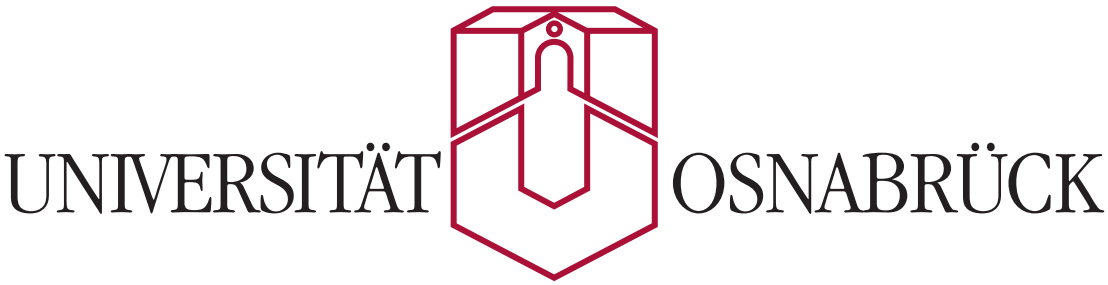

Institute of Computer Science

B.Sc. Thesis

Analysis of Adopting DevOps Tools for a Homogeneous Production and Development Environment

---

Henrik Gerdes
MatNr: 969272

Supervisor
Prof. Dr.-Ing. Elke Pulvermüller
Dipl.-Wirt.-Inf. Achim Hendriks

November 1, 2021

Abstract

Deutsch: Das Wachstum von Cloud Computing und die Möglichkeit zum schnellen, automatischen Bereitstellen von skalierbaren Anwendungen führt zur stetig steigenden Beschleunigung für die Verfügbarkeit von Softwarelösungen. Die Unabhängigkeit von Hardware und Betriebssystem wird dabei durch Containerisierung, einer Form der Virtualisierung, erreicht. Während dies zum Standard für den produktiven Betrieb von Software in der Cloud geworden ist, wird während des Entwicklungsprozesses weiterhin auf die lokale Installation und Konfiguration von individuellen Entwicklersystemen gesetzt, welche manuelle, zeitintensive Konfiguration benötigen, wodurch es zur erhöhten Fehleranfälligkeit kommt. Diese Arbeit schlägt den Einsatz von Containerisierung für den Entwicklungsprozess von Software vor. Diese Entwicklungscontainer (DevContainer) nutzen Prozessisolierung um eine schnellere, weniger fehleranfällige, homogene Entwicklungslandschaft bereitzustellen. Das konzeptionelle Design einer solchen Entwicklungsumgebung wird beschrieben und anschließend mittels einer exemplarischen Umsetzung auf Anwendbarkeit überprüft. Die daraus gewonnenen Erkenntnisse werden mit kommerziellen Alternativen verglichen und zeigen das Potenzial von DevContainer für effizientere Entwicklungsprozesse.

English: The growth of cloud computing and the possibility of rapid, automatic deployments of scalable applications is leading to an ever-growing acceleration in the availability of software solutions. Thereby, containerization a form of virtualization offers independence form hardware and operating system. While this approach has become the default in the cloud for offering software solutions, the development process continues to rely on local configuration and operation on individual systems. These systems require manual, time-intensive configuration, leading to increased error-proneness. This thesis proposes the use of the containerization concept for the development process of software. These development containers (DevContainers) use process isolation to provide a faster, less error-prone and homogeneous development landscape. The conceptual design of such a development environment is described, and then exemplary implemented in order to review the applicability. The resulting findings are compared to commercial alternatives and the potential of DevContainers for more efficient development processes are stated.

Contents

1 Introduction

Digital service solutions are becoming more and more relevant as a result of the ever-increasing possibilities and availability of technology. New remote working tools, digital education resources, a connected healthcare system and seamless (video) communication systems are needed. The overall revenue in the software market is expected to grow from $532 billion (2020) to $772 billion in 2025 [Sta21d]. The need for digital business solutions is bigger than ever before. Just as the availability of on-demand computing capacity, which is greater than ever before thanks to the ever-growing cloud platforms, such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud. Services can be made available quickly across the world without the need to build up an infrastructure on site. To provide fast and suitable software solutions, developers need adequate development setups, called development environments. This thesis analyses modern, agile software development environments, points out potential problems and purposes a virtualized software development solution to improve development efficiency. This solution is implemented on a real project and then evaluated.
The following section will briefly describe the fundamental question of this thesis, point out its relevance and presents a possible solution to the problems found. Subsequently, a clear delineation is given as to what is and what is not covered in this thesis.

1.1 Problem Description

The rise of agile development and microservices has been accelerated by the emergence of new technologies changing the way *where* and *how* software is running. Applications could be distributed and scaled quickly through cloud computing, to the liking of customers, favoring for quick changes. Yet, the actual coding setup has not changed significantly.
Technical requirements for projects and their development environments differ depending on their purpose. The development process of Graphical User Interface (GUI) applications for PCs and smartphones apps is quite different to the development of web-services. As the functionality of web services continues to grow, their adoption is a cost-effective, economical way to deliver cross-platform products, and it is becoming increasingly popular. However, the operating system used for the development process is often different from the operating system used to run these applications in production. This can cause platform-dependent errors. Managing the runtime versions of multiple programming languages between team members or different projects becomes a challenge, as these can lead to unexpected program behavior or result in library version conflicts. The usage of a microservice architecture poses further problems, due to its scalability, adaptability and rapid development. Microservices require additional configuration effort and increase the difficulty of whole system tests. The initial setup for new developers can get quite complex, requires time and might even discourage developers to contribute to open source projects.
These characteristics add additional effort, can introduce new errors and slow down the development speed, resulting in higher costs and a lower customer experience.

1.2 Goal of the Thesis

The aforementioned problems were also encountered by Symbic GmbH, in whose cooperation this thesis is being produced. In further detail, the aforementioned challenges and obstacles of modern software development environments are presented in detail and their cause is identified. Based on these findings a solution concept for these challenges is created, by using virtualization technologies. These allow to abstract the operating system used and provide greater similarity between development and productive environments. The technologies used are explained, and the developed concept is then applied to a real project at Symbic. Subsequently, the practicability of the solution is evaluated and classified. In principle, the goal of this thesis is to identify obstacles in current software development setups and to analyze the effectiveness of the proposed solution.

