen the light of day.

I remember Joe introducing Docker to us in the summer of 2013. Along with Craig and me, we were brainstorming ways to make cloud-native applications more accessible to a broader audience. The potential of Docker was instantly apparent. Yet, while Docker was great for building, packaging, and running containers on individual machines, a gap was evident: there was no tool to manage these containers across multiple systems. Kubernetes stepped in, simplifying the development process by incorporating essential features like logging and monitoring directly into the cluster. This allowed developers to leverage these features by merely deploying their apps onto the cluster.

Over time, our understanding deepened. For Joe, Craig, and me, it became evident that an orchestration tool was not just desirable—it was essential. And it was clear that this tool would be open-source. This realization in 2013 catalyzed the rapid development of what we now know as Kubernetes."

2.4.3 Champions of Container Technology

While visionaries laid out the roadmap for containerization, its widespread adoption required champions who would promote its benefits. Companies like Docker, Kubernetes, and Red Hat have been instrumental in evangelizing container technologies.

Docker played a pivotal role in bringing containerization to the forefront. Interestingly, Docker, Inc. did not begin as a software development company. Instead, they initially launched as a platform-as-a-service provider, aiming to help developers create and operate apps without the complexities of underlying infrastructure. However, financial challenges threatened the company's sustainability. In a turn of events, the team decided to open-source their container runtime engine, Docker, which was the backbone of their service, in 2013. Surprisingly, against their modest expectations, it gained significant traction. Realizing its potential, Docker, Inc. then strategically pivoted to focus primarily on the Docker engine (Sarmiento, 2020, p. 3).

Within just a month of its initial trial release, Docker captivated 10,000 developers. By the launch of Docker 1.0 in 2014, it had been downloaded an astounding 2.75 million times. And merely a year later, this figure soared past 100 million (Mell, 2023).

The emergence of Docker was timely, aligning seamlessly with the digital transformation wave that was reshaping businesses worldwide. This transformational wave drove companies of various scales and sectors to rapidly expand their digital capabilities. As a result, the IT sector saw a shift towards agile development, DevOps, cloud-native architectures based on microservices, and versatile cloud deployments (Chen, 2018).

Kubernetes has revolutionized the realm of container orchestration. To understand its significance, let’s delve into what container orchestration means. Essentially, it is a mechanism that facilitates the automation of tasks related to the deployment, management, scaling, networking, and ensuring the constant availability of applications housed in containers. In real-world production scenarios, ensuring that applications run continuously without interruptions is vital. Consider a situation where a particular container stops functioning; in such cases, it is crucial for a replacement container to kick in without delay. Ideally, we would want a system to manage this switch seamlessly. That is why Kubernetes is one of the champions of containers.

It serves as a comprehensive framework designed to manage distributed systems with a high level of resilience. Kubernetes simplifies many complexities associated with large-scale application deployment. Whether it is about dynamically scaling applications based on demand, ensuring there is a backup ready for failovers, or even implementing any deployment strategies, Kubernetes is equipped to handle it all. This platform offers a harmonious balance between flexibility and reliability, allowing developers and system administrators to focus on refining applications while Kubernetes takes care of the operational complexities (Malviya, 2020).

The widespread adoption of Kubernetes has prompted major cloud providers, including AWS, Google's Kubernetes Engine, Azure, and Oracle's Container Engine for Kubernetes, to introduce managed Kubernetes services. Additionally, top-tier software companies like VMWare, RedHat, and Rancher have launched Kubernetes-centric management platforms (Osnat, 2020).

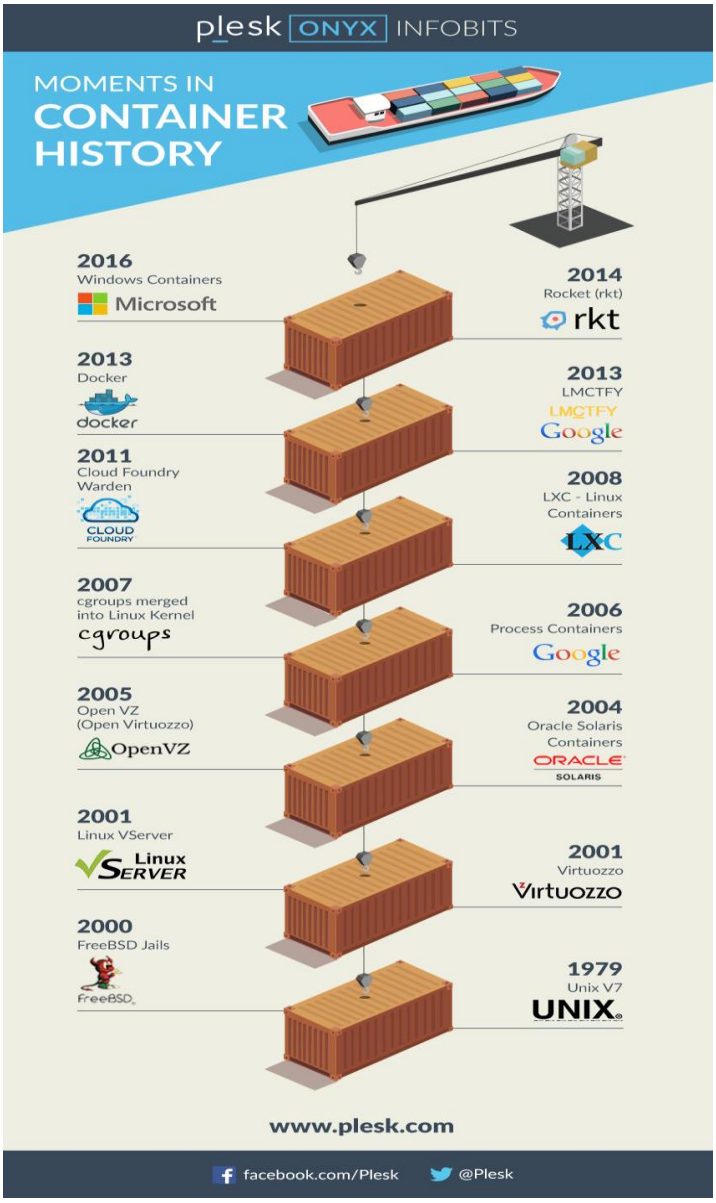
Red Hat is OpenShift stands out by offering an advanced platform tailored for enterprise-level, containerized applications. Defined as an enterprise-grade open-source application platform, OpenShift accelerates the development and deployment of cloud-native applications consistently across various cloud environments, extending to edge computing. Docker, too, acknowledges the enterprise requirements by offering its Enterprise Edition tailored for larger deployments (Red Hat, 2022).

While Kubernetes excels at container orchestration, it does not inherently encompass all the necessary components for a complete cloud-native ecosystem. Aspects like networking,

ingress, load balancing, storage, monitoring, logging, multi-cluster management, and Continuous Integration/Continuous Delivery (hereafter: CI/CD) are vital for the holistic development and deployment of containerized applications on a larger scale. Recognizing this, Red Hat OpenShift integrates these components, maintaining Kubernetes as its foundational layer, acknowledging that Kubernetes alone does not suffice for comprehensive cloud-native deployments (Red Hat, 2022).

Beyond companies, individual evangelists, open-source contributors, and community organizers have also played a role in championing the cause, ensuring that the advantages of container technology reach every corner of the IT industry. On the following Figure 4, we are having a more detailed visual representation of moments in container history.

Figure 4: Overview of Container history



Source: Strotmann (2016)

2.4.4 The Power of Community in Container Development

2.4.4.1 Open-Source Revolution

Open-source software's origins are deeply rooted in the early stages of computing, predominantly within academic and research circles where code-sharing was customary. The landscape shifted in the 1980s with Richard Stallman's establishment of the Free Software Foundation (FSF). His goal when starting the GNU Project was to create a free and open-source operating system that would resemble Unix. This project later served as the model for modern open-source concepts. A defining moment arrived in 1991 when Linus Torvalds introduced the open-source Linux kernel. This, when combined with the tools from the GNU Project, led to the development of the now-familiar GNU/Linux operating system (Israel, 2023).

Open-source represents a simple yet powerful principle: accessibility. It is the freedom to access and view someone else's, or even a company's, source code. This transparency allows individuals to adapt the code to their specific needs, identify errors, and correct them. What makes it even more compelling is the ability to share one's modified version of the code with the community, adding value to the original work. This open invitation to contribute and enhance is governed by specific licenses, each with its own set of guidelines and conditions. However, open-source is not just about the technical aspects; it signifies a broader ideology. In its early days, open-source emerged as an alternative to the closed, guarded world of proprietary software. Such software, with its restricted access to source code, was "owned" and tightly controlled by specific companies or entities. Contrarily, open-source championed openness, community engagement, and collective advancement. It was, in many ways, a counter-movement pushing for transparency, collaboration, and democratization of software development (Nduta, 2023)

Open-source software has increasingly resonated with developers, presenting them with a more efficient approach to building applications. Over time, these developers have become better in their skills, becoming adept at harnessing the capabilities of open-source technologies. Their influence on technology choices has expanded significantly. In addition, their preference for working together and openly discussing experiences online has sparked a change. Open-source companies have benefited from this change by gaining direct support from the developer community and avoiding traditional corporate IT decision-makers (Hilaly, 2014).

It was against this backdrop that Docker emerged. The project, grounded in open-source principles, swiftly captured the interest of developers. Its rapid growth did not go unnoticed. Tech giants such as Microsoft, IBM, and Red Hat were quick to take note, as were venture capitalists, who saw the potential and poured substantial investments into the burgeoning startup. It was clear that the era of containerization had dawned (Carey, 2021).

At the All Things Open conference, Jeffrey Hammond, who serves as Vice President and Principal Analyst at Forrester Research, unveiled a noteworthy trend: a staggering 80% of developers have, at one point or another, opted for open-source development tools community. This preference, he suggested, stems from a widely held belief among developers that open-source platforms deliver superior reliability and performance (Vaughan-Nichols, 2014).

Hammond's insights highlight a marked departure from earlier views about open-source. Traditionally, many professionals moved towards open-source primarily as a pragmatic solution to sidestep issues related to acquiring proprietary software and dealing with the burdens of licensing fees. However, Hammond identified a significant change in this perspective. Although, as he pointed out, cost-effectiveness used to be the main attraction of open-source, developers are now less concerned about it. In his words, the cost of the tool no longer sits atop the priority list for many in the developer community (Vaughan-Nichols, 2014).

2.4.4.2 Collaborative Initiatives and Global Gatherings

Open-source projects benefit from global contributors, offering varied insights and skills. This widespread cooperation fast-tracks innovation by merging knowledge and resources. Rapid deployment is essential in today's fast-paced software market, but establishing new infrastructure for each deployment can significantly delay product release. To reduce this lag, there is a growing need for a buffer between the physical machine and the deployable components. Containers adeptly fill this role, offering a convenient buffer that allows developers to essentially package mini-servers as the primary deployment units, optimizing the entire deployment flow (Israel, 2023).

Recognizing the transformative potential of containers, various technology leaders came together to forge a standard known as the Open Container Initiative (OCI) (Nortal, 2018). This initiative is not just supported by big industry names; it thrives due to the dedication of individual tech enthusiasts who committed both their time and technical prowess. Docker played a pivotal role as a founding member of the OCI, generously contributing the foundational code which later evolved into the runtime specification and, subsequently, the reference model (Walli, 2017). The cumulative efforts have culminated in OCI ensuring that container artifacts are consistent across different cloud and container platforms.

One particularly innovative outcome of container-based development is a novel architectural approach. In this setup, containers have minimal internal configurations. Instead, during startup, they communicate with external configuration servers. This approach allows containers to be more adaptable and versatile, operating seamlessly across various environments based solely on a simple change in an environment variable. Such a mechanism promotes a high degree of portability and flexibility. The emphasis on external

configuration ensures that the inflexible artifact remains consistent across various testing environments, resonating with one of the key factors of Continuous Delivery (Nortal, 2018).

Further amplifying the collaborative spirit in this domain is the Cloud Native Computing Foundation (hereafter: CNCF). Positioned as a prominent open-source project, the CNCF was kick-started by Google and soon gained the backing of tech giants like Cisco, Docker, IBM, Mesosphere, and Joyent, among others with their shared vision to harmonize and standardize scheduling and orchestration functionalities across the board. This collaborative effort is indicative of the broader industry's commitment to driving standardization and innovation in container technology (Hecht, 2015).