1.3 Scope of the Thesis

Different software development environments cover a wide range of tools, which differs significantly depending on the project. Since the development of web-based solutions is becoming more and more popular, priority is given to these types of projects only. Native application development for PCs and smartphones, as well as the development of embedded systems and other hardware-related solutions, are not covered by this thesis.
In particular, the concept shown is not a general solution. It is intended only as a solution template that contains many ways for dealing with the problems described in section 3. Although section 2 offers an explanation of fundamental topics, deeper insight into the technologies used is given in section 4.3. Nevertheless, not all basic concepts can be explained in depth. Basic knowledge of the software development process, essential programs and an abstract understanding of different operating systems and network techniques are recommended.

1.4 Structural Overview

The following section provides overall background information, which are helpful for further understanding. Common and modern development methodologies are described in section 2.1, followed by microservice and container basics in section 2.2. Section 3 will be a detailed analysis of current development setups and the problems encountered in these setups. The problems are illustrated with examples and their consequences for the development process. Section 4 proposes a conceptual solution, defines its use cases and workings while also providing limitations for such a solution. This concept is then applied to a practical reference project in section 5. The implementation process is described and additional challenges and handy tools for the practical use are presented. Section 6 discusses the proposed solution and shows its strengths and weaknesses. As to conclude section 7 summarizes the key findings of this thesis and gives an outlook on future application areas of the concept presented.

2 Background Information

The following section provides some basic definitions and principles recommended for the further understanding of this paper and ensures a common level of knowledge of these topics. In the process, modern software mythologies are presented and defined, followed by a brief description of the emergence and its current adoption. Next, the microservice software architecture is presented, and the tools frequently used with it. The connection of these tools with the DevOps culture is then clarified.

2.1 Modern Software Methodologies

This section contains formal definitions of the agile software development paradigm and the DevOps methodology, explains their connection and points out the fundamental principles of a DevOps enabled culture. These principles and their workflows will be revisited in Section 4 as part of the proposed concept.

2.1.1 Agile Development

Agile, such as the *V-Model*, *Waterfall* and *Prototyping* model, is a software development paradigm. It was proposed and popularized by the *"Manifesto for Agile Software Development"*, written and published in 2001 by various authors [BBVB+01]. Rapid adoption to changes, continuous evolution of software and customer communication are the fundamentals of agile software development.
The four core principles are:

- ⋆ **Individuals and interactions** over processes and tools
- ⋆ **Working software** over comprehensive documentation
- ⋆ **Customer collaboration** over contract negotiation
- ⋆ **Responding to change** over following a plan

Paradigms like Waterfall describe a comprehensive model with an in-depth analysis of requirements and detailed architecture design phase. This results in a fixed and exact sequential schedule for the implementation and testing phase as well as the phase in which the requirements fulfilment is verified. Errors in the requirements analysis or changes in the requirements can cause difficulties in later phases of the project. Agile on the other hand only provides guidelines instead of a complete model. Changes are expected and the project realization is designed to be adaptable. Project phases such as implementation and testing run simultaneously and the entire Software Development Life Cycle (SDLC) is shorter and more adaptable [SB08].
Agile provides a mindset for projects with uncertain or continuously changing requirements. With the increase of software solutions in rapidly changing markets, software requirements also became more uncertain. Accordingly, the agile manifesto became the foundation

Step		Description
1	**Coding**	Code development & review and source control.
2	**Building**	CI build and build status.
3	**Testing**	CI testing and testing feedback.
4	**Packaging**	Bundle and package to a central registry.
5	**Releasing**	Release management, approvals and automation.
6	**Configuring**	Infrastructure configuration and management.
7	**Monitoring**	Applications performance monitoring.

Table 1: Common DevOps Workflow Steps.
Source: [BWZ15]

of several other software development methodologies and frameworks that extend these fundamental guidelines and provide workflows and tooling for the software creation process. Well known exemplary methodologies are *SCRUM*, *Extreme Programming (XP)* and *DevOps* [SB08].

2.1.2 Definition of DevOps

DevOps is a set of practices in software development that aims to increase customer value and software quality by shortening the development life cycle through active collaboration and continuous delivery of improvements [BWZ15], [DD16]. The term DevOps is a neologism of development (Dev) and operation (Ops). The combination of these terms is symbolic for the tighter collaboration between the development and operations team, which were strictly separated beforehand. Therefore, DevOps is considered more than just software development principles, it is considered a mindset and company culture. Shorter development times and closer collaboration are the goals of the agile software development paradigm. DevOps builds upon the guidelines and goals of Agile and offers workflows and tools for the software development process striving for an increased user experience value [BvdG20], [DD16].

2.1.3 Principles of DevOps

In addition to Agile principles, DevOps also extends the *Extreme Programming* approach by applying its principles to the operations and infrastructure aspects of the application [DD16]. The goal is to provide a structured and comprehensive process from coding through testing, packaging and deployment, to operation and monitoring. This process is referred to as a workflow which is made up of several steps. Table 1 shows such an exemplary workflow with a minimum number of steps [BWZ15].
The tasks and activities resulting from such a workflow represent the core processes of the DevOps operations. These tasks include configuration management, release management, continuous integration (CI), continuous deployment (CD), infrastructure provisioning, test

automation and application performance monitoring [BvdG20]. Generally, it is the overall goal to conduct structuring and optimizing along the entire path of the SDLC, while also providing the team with appropriate tooling. Even the infrastructure provisioning is structured and automated as its specifications are stored as code, just like application configurations. This approach is referred to as *Infrastructure-as-Code (IaC)* [BWZ15]. This work will focus on applying this structuring of workflows to the development environments in order to enable homogeneous environments and an efficient development process.
A DevOps-principled architecture has multiple deployment stages. A staging and production environment is considered the minimal foundation of any such project [BvdG20]. In order to test and operate these different deployment stages software teams need to have reliable and efficient ways to manage their infrastructure states and configuration. This is closely related to the IaC principle, which describes the provisioning and maintenance process of computing resources. Tools like *Ansible*, *Puppet* and *Terraform* can automatically maintain, create and set up new (cloud) virtual machines (VM)s based on rules specified in a playbook. Playbooks are structured policies written in a custom language which are also stored in a Version Control System (VCS) [Mie20], [BvdG20]. These files describe how the host operating system is configured. Changes on these files can be handled the same way as regular code, starting with a pull request, reviews, test and approval of the changes. The IaC principle will later be used to properly handle the setup of development containers. It is also used to configure CI tools as described in section 2.3.