In a 2020 CNCF survey, there was an finding that container usage had grown by a staggering 300% since 2016. The International Data Corporation's (hereafter: IDC) study from May 2021 further highlighted the trend, suggesting a significant move towards containers over the ensuing three years. The research estimates that a substantial 80% of workloads will either migrate to, or start with, containers and microservices. This transformation is projected to reduce the infrastructure needs for each application by a noteworthy 60%, while at the same time enhancing the resilience of digital services by 70% (Hatheway, 2021).

There is also a noticeable increase in the creation of new application. As per IDC's insights, by the time we reach 2023, we can look forward to the birth of more than 500 million new logical applications. Remarkably, this number equates to the cumulative amount of applications crafted over the prior 40 years (Hatheway, 2021).

Transitioning to the 2022 CNCF survey, findings mirrored those of 2020: a significant 44% of participants reported using containers across the vast majority of their applications and business sectors. Furthermore, an additional 35% indicated that containers are implemented for at least some of their primary applications. However, it is interesting to note that the adoption of containers seems to be advancing faster than the maturity of cloud-native methodologies. This indicates many organizations are still in the exploring stages of their cloud-native transition. To illustrate, a mere 30% of the surveyed parties have fully embraced cloud-native methods in almost all their developmental and deployment activities. The positive side of this is that among those not deeply rooted in cloud-native techniques, 62% still utilize containers either for pilot projects or specific production scenarios, suggesting a potential for future expansion (CNCF, 2022).

The rising trend of adopting container technologies has not come without its fair share of queries and concerns from the community. To bridge the gaps in current container-based platforms and solutions, there is a pressing need for more in-depth studies. These investigations should encompass established best practices, essential functionalities, standards, and the array of tools available (Bentaleb et al., 2022).

While containers are celebrated for their minimal resource usage and streamlined deployment processes, the phrase that "there is always room for improvement" holds true.

The technological horizon is expansive, and the landscape is ever-evolving. As a testament to this evolution, unikernels are emerging rapidly as a potential successor or complementary technology to containers. Unikernels are designed to offer enhanced security and performance. Essentially, unikernels use a simplified operating system that is specially made for the particular application they run, providing improved separation and efficiency (Bentaleb et al., 2022).

2.5 Containerization

2.5.1 Understanding Containerization: Beyond Traditional IT approaches

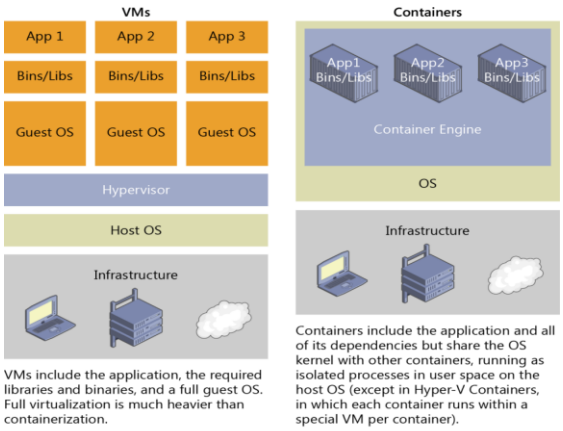
From its early days, virtualization technology has been a key part of computer technology's evolution. As today's information technology grows at a quick pace, both organizations and individuals increasingly rely on virtualization. This is due to its effectiveness in managing resource distribution and business tasks (Huawei, 2023, p. 109). Notably, IBM introduced virtualization in the 1960s as an alternative to multitasking systems (Humble & Farley, 2010, p. 303).

Virtualization primarily aims to streamline the presentation, access, and administration of IT resources, including infrastructure, systems, and software. Furthermore, it ensures standardized interfaces for these resources to engage in interactions (Huawei, 2023, p. 110). At its core, virtualization leverages the untapped potential of high-performance computers, negating the need for additional hardware purchases. This allows for optimized server utilization and facilitates quick system application delivery and recovery. To most, this is the clear essence of virtual computing. As virtualization continues to evolve, it is becoming integral to business management and operations. Its utility shows its value in scenarios like the speedy setup and transfer of servers and data centers, reflecting its power in transparent behavior management (Huawei, 2023, p. 111).

Within the realm of data center environments, there are two main categories of virtualization technology: hardware-level virtualization and operating system-level virtualization. Hardware-level virtualization leverages a tool known as a hypervisor. This tool takes the server's resources and partitions them into multiple Virtual Machines (hereafter: VM). Notably, in this hypervisor-based virtualization setup, each VM operates with a distinct operating system and specific libraries to manage diverse applications, often leading to inefficiencies. On the other hand, operating system-level virtualization, also known as containerization, is directed by the operating system itself. This form of virtualization bundles the OS libraries and their respective dependencies into a unified entity, resulting in the creation of containers. An intriguing aspect here is that all these containers, though distinct in functionality, are under the umbrella management of the host OS kernel (Yadav et al., 2021).

For a more visual representation of the differences between VM and containers, one can refer to Figure 5.

Figure 5: Differences between Virtual Machines and Containers



Source: de la Torre, (2022, p. 9)

In the above Figure 5, VMs have three foundational layers within the host server. Starting from the base, they are Infrastructure, Host Operating System, and the Hypervisor. Atop these layers, each VM houses its own distinct Operating System (hereafter: OS) and the requisite libraries. Contrasting this, when we look at containers, the host server is streamlined, consisting only of the Infrastructure and the OS. Built upon these, a container engine functions to maintain the individuality of each container while permitting them to collectively access the services of the foundational OS (de la Torre, 2022, p. 10).

A noteworthy advantage of containers is their resource efficiency. Unlike VMs, they do not require a comprehensive OS, making them both rapid to initiate and deploy. Containers do not need to start the entire operating system, so container deployment and startup are faster, less expensive, and easier to migrate (Huawei, 2023, p. 307). This inherent efficiency boosts the density, enabling a larger number of services to operate on a single hardware unit and subsequently leading to cost savings. However, an inherent trade-off exists; containers, by virtue of sharing the same kernel, achieve slightly less isolation compared to VMs (de la Torre, 2022, p. 10).

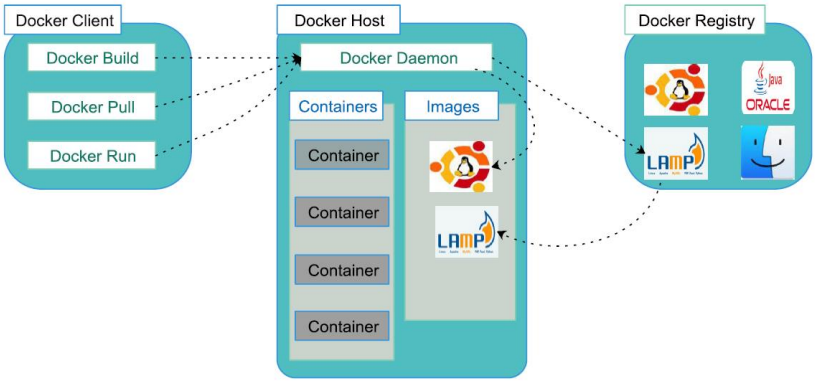
2.5.2 Anatomy of a Container

In this section, I will discuss the anatomy of containers with focus on the Docker ecosystem.

The Docker ecosystem consists of two primary components. First, there is the open-source platform known as Docker, dedicated to containerization. Then there is Docker Hub, a cloud-based service platform designed for sharing and managing Docker containers. At its core,

Docker operates on a client-server model. To offer a clearer visualization, the following Figure 6 provides a representation of Docker's architectural design (Singh & Singh, 2016).

Figure 6: Docker's architectural design



Source: Singh & Singh (2016)

Moving from a broader overview to specific components, the Docker Client serves a dual purpose. First, it receives commands directly from users. Secondly, it facilitates two-way communication with the Docker Daemon, ensuring seamless interactions and command execution within the Docker ecosystem. This bidirectional communication is vital for the efficient management and operation of Docker containers (Singh & Singh, 2016). Diving deeper into this interaction, the Docker Client interacts with the Docker Daemon using the REST API. This means that they establish communication using standardized web-based protocols, ensuring smooth and consistent exchanges of information (Sarmiento, 2020, p. 45).

Moving on to another fundamental aspect, it is essential to distinguish between an image and a container itself. (Docker) image serves as a static, read-only blueprint that sets the stage for creating application containers. Think of it as a non-active framework containing all necessary components for an application to function but without being in use itself (Sarmiento, 2020, p. 46). Its main goal is to ensure uniformity in environments across different deployment platforms. This means if you debug an application on one machine, you can be confident it will function similarly in another environment, thanks to the image. Such container images have evolved into a pivotal means of packaging applications or services, guaranteeing both consistency and dependability during deployment (de la Torre, 2022, p. 14).

Expanding on this concept, a Docker container comes to life when this image is activated (Sarmiento, 2020, p. 46). Docker images are created using a Dockerfile, which can be thought of as a recipe (Yadav et al., 2021). Having discussed the role of a Dockerfile, let's delve into its specific instructions. Table 1 on the following page summarizes some of the core commands used in a Dockerfile:

Table 1: Dockerfile commands

Name of command	**Description of command**
FROM	The FROM instruction specifies the starting point or base image for building a Docker image, such as OS or a pre-existing image.
LABEL	The LABEL instruction adds descriptive metadata to the image, like details about the base image used, the application it runs or its version.
RUN	The RUN instruction executes commands on new layer above the base image and saves the outcomes, allowing the building of the images in steps.
CMD	The Command-Prompt (hereafter: CMD) provdies default commands and parameters for containers based on the image. A Dockerfile contains just one CMD instruction presented in a Javascript Object Notation (hereafter: JSON) array format., if there are multiple the last one takes precedence.
EXPOSE	EXPOSE notifies Docker that the container listens on specific network ports, with an option to use TCP (Transmission Control Protocol) or UDP (User Datagram Protocol).
ENV	The ENV keyword sets environmental variables that remain consistent when a container is launched from the image, often used for configuring software contained in the image.

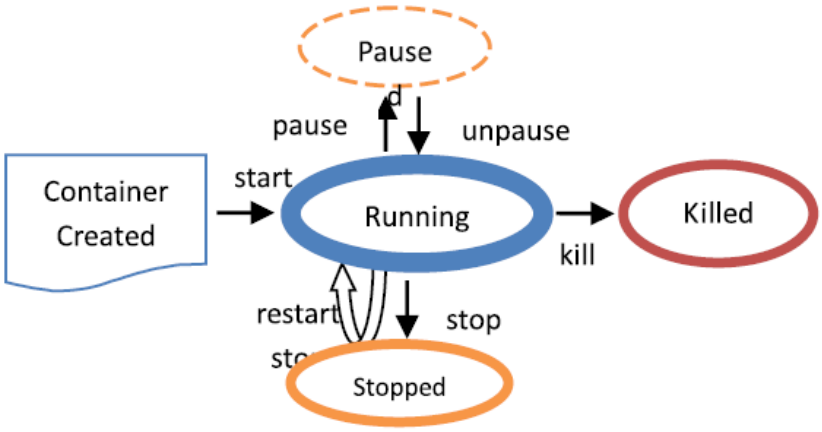
Source: own work.

These foundational commands play a crucial role in constructing a Docker image. At its core, a base image is streamlined, featuring only the essential instructions and drivers to run an application. To bring a Docker image to life, developers employ the Docker build command targeting the Dockerfile. It is noteworthy that every single instruction in a Dockerfile, once executed, adds a distinct layer to the image. If a Dockerfile undergoes modifications and the image is subsequently rebuilt, only the layers corresponding to the newly introduced instructions in the Dockerfile are refreshed or added (Yadav et al., 2021).

This process is followed until the desired working environment is fully set up and operational (de la Torre, 2022, p. 19). So, Docker container is the runtime instance of the earlier mentioned static image runned on the Docker host (Sarmiento, 2020, p. 46).

Building on this understanding, let's delve into the lifecycle of a container. As represented in Figure 7, the entire journey of a container begins with the creation of its foundational image and progresses towards its running state. But it does not just stop there. Containers can traverse through various states such as being paused, killed, or even stopped altogether. This dynamic progression offers a broad view of how containers evolve and interact within the Docker ecosystem (Bentaleb et al., 2022).

Figure 7: Containers lifecycle



Source: Bentaleb et al. (2022)

Having outlined the dynamic lifecycle of a container, it is imperative to understand the supporting infrastructure that facilitates these transitions and operations. Central to this is the Docker host. This pivotal component within the Docker ecosystem can be a physical machine or a virtual one. This machine serves as the foundational platform where the Docker Daemon orchestrates the deployment of containers. Docker Daemon, referred to as a service in the Windows environment, functions as a client-server application and serves as the backbone for both creating and managing Docker objects. This Daemon, or runtime engine, is pivotal to the Docker ecosystem, ensuring smooth creation and operation of Docker elements (Sarmiento, 2020, p. 45).

It is the central point of your interactions with Docker, serving as a point of entry for all the various parts and aspects that make up the Docker ecosystem. Therefore, the best place to start for anyone trying to understand the complexities and workings of Docker as a whole is by having a thorough understanding of the Docker Daemon (Sarmiento, 2020, p. 46). Essentially, the smooth functioning of containers can be mostly ascribed to the cooperative efforts of the Docker host and the Docker Daemon, which collaborate to oversee and control the lifetime and activities of containers in the Docker ecosystem (Yadav et al., 2021).

In controlled environments, such local development systems or personal PCs, containers are usually evaluated before being widely distributed. Consider a registry to be an enormous, well-organized library where these container pictures are kept. To see how crucial such registry is we are seeing through the scenario of shifting from testing to fully functional production (Sarmiento, 2020, p. 47).

Docker Hub, managed by Docker itself, emerges as a renowned public registry catering to this need. However, in the wide landscape of options, there is a spectrum of alternatives too, like the Azure Container Registry (hereafter: ACR), presenting a diverse array of image choices. Furthermore, for businesses keen on exerting greater control or ensuring heightened confidentiality, establishing a proprietary, on-premises image registry becomes a viable option. Now, to draw a more relatable analogy, envision this registry as an expansive bookshelf, each "book" representing a unique container image. This is somewhat parallel to platforms such as the Apple App Store or the Google Play Store, but specifically tailored for Docker images. These images, always at the ready, can be swiftly activated into containers, forming the backbone for various digital services and web applications (Sarmiento, 2020, p. 47).

While Docker Hub is handy, it comes with some challenges. Relying on a constant internet connection and facing slow download and upload speeds can be problematic. Images on Docker Hub are open for all to see. Yes, there is an option to use a private storage area, but it is not free. Many companies, thinking about security, prefer not to put their images on outside platforms. This is why many are looking into creating their own local storage spots or registries (Huawei, 2023, p. 317).

Addressing the need for more control, private image registries, whether on-site or in the cloud, present a solution. Using them is advisable when you need to keep your images confidential and prevent public access and when you aim to reduce network delays between your image storage and deployment setting. For instance, if you are using Azure as your main production environment, storing your images in the ACR can help achieve faster access due to reduced network latency. Similarly, for setups that are primarily on-site, having an on-premises Docker Trusted Registry in the same local network can be beneficial (de la Torre, 2022, p. 15).

Docker Hub does not stand alone; it is complemented by the Docker Trusted Registry, specifically created to meet enterprise-centric requirements. In the broader tech landscape, heavyweights like AWS and Google have staked their claim, rolling out their container registries that integrate as foundational elements in their cloud solutions (de la Torre, 2022, p. 14).

In conclusion, the anatomy of software containers, especially those present in the Docker ecosystem, serves as an example of the significant evolution of software deployment. Fundamentally, Docker provides an architecture that enables programs to function consistently on a variety of systems. Modern software is stored, distributed, and executed in a manner that is highlighted by its architecture, the distinction between images and containers, and the crucial role of registries like Docker Hub. Docker containers' modularity and consistency have proven indispensable to companies digital transformation, making them an essential element in the IT world.

2.5.3 Advanced Container Insights

While the information in the above section gives a basic overview of Docker containers, there are advanced features and deeper levels to investigate as with any technology. These small aspects provide information on networking, security, optimization, and the integration of Docker into broader ecosystems, all of which are crucial for companies wishing to fully utilize containerization. Let’s explore further into the world of software containers by delving into more sophisticated insights and elucidations now that this basic understanding has been established.

2.5.3.1 Container Networking

Containers can communicate with each other in three primary ways: through their IP addresses, using Docker's built-in DNS service, or by being joined together in a shared network environment (Huawei, 2023, p. 319). Docker has built-in a range of native networking options to accommodate various application needs, including None, Host, Bridge, Overlay, and Macvlan. Broadly, these networks can be categorized into two groups: those for containers on a single host and those for spanning across multiple hosts. Upon installing Docker, a user will find three default networks already set up on the host. These can be easily inspected using the docker network ls command.

In his book from 2020 Sarmiento provided a clearer understanding and a brief description of some of these networks:

– None: This network is isolated, essentially making it a closed-off environment. It is particularly useful for applications that prioritize high security and have no requirement for networking capabilities.

- Host: When containers are connected to the Host network, they directly tap into the Docker Host is networking stack. This means that the network setup of the container mirrors that of the Docker host itself.
- Bridge: Docker employs the Bridge network to establish a communication channel between two containers. It acts like a virtual link, allowing the containers to interact with each other.

These networking choices are adaptable and can accommodate a range of operational requirements, so containers can connect with one other as needed. Beyond the default trio of networks—None, Host, and Bridge—set up by Docker, users have the flexibility to establish custom networks that align with their business requirements. Docker offers three main drivers for this custom configuration: Bridge, Overlay, and Macvlan. It is worth highlighting that the Overlay and Macvlan drivers are especially designed for establishing networks that extend across multiple hosts, facilitating a more interconnected container environment (Huawei, 2023, p. 320).

2.5.3.2 Container Security

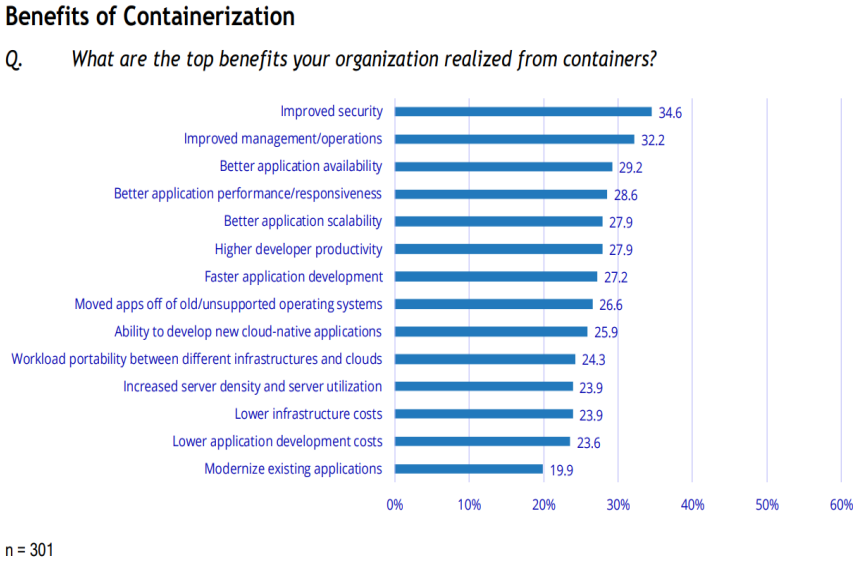
Docker employs multiple layers of security defenses to counteract breaches. This means even if a single defense line is compromised, others immediately step in to prevent unauthorized access to containers. When assessing Docker containers' security aspects, several key areas warrant close attention (Singh & Singh, 2016).

A crucial aspect of container security is the principle of isolation, which plays a pivotal role in bolstering security levels (Bentaleb et al., 2022). Docker has comprehensive security measures that allow teams to implement a model of microservices with the least privileges. In this setup, essential resources for the service—ranging from other applications to sensitive data and computing assets—are generated and accessed dynamically (Huawei, 2023, p. 317).

To enhance security, Docker containers utilize mechanisms like namespaces and Cgroups. Namespaces provide a variety of types, including user, net, PID, mnt, Cgroup, time, and others. These namespaces restrict user domains and offer dedicated Linux kernel resources (like user management, filesystem, network, and hostname) to the container. Meanwhile, the Cgroups kernel mechanism regulates resource usage by managing subsets of processes (Bentaleb et al., 2022).

A 2018 survey by IDC emphasized the importance of security in container infrastructure. As depicted in Figure 8, security was highlighted as both a top advantage and motivation for containers, granting users the capacity to develop enhanced applications and systems. This security layer not only expedites patching but also sharpens the response to emerging threats. However, as a relatively new technology, container security also presents challenges. The learning curve involved in mastering this new layer of protection can be steep for some (Chen, 2018).

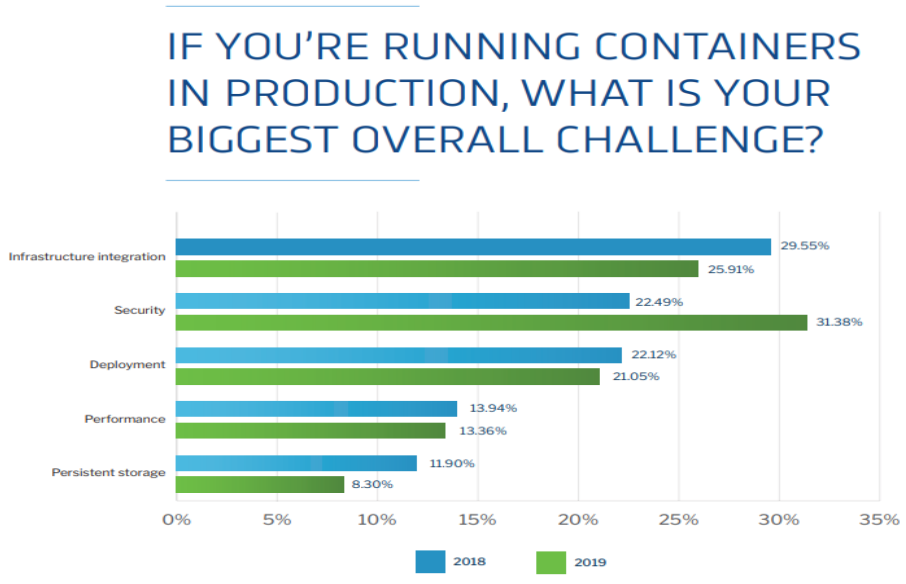
Figure 8: Benefits of Containerization survey – top benefits



Source: Chen (2018)

In Figure 9, we will be able to see backing this observation, a 2019 survey by Diamanti, involving over 500 IT organizations, pinpointed security as the paramount concern associated with the technology, closely followed by infrastructure integration (Mell, 2023).

Figure 9: Benefits of Containerization survey – biggest challenge



Source: Mell (2023)

An illustrative case in point is the Finnish Railways, which heavily relies on Docker Enterprise Edition's security capabilities. Recent measures include the integration of image scanning to preserve container code integrity. Role-based access control emerges as a

fundamental feature, and the organization also trusts the Docker Trusted Registry while avoiding public ones (Chen, 2018).

According to Garg & Garg (2019) widely endorsed tools for enhancing Docker's security include:

- Docker Bench for Security: This script checks for compliance to best practices in deploying Docker in live environments, also scrutinizing host and Daemon configurations.
- CoreOS's Clair: A tool to scan images, either locally or on public repositories, for security vulnerabilities.
- Docker Security Scanning: Assists in vulnerability detection for Docker-hosted containers.
- AppArmor/SELinux: While not exclusive to Docker or containers, it adds another protective layer.

2.5.3.3 Orchestration

Automation focuses on streamlining individual tasks, such as launching or configuring a web server. It is about simplifying specific operations to function without manual intervention. Orchestration, however, delves deeper. It involves coordinating multiple automated tasks in a structured sequence to execute a comprehensive process. In essence, while automation perfects single tasks, orchestration ensures these tasks flow harmoniously from start to finish, adhering to a predefined workflow.

In cloud environments, processes often involve numerous tasks executed in a particular order to accomplish business or operational objectives. Orchestration has thus emerged as a pivotal aspect for the effective functioning and evolution of cloud services. It is the foundation that holds the cloud paradigm together, orchestrating multiple tasks to produce desired outcomes.

Expanding on this, cloud orchestration is complex. While automation zeroes in on individual operations, orchestration supervises the entire process. It encompasses end-to-end management, from ensuring high availability and overseeing post-deployment to facilitating failure recovery and scaling. In simpler terms, while automation refines singular tasks, orchestration coordinates these refined tasks, managing their interdependencies. It is the higher-tier process that "automates the automation" (de la Torre, 2022, p. 13).

Orchestrators extend the capability to manage complex, multi-container workloads across a host cluster. By removing the direct interaction with the host infrastructure, these orchestration tools allow the entire cluster to be seen and operated as a single deployable unit. Orchestration, in essence, encompasses a platform equipped with tools that can manage an application’s full lifecycle. It spans the initial container deployment; the shifting of containers based on host health or performance; versioning, updates, and health monitoring

to aid scaling and recovery, and much more. When discussing orchestration, we refer to container scheduling, the overall management of clusters, and potentially the provisioning of additional host resources. Given the complexity of these tasks, it is generally more practical to adopt pre-existing orchestration solutions from established vendors (de la Torre, 2022, p. 13).