2.2 Microservice and Virtualization Concepts

The following section provides a basic understanding of the concepts of microservices as they are used at Symbic. Their implementation, the possibilities they bring with them as well as their limitations are presented below. In this context, different virtualization strategies are explained, as they are often used with a microservice architecture. These concepts are revisited in the proposed solutions in section 4.

2.2.1 Fundamental Idea of Microservices

The fundamental idea of a microservice architecture (MSA) is to have small self-contained, independently deployable software applications. Connections between these applications enable greater services with an extensive range of functions [IS18]. Figure 1 visualizes an exemplary structure of a microservice cluster. It shows multiple microservices that each expose one functionality to the outside world. Application end-points, such as the user interface (UI), provide a broad range of functionality by internally delegating the request to different applications. In the event of an application failure only that specific functionality becomes unavailable while the remaining system maintains its operational status. Due to this architectural design feature, it is advised that while working with data, each microservice has its own database (DB). This prevents a central database from becoming a single point of failure that can bring down the entire service. Since the service

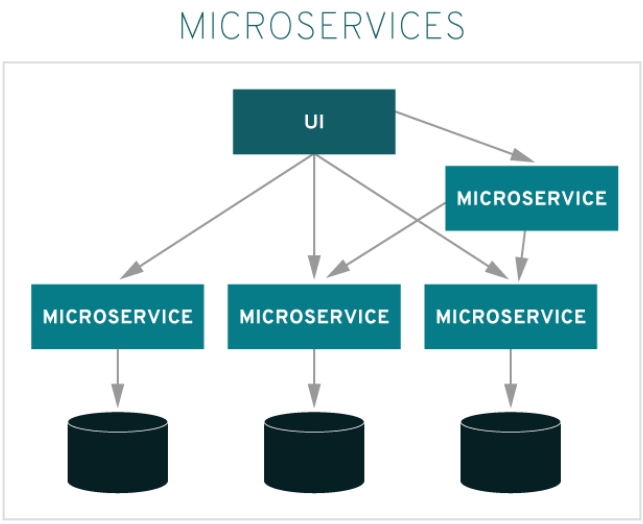

Figure 1: Structure of Microservices — [Altered],
Source: [Red21b]

is built upon multiple small applications, it is possible to use different technologies for every application. The initial cost of new technologies is thus lower and individual service parts can be replaced at a lower cost compared to a monolith. Another core feature of microservices is its scalability. If the load on one part of the service increases, new instances of that application can be deployed to balance the load across multiple instances [IS18]. This concept requires a fast and reliable process of creating new application instances which may build upon different technology stacks.
The installation and setup process of new hardware can be a time-consuming task. In order to reduce the amount of time it takes to create new application instances, the software industry uses the concept of virtualization.

2.2.2 Virtualization and Containerization

Virtualization is an abstraction layer. The physical hardware (host) runs a hypervisor that allows the execution of (multiple) virtual machines (guests), which act like a regular computer [Por12]. This approach allows the usage of heterogeneous hardware without an impact on the guest operating systems due to the abstraction provided by the hypervisor. Without the need for specialized hardware and the dynamic allocation of resources, efficiency is increased [Red21c]. Additionally, virtual systems can be managed more easily because they are fundamentally just one big file on the host's storage device. They can be created on command, cloned and deleted without the configuration steps of a physical system. In the enterprise industry it is common to use this flexibility to start additional VMs on high resource load. According to a study by the International Data Corporation (IDC) more than 80% of data-center workloads are virtualized [Che17]. Virtualization comes

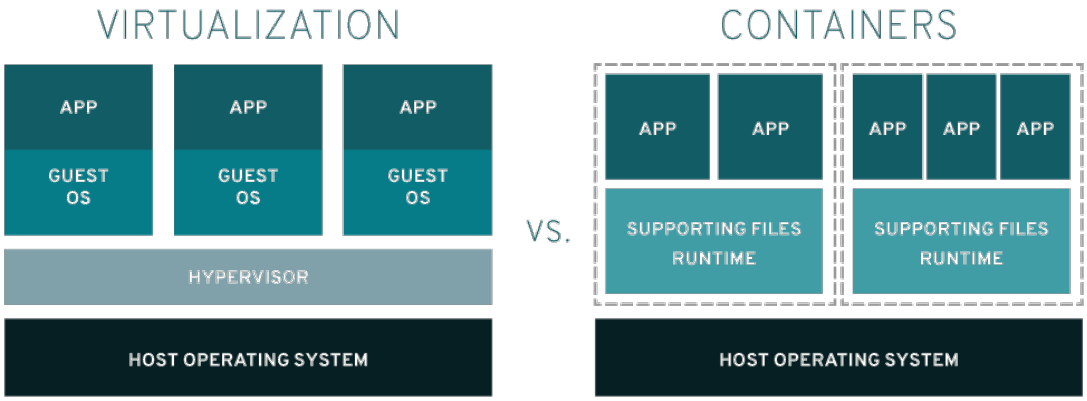

Figure 2: Comparison of VMs to Containers,
Source: [Red21a]

with the benefit of security. The majority of hypervisors strictly separate the host and the guest system in that the guest system is not allowed to use the hosts resources and access its files unless it is explicitly configured to do so. Compromising a VM does not affect the host or any other VM [Por12], [Red21c].
Full guest virtualization emulates a complete operating system (OS), including the kernel, system libraries and even the majority of hardware devices. This abstraction comes with a performance penalty called overhead [Por12]. A supposedly more lightweight approach of virtualization is called containerization. Studies by Ericsson Research, Nomadic Lab [MKK15] and the Zhengzhou University [WSCC17] conclude that, in fact, container-based virtualization solutions provide better performance, especially in disk I/O and network I/O bound scenarios. Containerization focuses on the isolation of a single application process in a virtual runtime using control groups and namespace technology [Kou18]. Unlike VMs, system and kernel functions are not virtualized and are passed through to the host machine. Consequently, the overhead is reduced and allows for the ability to run additional application instances compared to a VM-based approach with the same amount of compute resources. Figure 2 visualizes the differences between these approaches. The left side shows a traditional VM-based approach. On top of the host OS runs a hypervisor which provides three full guest operating systems with one application each. Each guest is fully isolated and features its own kernel, OS and runtime libraries. The container-based solution on the right side only needs one host OS and provides multiple application instances with shared libraries and runtimes in separated, isolated namespaces.
Apart from the performance benefits, the presumably main advantage of containers is their scalability, due to faster creation and startup times, which is what makes them an adequate fit for microservices [Joy15]. Docker, Podman and LXC are, amongst others, the most popular container-based virtualization solutions. A more detailed explanation of Docker can be found in section 2.3.5.