With orchestrators, users can command their images, containers, and hosts either through a command-line interface (hereafter: CLI) or a graphical user interface. These tools cover container networking, configurations, load balancing, service discovery, high availability, Docker host configurations, and more. At their core, orchestrators manage the operation, distribution, scaling, and maintenance of workloads across a node collection. Typically, the tools offering clustering infrastructure also provide orchestrator products (de la Torre, 2022, p. 13).

Popular orchestration solutions, such as Kubernetes, Docker Swarm, and Apache Mesos, focus on container lifecycle management. These platforms offer features optimized for scheduling containers, ensuring resources stay within set limits, maintaining fault tolerance, and facilitating auto-scaling. Among them, Kubernetes and OpenShift have seen substantial adoption across various sectors, from industry to research. In this context, Kubernetes will be our primary focus due to its superiority among available alternatives. Kubernetes has emerged as the go-to standard for container orchestration. It follows closely behind big data, cloud computing, and Docker in its popularity. Its significance for the IT industry cannot be understated (Huawei, 2023, p. 327). Firstly, it was built by Google, but in 2014 it went open-source (Bentaleb et al., 2022).

According to Huawei (2023, p. 328) the advantages of Kubernetes are following:

1. Robust Container Orchestration: Kubernetes has evolved alongside Docker, leading to a deep integration. It possesses robust container orchestration features, including container composition, label selection, and service discovery, aligning with enterprise demands.
2. Modularity and Lightweight: Abiding by microservice architecture principles, Kubernetes divides the system into distinct functional components. These have well-defined boundaries, ensuring easy deployment across diverse systems and environments. Moreover, many of Kubernetes' features are modular and can be conveniently expanded or swapped.
3. Open-source and Community-Driven: Kubernetes aligns with the trend towards open-source solutions. It has drawn numerous developers and businesses into its community, jointly fostering a rich ecosystem. Collaborating with other open-source communities, such as OpenStack and Docker, Kubernetes offers both businesses and individuals opportunities to contribute and benefit.

2.5.3.4 Integration of Docker and Kubernetes into Broader Ecosystems

When considering the creation of a containerized environment, turning to cloud providers can be the most straightforward approach. By leveraging cloud services, organizations not only alleviate upfront capital expenses but also reduce the maintenance burden on their teams. Such a model also empowers organizations with the agility to scale resources based on their immediate needs (Weave, 2017).

Two giants in the cloud arena, AWS and Microsoft Azure, have emerged as strong contenders for those seeking a container-centric infrastructure. Gartner, a renowned research company, has underscored their dominance by pointing out that AWS satisfies 92% of its cloud evaluation criteria, while Azure meets 88%. For many businesses, an existing relationship with Microsoft becomes a compelling reason to gravitate towards Azure (Weave, 2017).

AWS offers robust support tailored for Docker, encompassing both its open-source and commercial variants. There is a plethora of methods to deploy containers on AWS. For instance, ECS is a potent solution designed for efficient container management. It provides a seamless transition for businesses, allowing them to deploy containerized applications from their local Docker setup directly to ECS (AWS, 2023).

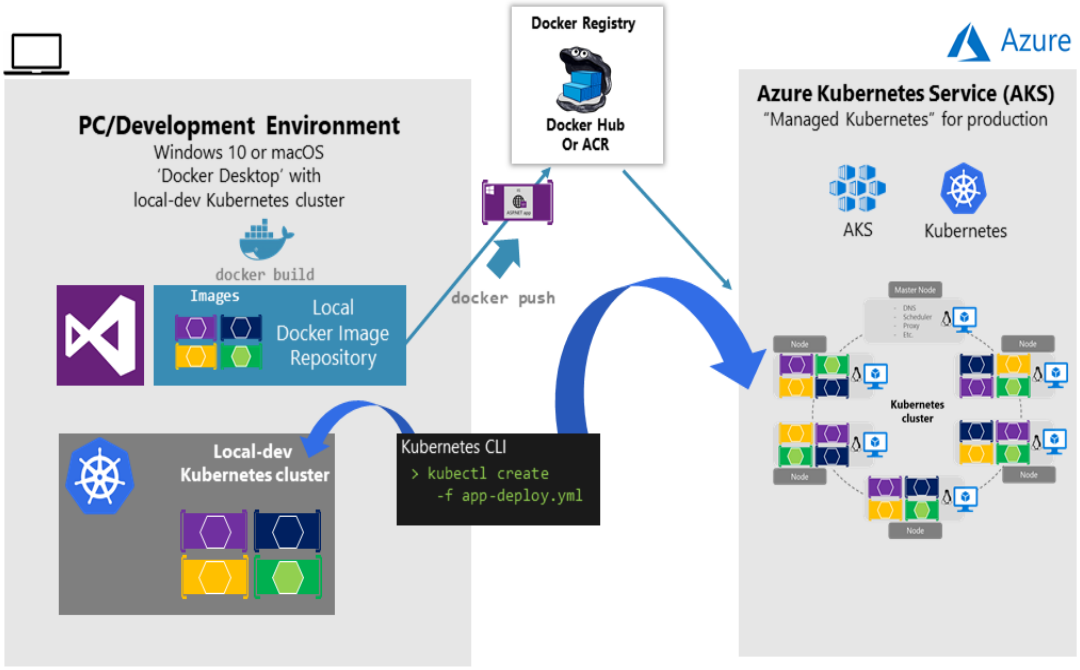
Additionally, AWS Fargate is an innovation that frees users from the complexities of infrastructure management, enabling them to focus solely on their containers. For those inclined towards Kubernetes, AWS has not lagged behind, introducing the Amazon Elastic Container Service for Kubernetes (hereafter: EKS). Similarly, the Amazon Elastic Container Registry (hereafter: ECR) provides a fortified environment for storing and handling Docker container images, emphasizing both speed and security. Another noteworthy service is AWS Batch, tailored for executing extensive batch processing tasks using Docker containers (AWS, 2023).

On the other hand, Microsoft has integrated Docker seamlessly with Azure. The Docker-Azure integration empowers developers to deploy applications in Azure Container Instances (hereafter: ACI) and AKS effortlessly. However, Docker's support for ACI ended in November 2023 (Docker, 2023). AKS is a specialized service within Azure that provides managed Kubernetes orchestration. This service is created to reduce the challenges associated with the management, deployment, and daily operations of Kubernetes clusters, making the user experience more seamless. By doing so, it ensures more efficient cluster operations within the Azure framework (de la Torre, 2022). This integration fosters a harmonious workflow between Docker Desktop and Microsoft Azure. Developers, therefore, find it convenient to toggle between local development and cloud deployments, be it through the Docker CLI or the Visual Studio Code extension (Docker, 2023).

In Figure 10, we will demonstrate the process of containerizing our application, which was developed in VS, within a local Docker environment. Afterward, these container images will

be uploaded to Docker Hub. Subsequently, we will deploy these microservices using Kubernetes. Following this setup, we will transition to a cloud-based approach, deploying through AKS and uploading the images to the ACR (de la Torre, 2022, p. 39).

Figure 10: Process of containerizing application using Docker, Visual Studio and Azure



Source: de la Torre (2022, p. 39)

2.5.3.5 Monitoring and Logging

Monitoring involves keeping an eye on an application as it runs, assessing its performance and stability. It encompasses defining thresholds for different system components, and if any metric crosses these set thresholds, the DevOps system springs into alert mode (Raj & Raman 2018).

To make the most of the data gathered by monitoring tools, it is crucial to analyze this data comprehensively. By delving deeply, teams can quickly obtain crucial insights that point them in the direction of the most effective solutions. These insights are indispensable for identifying and addressing the root causes of any identified issues (Raj & Raman 2018).

Through proactive performance monitoring combined with predictive analytics, IT teams are equipped to identify and tackle potential issues before they affect end-users. A robust platform should efficiently process data from diverse sources and APIs, presenting this data in a comprehensible manner. Accessible graphics and visual displays can significantly expedite IT teams' response to any events that may influence the service quality (Raj & Raman 2018).

There is a multitude of software solutions tailored for monitoring cloud infrastructure performance. One such notable solution is Prometheus, renowned for its seamless integration with Kubernetes. It is designed to track performance metrics from both hosts and containers, storing this data in a time-series database (Gonzalez & Arzuaga, 2020)

Kubernetes aids in balancing workloads, leveraging foundational monitoring, logging, and health-checking mechanisms (Bentaleb et al., 2022). Management and monitoring within IT encompass various methods to oversee production applications and services, creating a holistic view by merging multiple advantage points (de la Torre, 2022, p. 24).

The monitoring landscape is expansive, with a multitude of specialized monitoring solutions tailored to meet the distinct requirements of different organizations. Each solution is typically designed to fit the particular operational characteristics of individual companies (Tamburri et al., 2019).

3 METHODOLOGY

In the world of IT, understanding software containers is very important. Having outlined the research methodologies in Introduction, this section explores into specifics how exactly the research was conducted. It provides a thorough explanation of the actual procedures followed in order to gather and evaluate data, guaranteeing transparency and comparability of the study procedure.

3.1 Description of the company

In this master thesis, we examine the company that was found in 2020. This company is a member of a group, which was established back in 1990 in Slovenia. For over 33 years, they have been assisting businesses by providing advanced tech solutions that are specifically designed for the needs of their partners.

Having started in Slovenia, the company has now grown its presence in North America, Asia, and many parts of Europe like Bosnia and Herzegovina and Serbia. Its customers include big names in software across the world. They trust the company for its top-quality software services that tackle all sorts of tech challenges. One of the things that stands out about the company is how active it is in the tech community. They join various tech events, from workshops to big conferences. Whether they are hosting or just attending, they are always eager to learn and stay updated with the latest in the industry. Safety and trust are big for this company. They take great care to protect their information and follow all the rules, especially when it comes to handling personal data, notably GDPR (General Data Protection Regulation). They are always looking for new ways to grow and innovate. This is why they invest in research and work closely with schools and colleges.

Over the past few years, they have started working in new areas such as Cybersecurity, Customer Relationship Management (hereafter: CRM) solutions, Cloud & Hybrid cloud architectures, DevOps, Monitoring systems, and Analytical services. They are also exploring advanced tech areas like Artificial Intelligence and Machine Learning. Right now, they have a strong team of over 300 skilled people working on various projects. And with the business doing so well, they are always looking for more talented people to join them.

I chose this company for the case study due to its active involvement in a range of technological advancements, including DevOps and containerization, areas of personal interest to me. The company's pioneering use of containers, combined with their rapid growth and expansion into innovative technological domains, makes them an ideal candidate for this study. Their unique approach to integrating container technology across diverse IT landscapes provides a rich context for analysis.

3.2 Description of the case

During my time at the company, I worked as a backend software developer, specifically on a data backup project. I was part of the server team, collaborating closely with five other experienced software developers. Beyond our group, the company had a dedicated engine and a QA team, each consisting of four members.

My curiosity about exploring DevOps projects began to grow just as I was becoming interested in cloud technologies as well, particularly containers. As luck would have it, just as I was thinking about exploring these areas, the company started doing DevOps in one of the projects at that point.

One of the main people involved in this new company project was a senior system engineer. Given his role, he was deeply involved in moving the company's systems to the cloud and using software containers to make everything work smoothly. We often discussed the project, and he would share updates, challenges, and successes.

Seeing my interest, he explained many of the technical details to me, helping me understand how software container strategies were being implemented in the company. This gave me a great idea. I decided to combine my personal interest with the ongoing company project and use it as the basis for my thesis research. Together, we went on a journey of analyzing the present business processes, and subsequently introducing innovative solutions. With this background knowledge and firsthand experience at the company, I designed my research approach as follows.

3.3 Research design

To answer the main research question: "What are the benefits and costs if we introduce software containers to the selected company?", as mentioned I used a mix of research

methods that were explained in first section of the thesis. Another beneficial thing was my role in the company which gave me a chance to see how the DevOps project was unfolding. By simply being there in office from December 2022 to January 2023, I could see some of the day-to-day work and the general challenges and successes related to using software containers.

Building on this observation, I conducted interviews with the senior system engineer, given his vital role in the process. These interviews were conducted in our Ljubljana headquarters every Friday in August 2023. Each interview session lasted approximately two hours, offering a deep dive into the motivations, challenges, successes, and learnings from the containerization initiative. Given the challenges of implementing software containers and the many factors involved, it was important to gather detailed data on firsthand experiences. Interviews, compared to methods like surveys or observation, give a direct insight into the firsthand experiences of experts working on the process. These discussions are open, which means we can get detailed insights that might be missed with other methods. Such a methodology aligns with the research principles outlined by Kothari (2009), who emphasizes the importance of detailed, firsthand information in understanding complex processes. Since the senior system engineer played a key role in using software containers, interviewing him seemed the best way to understand both the technical and practical sides of the work. This method was chosen because it seemed the best way to really understand the challenges, methods, and details involved.