2.2.3 Usage of Containerization in Microservices

As described above, microservices are small, bounded applications that communicate with each other. In order to follow the principles of loose coupling between applications, the communication should be performed via a protocol which is independent of the programming language used. Typical IP based protocols used are *Representational State Transfer (REST)*, *WebSockets* or *GraphQL* [IS18]. These loose-coupled applications have the advantage to be developed simultaneously from different teams as well as upgraded and replaced independently. As a result, applications can be developed much faster and more flexible, following the principles of agile development [IS18], [Red21b].
The usage of containers drives this speed and flexibility even further. Containers provide a consistent and isolated, yet flexible runtime for applications [KLT16]. Applications are packaged within known good runtimes. This reduces the setup time of the deployment and eliminates host-specific errors. New application instances can be started without additional configuration. As a result of these successful concepts, tools like *Docker Swarm* and *Kubernetes* have been developed which can scale distributed applications in a managed dynamic, even automatic, way across multiple computers.
Packaging and eventually deploying the application introduce additional work for developers, which was previously a task of the operations team. As already stated above, the developer and the operations team are not separated in a DevOps culture. This approach values team communication, flexibility and autonomy, enabled by the structured, automated workflows between the development and operation tasks [DD16]. Eliminating manual tasks allows developers to focus on the actual application development. One of the main concepts in this process is the usage of CI and CD workflows.

2.3 Tools for Achieving a DevOps Environment

Continuous integration and continuous deployment are two working concepts particularly well known in an MSA. Since every application only provides one part of the overall service functionality, it is uncertain what effects a change in one application has on other applications and on the whole service. Accordingly, testing must take place between applications, especially if they are developed by different teams. The following section provides concepts and workflows in order to enable these complex processes in efficient ways.

2.3.1 Version Control

A Version Control System (VCS) is an essential tool in the software development industry to keep track of which change has been made when and by whom. Contact persons for specific code sections are thus directly known and the responsibilities are clearly defined. *Git* has become the de facto standard over the last 10 years and is also used at Symbic, just like by over 93% of all developers according to the StackOverflow Developer Survey 2021 [Sta21a]. Git allows for collaborative, distributed work in own branches without affecting others. Changes can easily be merged back into the main branch, after an optional review has

approved any changes. The new code is then made available for all developers. At Symbic, these functionalities are provided by a self-hosted instance of *GitLab*. This service is used to coordinate teamwork via issues and to document projects. Additionally, it serves as a central code repository. Its adaptability, extensive integration options and built-in CI tools provide all the necessary software lifecycle programs by a single service bundle [Git21c]. *GitHub* and *Bitbucket* are alternatives, both of which offer code hosting, issues and CI/CD.

2.3.2 Continuous Integration

Continuous Integration (CI) is a practice in which new code is regularly integrated into the main code branch and into the overall service. Instead of having isolated functional branches that are worked on independently for months, in a DevOps environment changes flow back regularly into the main code branch to ensure it is free from conflicts and errors. Especially in interconnected services, it is indispensable to ensure that a change in one application do not have an unintended effect on other applications. continuous integration systems can perform automatic tasks, called jobs, on the code when it is checked into the VCS. These tasks can ensure a specific code-style, run unit or API-tests and can also run integration tests against other services. If a task fails, the corresponding developer is notified and the changes are not incorporated into the main development branch. Accordingly, errors causes can be identified quicker and be resolved without delay [BvdG20], [BWZ15]. Popular services providing continuous integration functionality are *Travis*, *CircleCI*, *GitHub Actions* and the previously mentioned GitLab CI. These services differ in their function range, type of job configuration, the level of control, in their pricing models, or the number of free jobs, respectively, and in whether they can be self-hosted.

2.3.3 Continuous Deployment

Continuous deployment, on the other hand, is a practice bringing these regular updates into production quickly and automatically, making them available for customers, respectively. It typically involves two steps, continuous delivery and continuous deployment. The continuous creation of software bundles, which are called artifacts, is called continuous delivery. The transfer of these artifacts and the process of making them usable is described as continuous deployment. To ensure product quality, it is best to have multiple deployment environments, as described in section 2.1.3. Each new commit is automatically deployed to a test or development environment, in which it undergoes automatic or manual testing. Common test types are security scans, load-, usability-, and acceptance tests. If all tests are successful, the version is promoted to the next deployment environment. In case of an error, the version is discontinued, and the developers get the corresponding notification. Only if none of the testing environments reveal any errors, the version will be transferred to production as a new release. CD is an extension of the CI principle and represents the next logical step in the software development workflow, which is why CI/CD are often used together [BvdG20]. Well known CD solutions are *Jenkins*, *Azure Pipelines*, *AWS CodeDeploy* and *Argo CD*.

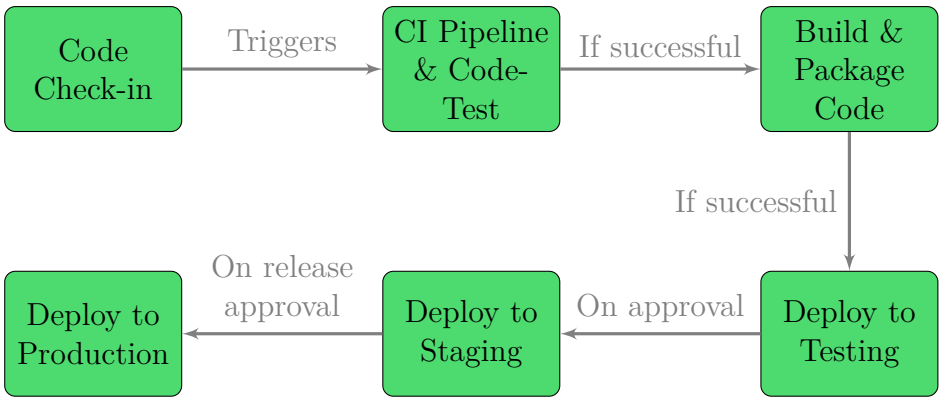

Figure 3: Continuous Deployment Workflow,
Source: Modeled after [BvdG20]

2.3.4 Building a Streamlined Workflow

By combining these principles powerful workflows can be created that accelerate software development through automation and ensure consistent quality [BWZ15]. Figure 3 visualizes a complete exemplary CI/CD pipeline. It is typically built from multiple stages, each stage bundles one or more related tasks. A stage gets triggered by an event, such as code check-in or a previously successful stage. Common tasks are code style tests just as unit tests, automatic compilation or packaging and the deployment to a test system [BvdG20]. Both CI and CD are practices to automate tasks that have previously been performed manually. Integration testing and packaging both happen within the scope of a dedicated, autonomous system, which gives developers and operators more time for other activities. Deploying and using such a system are typical tasks in a DevOps practicing team. Here, the focus is on collaboration between the development and operations teams. These workflows allow a team to react quickly and flexibly to changes and thus to comply with the agile principles. By collaborating closely and developing these well-functioning systems, the main goal is to add value to the product and to customer experience [BvdG20].
At Symbic the GitLab instance is used to automatically test, build and deploy the code to a testing environment. In order to quickly deploy applications, Symbic uses the container software Docker. The artifacts needed for this are automatically created by the GitLab CI and made available in built-in private container registries accessible to all authorized entities [Git21c].

2.3.5 Containerization with Docker

In order to use the containerization concepts described in section 2.2.2, they have to be implemented in software. One solution for the implementation of container-based virtualization is Docker. Currently, Docker is the most popular containerization platform supported on Linux, Windows and macOS [Sli21]. It utilizes the hosts Linux kernel and its cgroups functionality for resource isolation. On none Linux host systems, Docker

virtualizes the kernel via a built-in hypervisor. Docker was chosen for its wide usage, extensive documentation and its platform independency, which enables the usage in a local and productive operation.
Docker allows the isolated execution of applications within a container in a well-defined state without affecting the host. The initial state of a container is defined in a disk-image that bundles all the software, libraries and configuration files needed. Images are built based on a *Dockerfile* that contains sequential, imperative instructions on how the container OS should be configured. An exemplary Dockerfile can be found in Listing 1. Common steps are the `COPY` command to add files from the host to the container and the `RUN` command to execute shell commands within the container. Each file-changing instruction adds another layer to the overlay-filesystem that is used by Docker. Effectively making each layer a read only filesystem once it is written. At runtime, a file-lookup is performed in the top filesystem layer; lookups in the layers below follow if the file is not found in the first one. This allows the reuse of layers, because changed versions of files are simply pushed into a new layer and overshadow the original files. Images are usually kept as narrow as possible, so that they can be transferred more quickly and present less a target, when it comes to matters of security. Therefore, only the minimum of necessary programs are provided in an image. Docker images are distributed via a registry to other systems. The company which develops Docker provides an official and public Docker image registry called *DockerHub*. It offers official images of common applications, such as databases and webservers, as well as base images for own projects [ÖK20], [Doc21b].

```
1  # The base image to start from
2  FROM ubuntu:18
3
4  # Install nodeJS
5  RUN apt-get update && apt-get install -y nodejs npm
6
7  # Copy content of the Hosts "backend" folder to "app"
8  COPY ./backend /app
9
10 # Install all node-modules
11 RUN npm install
12
13 # Specifies the command that is executed at container start
14 ENTRYPOINT [ "npm", "start" ]
```

Listing 1: Exemplary NodeJS Dockerfile

When a container is started, the program specified in the image is executed, and the container runs until the started program exits. Each container is treated as an independent full-fledged system with its own OS. For this reason, each container has its own IP address and is by default not accessible from the host. The user has to explicitly expose a container port to the host system in order to access the application inside the container. Each container does not only have its own IP address, but is also on its own virtual, software-defined network, which is managed by Docker and even uses a Domain Name System (DNS) server that is inherent to Docker. Accordingly, containers within a defined network can communicate with each other by their (host-) names and provide their functionality outside the network by exposing their published ports to the host OS [ÖK20].
Unlike VMs, the container's guest OS is not maintained, as containers are considered disposable. Instead of updating the software within the container, it is simply replaced by a newer one from an updated image. All changes and new files within the old container are irreversibly lost. In order to avoid the loss of user data, corresponding files can be stored persistently on the host system and made available within the container via mounts. A mounted directory behaves like a new top layer in the containers overlay-filesystem. Docker distinguishes between volume-mounts and bind-mounts. Volumes are managed by Docker, have a unique name, can easily be shared between containers and are transferable between hosts whereas, in the case of bind-mounts, a specified directory is mounted from the host to the container. All filesystem attributes and file properties are passed on to the container, while their management is up to the user [ÖK20], [Doc21b].
Alternatives to Docker are *Podman*, *Rkt* and *LXC*, which Docker was originally based on. The downside of these tools is their lower degree of integrability and lack of multi-platform support.

2.4 Relevance of Modern Development Techniques

The concepts explained above are becoming more and more relevant with the increasing adoption of DevOps and agile principles, even outside the software industry. The annual agile report shows a significant growth in agile adoption from 37% in 2020 to 86% in 2021. The main reason for adopting agile practices seem to be enhanced ability to manage changing priorities and an increased speed of software delivery. Customers benefit from new features directly and developers receive immediate feedback. The same reasons for adopting agile also apply to the investment in a DevOps culture. The annual survey reveals, that more than 70% of all respondents, are executing or planning a DevOps initiative. One of the biggest challenges in implementing DevOps seems to be inconsistencies in processes and practices, which is also reflected in the fragmented and heterogeneous processes described in the next section [Ver21].

3 Analysis of Current Development Environments

The following section will display the state of typical development environments. The most commonly used operating system is compared to productive environments their differences are pointed out. The problems caused by this are described and further challenges in the development work, especially considering a microservice architecture are revealed.

3.1 Current State of Development Environments

Software development involves a broad variety of tasks. Depending on the project, tasks can range from web development, embedded system development, desktop or mobile application development, data analysis or the pure maintenance of one of these areas. Each of these software development fields has its own workflows and requirements for the actual development setup. Even within these specialized categories, there are different requirements, depending on the scope and size of a project. Accordingly, the setups of development environments can differ greatly from one another. Due to the large scope of different variants, this work will only refer to web services built on a microservice architecture. Nevertheless, this work can also be adapted to other types of software architectures.
According to StackOverflow Survey 2021, traditional local computers are the most widely used solution for developing software among professional Developer. The primary operating system used is Windows, followed by macOS and Linux [Sta21b]. Accordingly, developers need modern hardware, on which all the required applications must be installed and set up individually. Depending on the application area, this may include some applications for which license costs may have to be paid and which must be kept up to date after the initial installation. The setup and maintenance of these tasks are necessary supporting processes in software development, but they cost a lot of time and effort, and therefore money without producing any actual progress and value to the project. This is especially true if there are many developers in a large company with frequently changing developers. Once the environment is running, developers can start to code and contribute to the project. The developers of GitHub.com created a series of scripts just to get developers to a working environment in about one day. Because of frequent error these scripts included a global clean option (called `--nuke-from-orbit`) to reset the environment the initial, known good, state [Git21b].
Such a local working environment leaves developers in a position, in which they have lots control over their development environment. However, local manual setups require much effort and result in an elaborately error-proneness's environment as the next section demonstrates. A possible solution to these problems is presented in section 4 and compared with still fairly recently developed alternatives in section 6.2.