By merging my research insights with the senior system engineer's firsthand experiences, we achieved a decent view of the software container implementation process. For clarity and ease of reference, all findings, observational notes, and interview details have been systematically documented in an appendix.

The secondary research was already discussed in the earlier "Literature Review" section. In that part, we looked at what other experts and researchers have said about software containers. We used important books, articles, and research papers to understand the topic better. We chose these resources carefully, making sure they were relevant and trustworthy.

Once we collect all the data, both from interview and written sources, we analyze it carefully. We will share what we learned from the IT engineer in the "Interview Analysis" section. This will include both the challenges they faced and their successes. The "Analysis of Present Business Processes" section will then take a close look at how business operates today, leading to suggestions on how software containers can make things better.

Throughout our research, we made sure to follow ethical guidelines. We told the person we interviewed about our study's goals, and we kept their information private. We also made sure to give credit to all the sources we used in the secondary research.

To wrap up, the way we approached this study was thorough and clear. Our goal is to give a full view of how software containers are used in IT today. In terms of my contribution to the

research, I was involved in all aspects of it. I was responsible for designing the research plan, collecting and analyzing data and drawing conclusions that we could actually use. I conducted in-depth interviews with an IT engineer, paid attention to the business's procedures, and carefully looked through a number of other relevant sources. My main goal was to contribute to the advancement of knowledge in this field by clearly demonstrating the current use of software containers in the IT industry. The findings and their application to optimize software containers for IT operations are presented in the following sections of this thesis.

4 RESEARCH ANALYSIS

This segment seeks to bridge the gap between the theoretical knowledge presented earlier and the practical insights from the field, aiming to provide a comprehensive understanding of the significance, challenges, and prospects of software containers in the modern IT landscape.

The personal interview method involves an interviewer who asks questions directly to individuals in face-to-face interactions. This method can be used for intensive investigations, where the interviewer collects information directly from the sources. However, it may not always be feasible or practical to contact individuals directly, especially for extensive inquiries. In such cases, an indirect oral examination can be conducted, where the interviewer questions individuals with knowledge about the subject and records their responses which was in our case (Kothari, 2009).

Given our research objective to determine the benefits and costs of introducing software containers to the company, this interview with a IT engineer offers critical firsthand insights into its implementation. The adoption and integration of software containers into the technology stack of a company invariably affects its business processes. On the other hand, initial challenges with Kubernetes and security concerns remind us that embracing container technology is not without its complexities. This chapter offers an analysis of the present business processes and practices surrounding the utilization of software containers, based on insights from the interview with a senior systems engineer.

4.1 Analysis of Present Business Processes

Using the insights gained from the interview, let’s contextualize them by examining the company's current operational methodologies. The world of software development and deployment is constantly changing. This means companies must keep adjusting and looking at their methods to make sure they are up-to-date and working effectively.

In this section, we will take a look at the present business processes, focusing on how the organization uses software containers. Software containers, as understood from the

interview, have become a crucial asset for the company, playing a pivotal role in the onset of new projects. Delving deeper, it is observed that tools like GitHub, GitLab, Jenkins, and Kubernetes form the backbone of their technological infrastructure. Docker containers, for example, enhanced the deployment speed of the health and wellness tracker project by approximately 40% highlighting their practical utility in real-world scenarios. To further understand the broader implications of adopting such technologies, it is crucial to evaluate both the tangible and intangible costs and benefits associated with them.

4.2 Current Implementation and Future Vision

This part analyzes the current state of software containers inside the organization, their projected future, and a thorough examination of the actual advantages and difficulties faced, drawing on an in-depth interview with a senior system engineer at the company. The primary goal is to answer the research question: "What are the benefits and costs if we introduce software containers to the selected company?"

4.2.1 Benefits of Introducing Software Containers

The integration of software containers into the company's operations has not only streamlined processes but also offered strategic advantages that enhance competitiveness in the digital marketplace.

Operational Efficiency and Agility: The company's operations have changed as a result of the deployment of software containers. The senior system engineer emphasized that rollouts have been accelerated by at least 40%. Due to the agility that containers bring, it is often possible to update software quickly and recover quickly from any problems without any downtime.

Financial savings: The introduction of containers has increased hardware efficiency and produced measurable cost reductions. The business has greatly reduced the cost of software licenses and other related expenses by maximizing the use of technology.

Security Improvements: Their container-driven solutions are well-protected because to the thorough automated checks that are triggered with each pull request, ranging from vulnerability scans to penetration testing. The risk of breaches and potential costs is decreased by this thorough security approach.

Versatility and Scalability: The company's capacity to effectively handle demand increases is made possible by containers, as evidenced by their use in the health and wellness tracker project. They are now completely necessary for the company's long-term technological vision due to their adaptability.

4.2.2 The Costs and Challenges of Introducing Software Containers

While the inception of software containers has proven revolutionary for the company, it has not been without its fair share of challenges. The learning process and the need to ensure security and efficiency often come with an associated cost.

Learning curve and skill gaps: Kubernetes' complexity initially presented problems because of its depth and breadth. There were initial costs in terms of time and resources needed on training and upskilling due to the small number of in-house experts conversant with the subtleties of containerization.

Integration with Existing Systems: Migrating to a containerized environment or integrating it with current systems may present a number of technical and economic difficulties. This covers prospective software license modifications, resource reallocations, and system outages.

Security overheads: Although containers brought strong security standards, originally setting up these extensive security measures necessitated investments in terms of resources like tools, labor, and time.

Continual Investments in Training and Professional Development: The company is committed to the professional development of its employees, and the ever-evolving state of container technology necessitates continual investments in training and certifications. This covers the price of various certification programs and websites like Udemy.

4.3 Economic justification of new solution – Cost-Benefit Analysis

In this section, I will provide a Cost-Benefit Analysis (hereafter: CBA), which is crucial for companies considering investments in new technologies. Drawing on insights from a senior systems engineer, I will establish the assumptions for an economic evaluation of a health and wellness tracker project. My aim is to deliver an easily comprehensible economic assessment to assist the company in making informed decisions about their investment.

CBA is a tool used to determine if the financial gains from a project will exceed its costs. To do this, you add up all the potential costs and then subtract them from the total benefits you expect to gain. If the benefits are higher than the costs, the project could be considered worthwhile. Conversely, if the costs are higher than the benefits, it might be wise to reconsider the project. This method can provide crucial financial insights, helping to clarify the potential return on investment (hereafter: ROI) for the organization. Running a CBA before making big decisions can lay out the financial landscape, allowing a company to make better, informed choices (Stobierski, 2019).

With this understanding of CBA, it becomes evident why knowing the projected ROI beforehand is pivotal. Let’s unpack the concept of ROI and how it integrates with our

previous discussion on CBA. Before a project starts, we estimate its expected ROI to decide if it is worth going after. This is done by guessing the costs and potential income and figuring out the profit margin. If a project is ROI is positive, it is a winner because it brings in more money than it costs. But if it is negative, the project ends up losing money because the costs were higher than the income. When a project just breaks even, the money made is exactly what was spent on it (Stobierski, 2020).

In software development, adopting new technologies often requires a comprehensive understanding of both their operational and economic implications. Software containers, with their transformative potential, underscore this need. For a detailed evaluation of the costs and benefits associated with software containers, a CBA becomes very useful. One of the primary advantages of a CBA in this context is its ability to offer clear insights. This means it can provide measurable metrics that highlight the financial advantages and disadvantages of using software containers compared to traditional methods. A thorough CBA not only measures the tangible benefits but also sets the direction for upcoming strategic decisions (Klotins et al., 2022).

Applying a CBA to this sector presents its own set of challenges. The ever-evolving nature of the tech landscape means that rapid advancements in software can soon make the CBA outdated. Furthermore, there are certain benefits associated with software containers, such as the potential to foster innovation or improve team morale, which, despite their significance, are hard to quantify. Lastly, the detailed nature of software deployment can make it difficult to accurately allocate costs and benefits.

A more detailed approach was presented in a 2022 paper by Klotins et al. The paper explains the specific steps of conducting a CBA for software development activities, emphasizing the need for:

1. Identifying the main CI/CD pipeline stakeholders and outlining each group's specific objectives.
2. An in-depth understanding of the cost commitments (costs) and projected advantages related to CI/CD and its components.
3. Setting up helpful metrics to monitor the progress made toward the stated objectives.
4. Technique for calculating and comparing different costs and advantages.
5. Strategy to choose a configuration of ongoing practices that is appropriate for the organization's situation.

Considering these points, the company stands to gain significantly from a detailed CBA. Such an analysis can serve as a guiding light, pointing the way for informed decisions regarding the future potential of container technology. Their initiative in training and collaborations, especially with industry leaders like HP, emphasizes the strategic importance they attribute to container technology. The clear benefits of container implementation are

evident, as seen by improved system uptime, faster project deployments, and a culture that thrives on innovation.

Now, having established the key financial concepts that support our analysis, we will proceed to apply these principles to the health and wellness tracker project, examining the information closely in order to make an informed decision on the project's economic viability.

I will outline assumptions for constructing a CBA using the health and wellness tracker project as an illustrative example. From the insights shared by our senior system engineer during the interview, it was evident that the company extensively uses tools such as GitHub, GitLab, Docker, Kubernetes, Jenkins, AKS, AWS EKS and more. They also invest in training through platforms like Udemy. Interestingly, many of these tools are open-source, but they could potentially incur added expenses due to plugins, specialized support, or premium services.

Considering the company's robust use of software containers and their reliance on GitHub/GitLab and Jenkins, it is reasonable to infer that they had pre-existing licenses or subscriptions for these applications. Furthermore, their consistent utilization of cloud platforms like Azure's AKS and AWS's EKS suggests a possible pre-established licensing or collaborative agreement with these cloud providers.

An important observation to note is the company's transition to Docker, likely requiring the acquisition of a Docker Team license, which is priced at around 300€ annually. The interview also hinted at the company's prior engagement with Docker Swarm, which might not have met their expectations, leading them to adopt Kubernetes. Given their satisfaction with Azure and AWS services, it is logical that they opted for AKS and EKS for their container management needs.

For the health and wellness tracker project, I am making an informed assumption that Azure Kubernetes Service (AKS) was utilized. Azure's official website offers a helpful pricing calculator, allowing for tailored cost estimations tailored to specific project plans. In our context, I have decided to focus on the "CI/CD for Containers" scenario, which seems most aligned with our project needs (Azure, n.d.).

The bundled cost of this scenario includes vital services: ACR, AKS, Azure Monitor, and Azure DevOps, with added support. Initially, the monthly estimated expense comes to approximately €250. However, there is an opportunity to achieve substantial savings by opting for "reserved instances". Azure offers this feature where users can anticipate their resource requirements and make payments in advance, either on a one or three-year commitment basis. This approach not only offers predictability but also a reduced rate. Opting for a three-year reserved instance, for instance, can bring down the monthly costs to around €160. In our financial calculations for this project, I have chosen to work with the

€160 figure, reflecting the long-term, three-year commitment to Azure's reserved instances. This choice is rooted in the project is longevity and our desire for cost-effectiveness.

As they ventured into Kubernetes, employee upskilling became imperative. Udemy became a significant training resource, and if we account for five courses per person, the total number of courses purchased would be 15. However, in the company we usually share these resources amongst ourselves, so I am assuming they did in this case, which means the total cost for the courses is 5 courses * €30 = €150.

Diving deeper into the manpower and associated costs, two full-time DevOps engineers were indispensable from the project is outset. The senior system engineer, our interviewee, contributed approximately 80 hours monthly, amounting to half the standard full-time commitment. Conversely, each DevOps engineer dedicated a solid 160 hours each month. This intense collaboration spanned the two-month migration phase and extended into a 5-year maintenance period, which entailed a more relaxed commitment of 4 hours monthly.

To ascertain the financial implications of these commitments, I turned to a survey from the Facebook group 'Slovenski developerji'. This data suggests that a senior system engineer in Slovenia, primarily operating in the system domain, draws an average gross salary of €4250. In contrast, DevOps engineers receive around €3000 (klele.si, 2023).

So, assumed for the health and wellness tracker project, the financial commitments can be broken down into two main phases: the beginning or the migration phase and the subsequent maintenance phase.

Beginning Phase:

- Senior System Engineer: With a role crucial to the migration process, the senior system engineer was working approximately half of the full-time, equating to 80 hours a month. This operational involvement for two months translated to a financial cost of €4,250.
- DevOps Engineers: Two dedicated DevOps engineers, vital to ensuring the project is success, were actively involved full-time, working 160 hours each per month. Their combined remuneration over the two-month migration period totaled €12,000.
- Docker Team License: Transitioning to Docker, the company procured a Docker Team license. This subscription-based license costs €300 annually.
- Training Costs (Udemy): With a transition to new technologies, employee upskilling is a must. The team procured courses from Udemy, with five courses shared amongst three individuals. This came to a one-time training cost of €150.

Combining the expenses from these elements, the total expenditure for the beginning phase amounts to approximately €16,700.

Maintenance Phase (Over the Next 5 Years):

- Senior System Engineer's Maintenance: Post the migration phase, the senior system engineer devoted 4 hours monthly for ongoing maintenance. This, over a 5-year period, aggregates to a cost of about €6,375.
- DevOps Engineers' Maintenance: The pair of DevOps engineers, equally integral to the project is sustenance, also dedicated 4 hours each monthly for maintenance activities. Their combined commitment over a 5-year period is a cost of €9,000.
- Recurring Docker Team License: The Docker Team license is a recurring annual cost, contributing an additional €1,500 to the maintenance phase over a 5-year period.

Summing up the costs associated with these maintenance tasks, the total ongoing expenditure for the next 5 years is projected to be around €16,875.

Regarding the value of benefits, as seen from the interview, the incorporation of Docker has markedly improved deployment efficiencies. Prior to Docker's integration, the company would allocate roughly 4 hours for a standard deployment. However, this duration was cut by a substantial 40% with Docker, resulting in a 2.4-hour deployment process. Assumption indicated that the expenses associated with each hour of the conventional deployment rounded up to €150.

This comprehensive figure enveloped everything from the salaries of the team members involved to the operational costs of our tech resources, not to mention software licensing and other miscellaneous overheads. Simply put, every hour spared in the deployment process translates to an approximate saving of €150 for the company. To crystallize these figures further, the per-deployment saving can be determined by the hourly cost and the difference in deployment durations. Essentially, the mathematical representation would have us multiplying the €150 hourly rate with the 1.6-hour time difference between the traditional and Docker-enhanced deployment. This calculation gives us a direct benefit of €240 for every deployment.

Expanding this perspective to an annual scale, if we were to conservatively estimate about 50 deployments annually, the total yearly benefit would aggregate to €12,000. This figure, which may seem modest in isolation, accumulates to a significant amount over time. And if these benefits were to be consistent over the next five years, it offers a compelling testament to Docker's economic impact in streamlining our deployment processes.

Here, the discount rate is used to project the anticipated costs and benefits of the project over its lifespan, by converting the value of future revenue into today's worth. The discount rate can vary based on each country, the project is duration, and the type of project in question. In Slovenia, the current discount rate is set at 3.64%. This rate helps in accurately determining the present value of future cash flows, providing a clearer perspective on the project is financial feasibility (European Comission, 2023). Figure 11 shows overview of CBA based on interview and assumptions.

Table 2: Cost-Benefit Analysis

	RMO cost benefit analysis	Year 0 - 2023	Year 1 - 2024	Year 2 - 2025	Year 3 - 2026	Year 4 - 2027	Year 5 - 2028	Total
1	Value of benefits	-	€12.000	€12.000	€12.000	€12.000	€12.000	
2	**Discount factor (3.64%)**	1	0,964	0,93	0,897	0,865	0,835	
3	Present value of benefits	-	€11.568	€11.160	€10.764	€10.380	€10.020	**€53.892**
4	Development costs	€16.700	-	-	-	-	-	**−€16.700**
5	Ongoing costs		€3.375	€3.375	€3.375	€3.375	€3.375	
6	**Discount factor (3.64%)**	1	0,964	0,93	0,897	0,865	0,835	
7	Present value of costs	-	€3.254	€3.139	€3.027	€2.919	€2.818	**€15.157**
8	PV of net of benefits and costs	−€16.700	−€16.700	€8.021	€7.737	€7.461	€7.202	
9	Cumulative NPV	−€16.700	−€33.400	−€25.379	−€17.642	−€10.182	−€2.980	
10	Payback period		**2 years + (-(-364)/7737) = 2 + 0.047 or 2 years and 17 days**					
11	5-year return on investment		**[(€22,035 / (€16,700+€15,157)] * 100% = 69.14%**					

Source: own work.

As visible from Table 2, cumulative Net Present Value (hereafter: NPV) turns from negative to positive between Year 2 (2025) and Year 3 (2026), marking the payback period where the benefits have covered the initial development costs and ongoing costs. The exact payback period is calculated to be approximately 2.047 years into the project, which translates to 2 years and 17 days into the 3rd year. This indicates that the investment in Docker starts to pay off early in the 3rd year.

The 5-year ROI is calculated by dividing the cumulative NPV (€22,035) and the total investment (€31,857), and then multiplied by 100% to express it as a percentage. The ROI comes out to be approximately 69.14%, indicating a positive return on the investment over the 5-year period. This positive ROI suggests that the savings generated from the improved deployment efficiencies have indeed surpassed the total costs incurred over the 5-year span. In addition to this, the positive trend in the cumulative NPV suggests that with more time, the investment in Docker is likely to be even more beneficial, as the benefits continue to accumulate beyond the 5-year mark.

In the case if we did not use Docker in our deployment procedures, the 5-year ROI would surely be lower. This would be result of Docker's notable 40% deployment time reduction, which saves a significant amount of time and money. Without Docker, business would have to deal with longer deployment periods, which would increase operating expenses—especially considering that each deployment hour costs about €150. In addition, integration of Docker optimizes procedures, increasing total effectiveness and output. A less favorable ROI would arise from the lack of these enhancements and the associated cost savings, highlighting Docker's beneficial influence on the company's financial stability and operational effectiveness.

In the health and wellness tracker project as pointed out by senior system engineer, there are also less obvious, yet valuable, advantages to consider. These containers have made processes smoother and our deployment faster. This has naturally led to a better work environment, with teams experiencing fewer mistakes. Such a positive change has made

work more enjoyable and less stressful for everyone involved. It might be hard to put an exact number on these benefits, but they are essential.

These so-called 'intangible benefits' are increasingly significant in assessing a business's true value. Economists now estimate that intangibles like brand recognition and market presence make up over a quarter of a company’s worth. Despite this, many financial decision-makers have not fully embraced this reality. They have seen projects fail that relied too much on these hard-to-measure benefits and have drawn the incorrect conclusion that all intangibles are unreliable. This skepticism results in a critical oversight: when intangibles are not included in the analysis, significant projects may not receive funding because they do not show an immediate, tangible return. On the other hand, less important projects that promise a quick, obvious return—often only because they are smaller in scale—end up getting funded. This method can be short-sighted, as it may overlook strategic objectives that could improve market share or enhance a competitive edge, which are crucial for long-term success (Keen, 2003).

4.4 Possible Improvements of Existing Processes Based on Containers

The business mentality is one of constant improvement. It is possible to further enhance and improve container-related procedures in light of the industry's lengthy history of more than 25 years and its global expansion. I have provided suggestions for enhancements that will considerably help the company's DevOps and containerization efforts, drawing on both my personal experience with the company and their larger operational environment.

Containerized AI & ML: Given the business's interest in AI and ML, concentrate on container solutions designed for these workloads, to provide the best resource allocation and scalability for these demanding jobs.

GDPR Compliance in Containers: Particularly for projects managing the data of European clients, make sure that container orchestration and deployment techniques have built-in procedures to abide by GDPR regulations.

CRM integration: Create container monitoring systems that can integrate with CRMs considering the business's shift into CRM solutions. Better tracking of customer feedback and problem solving would be possible as a result.

Focused Workshops: Conduct in-house workshops emphasizing specific container-related topics, e.g., "Best Practices in Kubernetes Management" or "Advanced Container Security Measures."

External Certifications: Encourage employees to attain container-centric certifications. Consider reimbursing or subsidizing costs for certifications such as the Certified Kubernetes Administrator (CKA).

5 CONCLUSION

Over the course of this thesis, we started on a comprehensive exploration of the integration and optimization of software containers within the organizational framework. This deep dive illuminated the transformative potential of container technologies, revealing their profound implications for operational efficiency, security postures, and innovation capabilities.

Software containers, as we deduced, are not just technological assets but pivotal tools that can reshape the contours of organizational operations. The company's journey, from the initial stages of adoption to mastering the complication of container orchestration, underscored the value of being adaptable in the face of rapid technological evolution. Alongside these technical insights, the organizational strategies around training, collaboration, and metrics emerged as crucial determinants of successful container adoption.

Furthermore, the analysis illuminated challenges that accompany the adoption journey, predominantly the steep learning curve and the imperative for heightened security measures. However, with the right strategies in place, these challenges can be effectively navigated. In terms of operational efficiency, the evidence was conclusive: software containers enhance deployment speed, strengthen resource allocation, and enhance recovery processes. The strategic incorporation of security measures within the DevOps framework highlighted the organization's commitment to protecting its digital assets, while the focus on innovation showcased the company's forward-thinking approach.

The thesis successfully addressed its goals and answered the research question: “What are the benefits and costs if we introduced software containers to the selected company?”. Through our data collection from secondary and primary sources, we have comprehensively analyzed the usage of software containers, underlining both their benefits and the associated costs. The journey of exploration has culminated in a clearer understanding of the significant advantages of adopting software containers.

An initial CBA highlighted the financial benefits achieved through integrating Docker, especially in cutting down deployment times. This resulted in notable savings with each deployment, showing a positive trend in the NPV over time, suggesting a potentially favorable financial outlook. However, the analysis also pointed out areas for improvement to get a clearer understanding of the financial benefits, indicating that with more detailed calculations and ongoing performance checks, the financial gains could possibly be enhanced further.

In retrospect, this thesis has not only mapped the company's journey with software containers but has also charted a course for its future trajectory. The findings presented here provide a robust framework for organizations seeking to optimize their operations using software containers, emphasizing the blend of technical skill with strategic organizational planning.

Despite the comprehensive exploration of software containers and their integration within the organizational framework, this study has certain limitations. It focused on a specific company, and the findings may not universally apply to all organizations due to variations in industries, sizes, and contexts. The dynamic nature of technology introduces temporal considerations, with rapid evolution potentially rendering specifics and recommendations outdated. Generalizing the findings should be approached cautiously, considering diverse organizational structures and cultures.

The financial analysis relied on a preliminary Cost-Benefit Analysis with assumptions, highlighting the need for more granular quantitative data for complex economic insights. External factors such as regulatory changes, market conditions, and global events were not extensively explored. The study acknowledged the importance of heightened security measures but did not delve deeply into compliance intricacies specific to industry or region. Recognizing these limitations emphasizes the necessity for ongoing research, continual monitoring of industry trends, and a flexible approach to adapting strategies as technology and organizational landscapes evolve.

To conclude, software containers are more than a technological trend; they represent a paradigm shift in how organizations perceive and execute their software development and deployment strategies. As the digital landscape continues to evolve, companies that embrace, adapt, and innovate will undoubtedly lead the charge, setting new benchmarks for excellence in the industry.