3.2 Common Issues in Modern Development Setups

The current congestion of development environments, as described above, is a collection of different programs and their configurations that each developer has to spend worktime on in order to set up a working state suitable to their personal preferences. Only when this has been accounted for the respective project the developers can begin with their actual work. However, their work may be restricted by a chosen software architecture, and interrupted if changes in the code or its runtime cause problems with the development environment. The following section describes the most common issues with local development environments.

3.2.1 The Initial Setup Process

Newly recruited developers or those changing projects need time and support to settle into the new project. They are not familiar with the code base and instructions on how to set up the environments correctly. Evidently, good code documentation helps as long as it is available, up-to-date and detailed enough. Specific software packages such as DBs, interpreters or compilers need to be installed and configured in the correct versions in order to make the local environment operational. Programming languages such as C/C++ and PHP do not included a debugger in their language framework, and they need to be installed and set up separately. Depending on the project, this can be very extensive and require a lot of time and support from other team members. Setting up a platform independent NodeJS project is considerably simpler as system specific C++ or PHP projects, thanks to the included package manager *npm*. A survey by ActiveState shows that more than 25% of developers need five more hours to set up a development environment. Considering that more than 65% of all developers do this one to four times a year, 30% even five to twelve times a year, this adds up to a significant percentage of work hours [Inc19]. A Web-Search also reveals that broad discussions about automating this process actually do exist. The topic seems to be mostly practice oriented and scientific papers on the topic are unfortunately rare. Even though some of this initial work can be automated through the use of scripts, like the ones used by GitHub, these scripts must be created and maintained for each operating system used. They may be a time saver as long as they don't break due to moved download links or unavailable files.
Further problems arising from the configuration of the respective programming language are discussed in more detail in the next point.

3.2.2 Lack of Testing Options

The goal of tests is to verify the behavior of the system under the given conditions. It is a crucial practice to ensure product quality and high custom value [BvdG20]. Functional tests can be categorized in the four stages shown in Table 2. According to the test pyramid, unit tests are the tests with the smallest scope which, however, should be implemented most heavily. They ensure the correct behavior of a function or a class and only depend on the code to be tested itself. Only preceded by a compiler or a linter,

they represent the earliest stage to detect errors. In a Test-Driven Development (TDD) environment, the tests are even written before the application code itself to ensure that the application meets the project requirements. The use of unit tests remains unchanged in a microservice architecture, as each service can still have its own unit test suite. Due to the number of applications, though, interprocess communication (IPC) in a microservice architecture increases significantly, making integration and higher testing scenarios more complex [Ric18].
Integration tests should verify that multiple applications can successfully communicate with each other. In order to perform these kinds of tests, it is necessary to run both applications simultaneously [BvdG20]. While developers can use tools such as *Postman* to check the results of a application programming interface (API) request, this does not guarantee that two applications can actually communicate successfully. Integration tests can either be performed in a CI system, where an error can only be detected after the code has already been pushed the central repository and all CI tasks are completed, or locally with immense configuration effort. Due to the high rate of interprocess communication, the use of tracing tools may be necessary to isolate any inter-application errors. Integrating them individually into local developer setups is another tedious task that does not add any direct value to the project. Besides software design and meetings, explicit testing of applications consumes most of a programmer's time when he is not coding [Inc19].
The result of both integration testing variants is either in a very late error detection, which slows down development and makes it more expensive, or in an individual, local environment that is difficult to maintain and prone to errors, as the next section will show. Both scenarios leave developers with insufficient testing capacities for efficient software development. Although contract testing for compliance of API specifications can be used for integration testing, this merely postpones the problem to a later test stage [Ric18].
Entire application tests, called end-to-end tests, are extensive and time-consuming. They require all applications to be deployed in order to cover entire business logic operations [Ric18]. They should be used thoughtful and performed on a testing or staging environment. However, these only form a small part of the test scope and are typically performed by a separate quality assurance (QA) team. When an error is detected, its cause must be found, for which tracing and debugging programs are used. The absence or tedious configuration of these tools complicates identifying the bug and is one of the problems already described above. The lack of tools and too complex, divagating development environments are another main challenges for implementing DevOps practices [TSM19].

3.2.3 Configuration and Dependency Management

Once the development environment is set up, it is a crucial to maintain it in such a way that developers can perform their actual tasks and contribute value to the product. This includes keeping both configuration and dependencies in a good, consistent state with colleagues and the productive environment. Some programming language frameworks are able to keep dependencies consistent across multiple systems without much effort thanks to their integrated tools. The current NodeJS 14 long-term support (LTS)-Version, for

Test-Type	Description
Unit test	Test a small part of a service, such as a class.
Integration tests	Verify that a service can interact with infrastructure services
Component tests	Acceptance tests for an individual service.
End-to-end tests	Acceptance tests for the entire application.