REFERENCE LIST

1. Agilemanifesto.org. (2001). *Manifesto for Agile Software Development.* https://agilemanifesto.org/principles.html
2. AWS. (2023). *What is Docker?* https://aws.amazon.com/docker/
3. Azad, N. (2022). Understanding DevOps critical success factors and organizational practices. *2022 IEEE/ACM International Workshop on Software-Intensive Business (IWSiB)*, 83–90. https://doi.org/10.1145/3524614.3528627
4. Bahadori, K., & Vardanega, T. (2019). DevOps meets dynamic orchestration. In J.-M. Bruel, M. Mazzara, & B. Meyer (Eds.), *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 142–154). Springer International Publishing. https://doi.org/10.1007/978-3-030-06019-0_11
5. Bagaria, R. (2018, October 22). *Seven best practices for Continuous Monitoring with Azure Monitor* [published on blog]. https://azure.microsoft.com/en-us/blog/7-best-practices-for-continuous-monitoring-with-azure-monitor/
6. Battina, D. S. (2021). The challenges and mitigation strategies of using DevOps during Software Development. *Journal of Software Engineering Research and Development*, *9*(1), 4760-4765

7. Bentaleb, O., Belloum, A. S. Z., Sebaa, A., & El-Maouhab, A. (2022). Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing, 78*(1), 1144–1181. https://doi.org/10.1007/s11227-021-03914-1
8. Bhartiya, S. (2017). *Open source leaders: Solomon Hykes and the Docker revolution.* https://thenewstack.io/solomon-hykes-leader-open-source-world-needs/
9. Bird, M.S. (2010). *Utilizing agile software development as an effective and efficient process to reduce development time and maintain quality software delivery* (doctoral dissertation). Capella University.
10. Burns, B. (2018, July 20). *The History of Kubernetes & the Community Behind It* [published on blog]. https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/
11. Carey, S. (2021). *What is Docker? The spark for the container revolution.* https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html
12. Chen, G. (2018). *The rise of the enterprise container platform.* https://www.diaxion.com/wp-content/uploads/2018/10/IDC-containerplatform-wp.pdf
13. CNCF. (2023). *CNCF Annual Survey 2022.* Cloud Native Computing Foundation. https://www.cncf.io/reports/cncf-annual-survey-2022/
14. Cois, C. A., Yankel, J., & Connell, A. (2014). Modern DevOps: Optimizing software development through effective system interactions. *2014 IEEE International Professional Communication Conference (IPCC)*, 1–7. https://doi.org/10.1109/IPCC.2014.7020388
15. de la Torre, C., Wagner, B., & Rousos, M. (2023*). .NET Microservices. Architecture for containerized .NET applications.* https://learn.microsoft.com/en-us/dotnet/architecture/microservices/
16. de la Torre, C. (2022). *Containerized Docker application lifecycle with Microsoft platform and tools.* https://learn.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/
17. Docker. (2023). *Deploying Docker containers on Azure*. https://docs.docker.com/cloud/aci-integration/
18. Erich, F. (2019). DevOps is simply interaction between development and operations. In J.-M. Bruel, M. Mazzara, & B. Meyer (Eds.), *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 89–99). Springer International Publishing. https://doi.org/10.1007/978-3-030-06019-0_7
19. European Commission. (2023). *Reference and discount rates.* https://competition-policy.ec.europa.eu/state-aid/legislation/reference-discount-rates-and-recovery-interest-rates/reference-and-discount-rates_en
20. Galin, D. (2018). *Software quality: Concepts and Practice: Concepts and Practice* (1st ed.). Wiley. https://doi.org/10.1002/9781119134527
21. Garg, S., & Garg, S. (2019). Automated cloud infrastructure, continuous integration and continuous delivery using Docker with robust container security. *2019 IEEE Conference*

on Multimedia Information Processing and Retrieval (MIPR), 467–470. https://doi.org/10.1109/MIPR.2019.00094

22. Gonzalez, A. E., & Arzuaga, E. (2020). HerdMonitor: Monitoring live migrating containers in cloud environments. *2020 IEEE International Conference on Big Data (Big Data)*, 2180–2189. https://doi.org/10.1109/BigData50022.2020.9378473
23. Hall, T. (2023). *4 key DevOps metrics to know.* https://www.atlassian.com/devops/frameworks/devops-metrics
24. Hatheway, R. (2021, November 10). *Why Enterprise IT Organizations Will Benefit from Application Containerization*. https://www.cio.com/article/189567/why-enterprise-it-organizations-will-benefit-from-application-containerization.html
25. Hecht, L. E. (2015). *How open source communities power Docker and the container ecosystem.* https://thenewstack.io/open-source-communities-define-docker-container-ecosystem/
26. Hilaly, A. (2014). *Docker and the rise of open source.* https://www.linkedin.com/pulse/20141118181806-131990-docker-and-the-rise-of-open-source/
27. Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation.* Addison-Wesley.
28. Huawei Technologies Co., Ltd. (2023). *Cloud Computing Technology* (1st ed.). Springer Nature. https://doi.org/10.1007/978-981-19-3026-3
29. IBM. (2023). *What is software development?*. https://www.ibm.com/topics/software-development
30. Israel. (2023). *Open source software: Contributions, challenges, and future prospects* [published on blog]. https://bootcamp.uxdesign.cc/open-source-software-contributions-challenges-and-future-prospects-%EF%B8%8F-d8a9f96fa726
31. Jacobs, M. (2023, January 25). *What is DevOps? Azure DevOps.* https://learn.microsoft.com/en-us/devops/what-is-devops
32. Jayathilaka, C. (2020, July 30). *Agile methodology* [published on blog]. https://medium.com/@chathmini96/agile-methodology-30ec4cdf3fc
33. Jones, C. (2019). A proposal for integrating DevOps into software engineering curricula. In J.-M. Bruel, M. Mazzara, & B. Meyer (Eds.), *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 33–47). Springer International Publishing. https://doi.org/10.1007/978-3-030-06019-0_3
34. Keen, J. (2003, September 1). *Intangible benefits can play key role in business case.* https://www.cio.com/article/267233/it-organization-intangible-benefits-can-play-key-role-in-business-case.html
35. Khong, L., Yu Beng, L., Yip, T., & Soofun, T. (2012). Software Development Life Cycle AGILE vs Traditional Approaches. *Conference: International Conference on Information and Network Technology, Chennai, India.* 37(1), 162–167.

36. Klotins, E., Gorschek, T., Sundelin, K., & Falk, E. (2022). Towards cost-benefit evaluation for continuous software engineering activities. *Empirical Software Engineering*, 27(6), 157. https://doi.org/10.1007/s10664-022-10191-w
37. Kothari, C. R. (2009). *Research methodology: Methods and techniques* (2nd ed). New Age International Publishers.
38. Malviya, K. (2020). *Kubernetes: The evolution of a technology* [published on blog]. https://medium.com/swlh/kubernetes-the-evolution-of-a-technology-revolution-805302172ea6
39. Maynard, C. (2023). *Software testing in continuous delivery.* https://www.atlassian.com/continuous-delivery/software-testing
40. McCormick, M. (2021). Software development life cycle. In *The Agile Codex: Re-inventing Agile Through the Science of Invention and Assembly* (pp. 79–85). Apress. https://doi.org/10.1007/978-1-4842-7280-0_13
41. Medven, M. (2022) *Plače slovenskih razvijalec 2022*. https://klele.si/place/2022
42. Mell, E. (2023). *The evolution of containers: Docker, Kubernetes and the future*. https://www.techtarget.com/searchitoperations/feature/Dive-into-the-decades-long-history-of-container-technology
43. Mens, T., & Demeyer, S. (2008). *Software evolution.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-76440-3
44. Microsoft Azure. (n.d.). *Pricing overview—How Azure pricing works.* https://azure.microsoft.com/en-us/pricing/
45. Mishra, A. (2013). A comparative study of different software development life cycle models in different scenarios. *International Journal of Advance Research in Computer Science and Management Studies, 1*, 64–69.
46. Najihi, S., Elhadi, S., Abdelouahid, R. A., & Marzak, A. (2022). Software testing from an Agile and traditional view. *Procedia Computer Science, 203*, 775–782. https://doi.org/10.1016/j.procs.2022.07.116
47. Nduta, A. (2023). *A brief history of open source* [published on blog]. https://www.freecodecamp.org/news/brief-history-of-open-source/
48. Nortal. (2018, March 20). *Containers: The software development lifecycle's last mile*. Nortal. https://nortal.com/insights/containers-the-software-development-life-cycles-last-mile/
49. Osnat, R. (2020). *A brief history of containers: From the 1970s till now* [published on blog]. https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016
50. Pittet, S. (2021). *Continuous integration vs. Delivery vs. Deployment*. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment
51. Raj, P., Chelladhurai, J. S., & Singh, V. (2015). *Learning Docker: Optimize the power of Docker to run your applications quickly and easily*. Packt Publishing.
52. Raj, P., & Raman, A. (2018). Automated multi-cloud operations and container orchestration. In P. Raj & A. Raman (Eds.), *Software-Defined Cloud Centers:*

Operational and Management Technologies and Tools (pp. 185–218). Springer International Publishing. https://doi.org/10.1007/978-3-319-78637-7_9

53. Red Hat. (2022). *Red Hat OpenShift vs. Kubernetes: What is the difference?* https://www.redhat.com/en/technologies/cloud-computing/openshift/red-hat-openshift-kubernetes
54. Rehkopf, M. (2023). *What is continuous integration.* https://www.atlassian.com/continuous-delivery/continuous-integration
55. Sarmiento, E. M. (2020). *The SQL Server DBA's Guide to Docker Containers: Agile Deployment without Infrastructure Lock-in.* Apress. https://doi.org/10.1007/978-1-4842-5826-2
56. Sekgweleo, D. T. (2019). Comparing Agile and traditional system development methodologies. *IJIRAS JOURNAL*, *6*(5).
57. Sekgweleo, T., & Iyamu, T. (2022). Understanding the factors that influence software testing through moments of translation. *Journal of Systems and Information Technology*, *24*(3), 202–220. https://doi.org/10.1108/JSIT-07-2021-0125
58. Singh, S., & Singh, N. (2016). Containers & Docker: Emerging roles & future of cloud technology. *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 804–807. https://doi.org/10.1109/ICATCCT.2016.7912109
59. Stobierski, T. (2019). Cost-benefit analysis: What it is & how to do it. *Harvard Business School Online.* https://online.hbs.edu/blog/post/cost-benefit-analysis
60. Stobierski, T. (2020). How to calculate ROI to justify a project. *Harvard Business School Online*. https://online.hbs.edu/blog/post/how-to-calculate-roi-for-a-project
61. Strotmann, J. (2016, December 13). Infographic: *A brief history of containerization* [published on blog]. https://www.plesk.com/blog/business-industry/infographic-brief-history-linux-containerization/
62. Tamburri, D. A., Di Nucci, D., Di Giacomo, L., & Palomba, F. (2019). Omniscient DevOps analytics. In J.-M. Bruel, M. Mazzara, & B. Meyer (Eds.), *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 48–59). Springer International Publishing. https://doi.org/10.1007/978-3-030-06019-0_4
63. Utekar, A. (2022). *Why Docker was built* [published on blog]. https://www.linkedin.com/pulse/why-docker-container-built-ankit-utekar/
64. Vaughan-Nichols, S. (2014). *Survey indicates four out of five developers now use open source.* https://www.zdnet.com/article/survey-indicates-four-out-of-five-developers-now-use-open-source/
65. Walli, S. (2017). *Demystifying the Open Container Initiative (OCI) specifications. Docker* [published on blog]. https://www.docker.com/blog/demystifying-open-container-initiative-oci-specifications/
66. Weave. (2017, January 17). *Cloud Provider Options for a Container-Friendly Environment* [published on blog]. https://www.weave.works/blog/cloud-provider-options-container-friendly-environment/

67. Westfall, L. (2016). *The Certified Software Quality Engineer Handbook* (2nd ed.). ASQ Quality Press.
68. Yadav, M. P., Pal, N., & Yadav, D. K. (2021). A formal approach for Docker container deployment. *Concurrency and Computation: Practice and Experience*, *33*(20), e6364. https://doi.org/10.1002/cpe.6364

APPENDICES

Appendix 1: Summary of the thesis in Slovene Language

Avtomatizacija v tehnologiji ima zdaj pomembno vlogo v številnih dejavnostih podjetja in je postala sestavni del našega vsakdana. Podjetjem na primer pomaga pri učinkovitejšem delovanju in povečuje zadovoljstvo potrošnikov (Najihi et al., 2022). Digitalizacija poslovanja s ciljem avtomatizacije ima potencialne koristi v vseh sektorjih, pri čemer panoga informacijske tehnologije (v nadaljevanju: IT) ni izjema.

Zelo zapletena kombinacija aplikacij, računalniških sistemov in postopkov je temelj večine programskih podjetij kot enega največjih delov IT sektorja. Večje kot je podjetje, težje je integrirati tehnologije. Življenjski cikel razvoja programske opreme opisuje, kako so izdelki programske opreme načrtovani, izvedeni in pakirani za dostavo (McCormick, 2021).

Posel dostave programske opreme se je zaradi nenehne analize in spreminjanja naših metod dostave razvil v zelo praktičen poklic. Avtomatiziran in sofisticiran cevovod za gradnjo, ki razvijalcem omogoča izboljšanje povratnih informacij na vseh stopnjah življenjskega cikla razvoja programske opreme, je nadomestil prejšnjo prakso v industriji gradenj programske opreme na namenskih računalnikih posameznih razvijalcev. Za povečanje zmožnosti dostave programske opreme na avtomatiziran način je pomembno implementirati in sprejeti koncept, imenovan razvojne operacije (v nadaljevanju: DevOps) (Azad, 2022).

Koncept DevOps temelji na integracijskih procesih in tehnikah za aplikacije poslovne programske opreme, ki jih je treba nenehno integrirati in dobavljati, ter povezana orodja, da se zmanjšajo operativni stroški IT, hkrati pa se poveča kakovost, stabilnost in hitrost programske opreme na trg. DevOps načeloma spodbuja bolj poglobljeno osredotočenost na uspešnost in rezultate (Bahadori & Vardanega, 2019).

Po uspešnem sledenju metodologiji razvoja programskih produktov se proces razvoja programske opreme v podjetjih nadaljuje z integracijo in uvedbo programske opreme. Preprosto vzame vse stvari, ki so bile razvite, in jih pripelje na pravo mesto. Njegov cilj je biti sistem, ki gladko distribuira vse dobrine, ustvarjene kot posledica različnih sprintov. Cilj DevOps je popolnoma avtomatizirati postopek in spremljati vsak posamezen dogodek, ki se zgodi med različnimi stopnjami uvajanja in integracije programske opreme. Za hitro izdelavo programske opreme s kratkimi dobavnimi cikli sta neprekinjena dostava in uvajanje temeljni komponenti DevOps v podjetju (Azad, 2022). Ko je DevOps pravilno uveden in integriran v programska podjetja, so vrzeli med operativnimi in razvojnimi ekipami zmanjšane.

Tehnologija kontejnera programske opreme je koncept, ki zelo olajša te naloge, še posebej, če upoštevamo neprekinjeno dostavo in stalno integracijo kot ključni komponenti katerega koli današnjega projekta razvoja programske opreme. To je široko uporabljena tehnologija v ekosistemih računalništva v oblaku in IT, ki učinkovito ločuje vire posameznega

operacijskega sistema v ločene skupine, da pravičneje uravnoteži konkurenčne zahteve po uporabi virov izoliranih skupin (Huawei, 2023, str. 295).

Kontejnerizacija vključuje združevanje programa skupaj z vsemi njegovimi odvisnostmi, kot so izvajalno okolje, sistemski pripomočki, sistemske knjižnice in nastavitve aplikacije, v kompakten samostojen izvedljiv paket. Ker je vse, kar je potrebno za izvajanje programa, v kontejneru, ga je mogoče prenesti iz fizičnega računalnika v virtualni stroj ali v javni ali zasebni oblak, v bistvu katero koli okolje, ne da bi pri tem tvegali, da bi se povezali z operacijskim sistemom dejanski stroj. Prav to je največja prednost zabojnikov, ki jo bomo v nadaljevanju obravnavali in primerjali v diplomskem delu. Eden najbolj priljubljenih kontejnerov je Docker, ki je odprtokodna platforma in platforma, ki omogoča kontejnerizacijo za učinkovit razvoj, delovanje, integracijo in uvajanje aplikacij (Raj & Raman, 2018).

Pri večjih razmestitvah aplikacij je lahko več kontejnerov razporejenih kot ena ali več gruč kontejnerov. Takšne gruče mora nadzorovati nekakšno orodje za upravljanje kontejnerov ali orkestrator kontejnerov, kot je Kubernetes. Odprtokodni sistem avtomatizira razširljivost, uvajanje in upravljanje aplikacij v kontejnerih. Za enostavno upravljanje in odkrivanje razdeli kontejnere, ki sestavljajo aplikacijo, na logične dele. Visoke ravni interoperabilnosti, samopopravljanja, avtomatiziranih uvedb in povrnitev nazaj ter orkestracije shranjevanja so vse značilnosti Kubernetesa. Kar zadeva samodejno reševanje težav, se Kubernetes odlikuje. Dejstvo, da se zabojniki lahko tako hitro zrušijo in znova zaženejo, pomeni, da se sploh ne bomo zavedali, kdaj se naši zabojniki zrušijo (Raj & Raman 2018).

Največja težava v izbranem podjetju je nezadostna izkoriščenost prednosti kontejnerjev v IT poslovanju. Predlagal bom izboljšave za ključna področja, kjer bi lahko s kontejnerizacijo izboljšali proces razvoja programske opreme podjetja, s primerjavo trenutnih in predlaganih. Želim opozoriti na problematiko in težave, s katerimi se sooča podjetje, ter podati predloge za spremembe, ki bodo omogočale hitrejši, učinkovitejši in natančnejši razvojni proces. Rezultati bodo zanimivi tudi za širšo javnost, vključno z drugimi znanstveniki in podjetji, ki se ukvarjajo z enakimi ali primerljivimi problemi.

Namen magistrskega dela je prispevati k razumevanju vpliva programskih kontejnerov, raziskovati tako stroške, ki jih imajo podjetja, kot potencialne koristi, ki izhajajo iz njihove integracije v razvojne procese trenutnih ali prihodnjih aplikacij. Diplomsko delo skuša razjasniti, kako kontejneri programske opreme prispevajo k aktivnostim v podjetju glede življenjskega cikla razvoja programske opreme in delovanja.

Cilji magistrskega dela je, da z zbiranjem podatkov iz različnih sekundarnih virov analiziramo možnosti uporabe programskih kontejnerov. S temi informacijami bomo lahko primerjali rezultate in ugotovili razloge za njihovo uvedbo. Še več, cilji diplomske naloge razširiti na zbiranje podatkov iz primarnih virov, kot je intervju z inženirjem IT, da bi se od njega neposredno naučili o prednostih in izzivih uporabe kontejnerov programske opreme v

določenih kontekstih. Vendar zadnji cilj je povzetek glavnih prednosti in izzivov kontejnerov programske opreme ter predlaganje ukrepov za premagovanje teh izzivov.

Raziskovalno vprašanje magistrskega dela je: »Kakšne so koristi in stroški, če v izbrano podjetje uvedemo programske konterjnere?«.

Za raziskavo v diplomski nalogi sem uporabil kombinacijo raziskovalnih metod. To poleg kvantitativnih metod vključuje kvalitativne metode, kot so pregled obstoječe literature, opazovanje in opravljanje poglobljenih intervjujev z inženirjem informatike. To vključuje analizo podatkov podjetja, ki temelji na predpostavkah in se izvaja po implementaciji programskih kontejnerov. Integracija teh metod omogoča analizo tako empiričnih kot izkustvenih vidikov sprejemanja kontejnera programske opreme v izbranem podjetju. Poleg tega sem opravil analizo stroškov in koristi. Z združevanjem informacij iz teh različnih metodologij želi raziskava oblikovati popolno sliko o vplivu kontejnerov programske opreme na delovanje IT.

To magistrsko delo je strukturirano v treh delih. Glavni cilj prvega dela je zagotoviti trdno teoretično osnovo za kontejnere programske opreme. Diplomsko delo bo sistematično osvetlilo ključne ideje, trenutne trende in se podrobno podalo v podrobnosti življenjskega cikla razvoja programske opreme s temeljitim pregledom literature, ki je temelj sekundarnega raziskovanja. Pristopi, ki se uporabljajo za dostavo programske opreme, specifike, vključene v njeno upravljanje, in povezava med kontejnerizacijo in storitvami v oblaku je le nekaj pomembnih vprašanj, ki bodo obravnavana v tem delu.

Drugi del se osredotoča na raziskovalno metodologijo, uporabljeno v tej študiji. V tem razdelku bomo podrobneje opisali raziskovalne metode s poudarkom na intervjuju z odgovornim inženirjem IT. Dva cilja tega intervjuja sta razumeti operativne spremembe, ki so sledile uvedbi kontejnerov programske opreme skozi čas, kot tudi pridobiti vpogled v upravičenost njihove uporabe. V tem kontekstu bodo obravnavana merila in logika, uporabljena pri izbiri akademskega gradiva, kot so pomembni učbeniki in pomembni znanstveni članki s tega področja.

Zadnji in tretji del se poglobi v ugotovitve študije. Inženir IT bo predstavil svoj pogled na njihove izkušnje z uporabo kontejnerov programske opreme v razdelku z naslovom "Analiza intervjuja". Pregled sedanjega poslovnega okolja bo na voljo v "Analizi sedanjih poslovnih procesov", ki mu bodo sledila priporočila, kako lahko kontejneri programske opreme vodijo k optimizaciji.

Appendix 2: Interview with Senior System Engineer

Question 1: How is your company currently implementing software containers, and what role do you envision for them in future technology plans?

At the heart of our operations, software containers play an integral role. They serve as our primary method for deploying microservices. As we look towards our strategic planning, it is evident that containers are not just another tool in our toolbox. We foresee them as the cornerstone of our future technological projects and endeavors. Their versatility and scalability make them indispensable for our roadmap.

Question 2: Can you discuss the benefits your company has experienced with software containers, particularly in terms of improving efficiency or productivity?

Since embracing containers, our operations have witnessed a transformative change. We have tapped into improved hardware efficiency, which has been a boon for our budgets. The tangible savings aren't just monetary, though. The streamlined software deployment processes have substantially reduced time-consuming rollouts for at least 40%. More importantly, in situations where systems might fail, our recovery process has become quicker, ensuring minimal interruptions and, quite frequently, no downtime at all.

Question 3: What challenges or drawbacks have you faced in using software containers, and how have these been addressed?

Like all transformative technologies, integrating container technology had its initial obstacles. Kubernetes, in particular, posed challenges due to its depth and breadth. With few in-house experts familiar with the nuances of containerization, we treated cautiously. We started by integrating containers into less complex projects, allowing our team to build competence progressively. Over time, and with accumulated hands-on experience, we have grown confident and adept in our containerization ventures.

Question 4: How does your company ensure the security of its software containers, and are there any best practices or guidelines that you follow?

Security remains paramount in our operations. We have seamlessly embedded the principles of DevOps and DevSecOps into our workflows. Whenever any pull request emerges in our GitHub or GitLab, a complex web of automated checks spring into action. This comprehensive vetting spans automatic tests, vulnerability probes, and even robust penetration tests. By implementing these measures, we ensure the sanctity and security of our container-driven solutions.

Question 5: Have you observed any significant trends or changes in the adoption of software containers within your industry?

Keeping a finger on the pulse of the industry, we have observed a noticeable shift. Kubernetes, with its feature-rich offerings, is rapidly becoming the preferred container orchestration platform. What sets it apart from systems like Docker Swarm are its capabilities such as self-healing, nuanced load balancing, and robust auto-scaling. Moreover, its built-in policies, as seen in tools like Kyverno, ensure deployments are standardized and efficient.

Question 6: Can you share specific use cases or projects where software containers have been particularly effective in your organization?

Our engagement with containers isn't theoretical; it is deeply practical. Using platforms like Azure (through AKS) and AWS (via EKS), we have been able to harness the full potential of their container management systems. A standout project is our health and wellness tracker project, aiding users in monitoring fitness activities and dietary habits. Thanks to container technology, even during periods of surging demand, our systems remain resilient, effectively balancing loads and scaling to meet requirements. Deployment speed of this application was increased by 40% by using Docker.

Question 7: Can you give us some information on the team working alongside you on health and wellness tracker project, particularly in regard to migration and maintenance?

The health and wellness tracker project was a collaborative effort. While we had developers and other specialists involved, I was leading two dedicated DevOps engineers, they were crucial in transitioning to a container-based infrastructure. They played an essential role in monitoring and maintaining the system after its launch, ensuring the application's stability and performance.

Question 8: Considering the timeline and the intensity of the health and wellness tracker project, how were the working hours distributed among your team?

The migration lasted two months. I was dedicating roughly half of the working day to ensure that everything was on track in regard of meetings between developers and my team, and to provide guidance where needed. Two mentioned DevOps engineers were completely immersed in the project working full-time to ensure the migration and implementation. Maintenance and monitoring take them about 4 hours a week for a lifetime of a project.

Question 9: What tools or technologies does your company rely on for managing and deploying software containers?

Our technological infrastructure comprises a blend of tools tailored for modern needs. GitHub or GitLab, depending on the use, remains our foundation for version control. Jenkins orchestrates our DevOps processes. While Kubernetes is our go-to for deployment, we are continually exploring and testing additional tools, ensuring we are always equipped with the best in class.

Question 10: What types of training or support are provided to your employees for effective use of software containers?

Investing in our team's growth is a priority. We actively promote and support their journey towards earning certifications, especially those synergistic with our core projects. Online learning platforms, like Udemy, are accessible to all, fostering an environment where learning and professional development are constant endeavors.

Question 11: Do you have any collaborations or partnerships with other organizations or vendors in the field of software containers?

In our journey with container technology, we are not alone. HP, a major player in the tech world, stands out as our premier partner. Our collaborative projects, rooted in container technology, have deepened our shared knowledge and expertise in this domain.

Question 12: How does your company measure the success of its software container implementations, and can you provide examples of how containers have been used to innovate or differentiate the company in the market?

Measuring the impact of our container strategies isn't just about numbers. It is about tangible outcomes. Enhanced system uptime, faster rollouts, and a uniform approach to infrastructure management stand out as markers of success. Containers have also revolutionized our development dynamics. By enabling multiple teams to work in parallel, using their preferred tech stacks, we have unlocked a level of flexibility that serves as a catalyst for innovation.