Table 2: Stages of Software Tests,
Source: [Ric18]

example, installs all required dependencies into the local project folder and keeps a precise record of their versions by means of a `package-lock.json` file. Other languages either come without any package manager at all (like C/C++, PHP and Java), or their package manager installs dependencies globally (Go, Python). The existence of tools like Pythons *venv* package, which creates virtual isolated dependency environments for each project, proves that dependency management is, in fact, a problem in local development environments [Pyt21]. Apart from architectural designing, collaborative meetings and testing time, the investigation of bugs and the maintenance of the application setups are the most time-consuming tasks for developers are not coding. Among the bugs that can occur in a project, dependency issues are ranked as the third-largest group, just after of packaging issues [Inc19].
Even the local package installation approach of NodeJS does not solve the problem of developing against multiple versions of the NodeJS framework. Testing a new major version of NodeJS can break existing project configurations, undoing these changes can be tedious and time-consuming. Additional tools like the *NodeJS Version Manager (NVM)* have to be used in order to run multiple NodeJS versions on the same system. It becomes particularly problematic when developers are working on several projects with different dependency requirements at the same time. This also includes the maintenance of legacy applications. Legacy applications may require runtimes and libraries that are no longer supported on modern systems. Versions prior to PHP 7, for example, can no longer be installed (without additional effort) on current Windows or Linux distributions. These issues are assigned to the category of runtime dependency management. In a microservice architecture, the service dependencies between different applications further extent this problem.
In a microservice architecture, there are several granular applications, each providing functionally related logic and having bounded endpoints. Each endpoint can be considered a public API, even if the service is only accessible within the overarching service. Communication between these endpoints is called inter-service communication. Maintaining compatibility between applications becomes a challenge, especially with many microservices or when using a service mesh [IS18]. If a team member changes the interface of one microservice, this affects other services and the developers responsible for these applications need to be notified and respond in order to avoid unexpected errors. This problem is particularly evident in the microservice Death-Star, which shows all of Netflix's microservices as part of a confusing point cloud with numerous connections to each other [DZo19].

Every single compatible API endpoint must be defined and configured in every other application that uses it. The management of these inter-service communication settings has a direct impact on the testing possibilities, which are discussed in the previous section. This illustrates once again how much effort local testing of applications can have.
In ideal environments, the local setup would always work as long as no changes are made. However, there are always changes, OS and security updates, changes to the dependencies or the temporary change of the network port for a side-by-side comparison. The modification of the database schema by one developer can cause a broken environment for other developers. These slight changes over time are called configuration shift. No system is identical to another, which can lead to unreproducible builds and fluky errors. To avoid such errors in production, tools like *Terraform*, and *Ansible* are used. These tools implement the IaC priceable and can create multiple server configurations that are exactly identical to the state defined in a playbook or config file. Each instance of a machine is configured in such an exact and consistent way a human being would be incapable of. If a VM behaves abnormally, it is torn down and recreated. However, local environments are not considered disposable, they are personalized and therefore hard to create and maintain automatically. Some projects may even require perfect reproducibility, especially in a scientific context. Yet, this is hard to achieve in indiscriminately configured and personalized environments. Managing different project configurations that conform to team members and the production environment thus takes considerable time and, according to ActiveState's study and GitHub's experience, is also responsible for a considerable number of errors that occur during development. Examples of the errors are further elaborated in the next section [Inc19], [Git21b].

3.2.4 Issues Caused by Heterogeneous Environments

According to the StackOverflow Survey 2021, Windows is the most widely used operating system for software development, but in the server area, Unix-based systems clearly dominate with a market share with a percentage 75.3% among webservers [Sta21b], [W3T21]. In data-centers 80% of workload is virtualized, the technology primarily used for this is based on Linux. The current state of affairs is thus obvious: The development takes place on Windows, whereas the actual operation occurs on Linux. This fundamental difference in the core runtime environment of applications can be the cause of a variety of operating system specific errors, even if the used programming language supports cross-platform compatibility. One major difference is the structure and functioning of the file system. Windows uses alphabetical identifiers like `c` and `d` for drive partitions. Linux, on the other hand, has a root directory (`/`), in which all underlying folders and partitions are located. External partitions can be mounted at any place and with any name. A frequently used directory for external partitions is to be found in the `/mnt` directory. Accordingly, the representation of file paths also differ. While user-data on Windows is located in `C:\Users\USERNAME` using backslashes, On Linux it is usually accessible via `/home/USERNAME`, with a normal forward-slash character. The inclusion of external libraries, assets and other files via paths is a common task in programming. Hardcoding these paths can lead to unexpected be-

havior on a different host. Although some programming languages offer constructs such as `File.Separator` (Java), or attempt to perform the path-separator conversion automatically (NodeJS and Python), errors still occur on heterogeneous systems. Prominent problematic examples of this are the scripting languages Bash and PowerShell, which cannot handle alternative path separators. One of the reasons is the special meaning of the backslash, which is often used to escape reserved special characters.
In a way similar to the file paths, the encoding of the newline character differs depending on the OS. While Windows uses, by default, the `CR` line ending format, Linux uses the `LF` format. There are applications, which can read both encodings, but nevertheless, there are also applications that either only support `CR` or that can only read newline characters encoded with the `LF` format. Bash scripts created on Windows cannot be executed under Linux without a conversion from `CR` to `LF`. The permission system of Windows and Linux also differs. Windows uses the central user account control in order to manage the processes of reading, modifying and changing the owner of a file. Linux, on the other hand, has a read, write and execute flags for every file, determining owner, group and other permissions. Windows does not support the executable flag at all. Accordingly, all files created on Windows, which are then copied to a Linux environment, cannot be executed without further steps. Another example for these problems is the handling of keys and certificates. The widely used key-based cryptosystem Rivest-Shamir-Adleman (RSA) can be used under both Windows and Linux, yet differently. RSA-Keys created under Windows are not accepted by many Linux applications because they are not readable due to `CR` encoded line endings or are rejected because *nginx*, *apache webserver* and *sshd* require that private keys are only readable by the user and the permissions are too open by default.
These heterogeneous based problems occur in addition to operating system specific programming. Low level operations, such as forking a process, spawning a new one and sending (exit) signals, are fundamentally system-specific. Higher level programming languages abstract some of these operations, yet, some functions are only available on one specific system such as Unix sockets. Dependencies with native libraries are also operating system-specific. Either there has to be different variant for each OS and each system architecture, or the library has be compiled into native library on the target system at install time. For example, NodeJS uses the *node-gyp* module and Python uses *wheels* in order to compile native liabilities on the host when it is needed. Native libraries are often included for cryptographic tasks in particular, since these are especially efficient. The uncertainty whether all the necessary tools are available on each system and have the correct version where these libraries are required adds extra complexity and creates new error sources. These potential differences are valid for the development environments between developers within a team as well as in comparison to the productive environment.

3.2.5 Additional Effort for Developers

The preceding sections already showed how heterogeneous environments, many small services and an extended testing effort are causing additional work for developers. In addition to that, there is also the management of secrets such as, API-tokens or keys, and, for new

developers, the setup of virtual private networks (VPN)s and communication tools. Even when the local environment works without errors, there are ongoing obstacles. As mentioned before, the value of microservice is the multiplicity of small applications, which together provide a greater functionality. In order to start multiple microservice developers most likely need multiple terminals to execute each application individually. A larger amount of terminal sessions can quickly become confusing. Eventually, services even have a specific order for startup. This phenomenon is also referred to as terminal hell. However, there are still obstacles to overcome after the start of the individual applications; tracking their output is also one of the tasks. Log outputs provide insights into what the application is currently doing. This makes it possible to quickly check the expected behavior of an application. The output of many logs on different terminals, however, only contributes to a rapidity increasing confusion. Analogous to the terminals, this is called log hell [IS18].
These circumstances lead to developers spending more time managing, configuring and analyzing applications rather than actively developing them, which leads to a slower pace of development. Slower development increases cost and leads to customers having to wait longer for new features and bug fixes, which diminishes the customer experience. If a company's business model is to offer software development as a service, slower and higher development costs mean that the company cannot compete with rival companies and loses contracts.

3.3 Proposed Solution for more Efficient Workflows

The root cause of these problems is that local development environments fundamentally differ from production environments. Local environments are heavily individualized and personalized, while production environments are standardized but very complex. Local environments cannot be set up automatically and thus cannot simply be torn down and replaced in the event of errors. Long troubleshooting sessions are the consequence resulting in a slow-down of the development progress.
In the server world, virtualization technology is used in order to archive uniform and well-defined environments. A consistent and isolated application runtime environment is provided by a container based virtualization solution such as Docker. In order to allow interaction between applications, they can be orchestrated and scaled with via a Docker Swarm or a Kubernetes cluster.
This thesis proposes to use exactly this containerization approach for local development environments. Thus, the configuration effort, the lack of testing possibilities and the occurrence of local and operating system specific errors should be reduced.

4 The Concept of DevContainers

This section proposes the usage of Development Containers (DevContainers) as a solution to the problems of local development environments as described in section 3. The concept was developed oriented towards the Platform as a Service (PaaS) principles predominant in the cloud. Developers should only have to deal with the application to be developed. The runtime environment and all configuration related tasks are managed by the DevContainer. They combine the application runtime and its configuration into an isolated environment by using the lightweight virtualization approach of containers. One difference between DevContainers and the PaaS principle is that DevContainers provide neither own hardware, nor is it management by and third party entity.
The details of such a DevContainer concept are described below. Followed by a delimitation of the cases, in which its application scenarios are sense and when they are not. Finally, the advantages and limitations of this solution are outlined.

4.1 Description of a Conceptual Environment

The idea of DevContainers is to bundle the application code, its runtime and configuration into an isolated system. Only the minimum necessary scope for accessing the applications functionality is made available on the host system, all other resources remain isolated. According to the design of VMs and containers, they can be started and stopped at will without encountering the risk of permanent changes on the host system by the containerized applications. When the container is started, all port, path and secret configurations are already set. Developers are not required to do any further manual configuration. Different versions and branches of an application have their own container, which are completely independent of each other and the host. Auxiliary applications and interdependent services can all be started simultaneously and in the correct order. The DevContainer environment builds upon the container implementation Docker and thus, is completely independent of the host OS. Since Docker is Linux based, the development environment is thereby much closer to the production environment, potentially decreasing system-specific errors. Even in case of an error and the environment ends up in an undefined state, the principles of containers can be applied. The DevContainer can simply be discarded and recreated from a well-defined template within seconds. This design enables environments that are uniform and 100% reproducible.
Such a setup allows developers to choose any host operating system because the entire application code is executed in a virtualized environment. It increases the initial setup time and prevents configuration drift since all configuration settings follow the IaC principle and are stored as code. This way, new runtimes or dependencies can be tested without the risk to corrupt the local development environment. Through automatic orchestration, integration tests can be performed easier and earlier, decreasing the time until an error is detected. Dependencies and common software like debugger are already present in the container, so they do not need to be installed separately. This newly designed concept of

DevContainers for local development environments promise to solve the problems described in section 3 and allow developers to focus on programming rather than configuring and maintaining their working environment.

4.2 Pre-requirements for DevContainers

Before using DevContainers, one has to verify that this approach is the appropriate strategy for solving the problems encountered. DevContainers are based on virtualization technology, and one of their goals is to achieve the greatest possible similarity between the development and the production environment. In order to take full advantage of this possibility, the production environment must already be designed for the use of virtualization with containers. As can be seen in section 4.3, the majority of virtualization solutions are based on Linux, which is also the most widely used system in the server domain. Applications that require Windows can also use Windows-based containers, but these may require additional licenses and configuration, accordingly they will not be discussed any further in this paper. The application to be developed must be suitable for the use in a container. Containers do not offer direct support of graphical output by default. Functions are exposed to the outside world via TCP/UDP sockets or mounted devices. Building the base images for DevContainer takes time, so this should not be performed by every developer, a functioning CI/CD pipeline for the rapid delivery of new images is therefore also recommended.
Although, DevContainers reduce the configuration effort for each developer, the architecture and settings files for the use of DevContainers must be created and then be maintained. As section 2.2.2 has already shown, any kind of virtualization, regardless of whether it is VM-based or container-based, there is a performance overhead. If, as in a typical microservice architecture, several applications are run at the same time, this overhead adds up and places an additional load on the developer's system. Modern hardware with sufficient memory and computing capacity is therefore necessary for the use of DevContainers. The amount of additional load depends on the technology stack used. The exact effects of this overhead are described in more detail in section 6.2.1.

4.3 Creating a DevContainer setup

Section 2.3 has already presented concrete programs for providing the virtualization concept and automating certain processes with a CI system. The usage of these existing tools built the foundation for provisioning DevContainers. Developers need to be able to initialize a DevContainer based environment quickly and must be offered a convenient way to interact with the DevContainer, and especially the application within, without an impacting on their workflow. How the tools described above are used beyond their intended purpose and what tactics are employed to solve the problems described in section 3 is described in the following section.

4.3.1 Defining the Application Runtime

The entire application runtime environment is virtualized by using Docker. However, the basics of this runtime environment must be defined beforehand. In the case of Docker, this is done with Dockerfiles. These provide the instructions for building Docker-images, which are needed for a container in order to start. CI platforms like GitLab typically execute the building process for images automatically on every new application version [ÖK20].

```
1 FROM debian:buster
2 RUN apt-get update && apt-get install -y \
3     git python3 python3-pip make vim emacs && \
4     pip3 install pandas numpy matpltlib pep8
5
6 Copy . /app #Optional
```

Listing 2: Python DevContainer Dockerfile

Listing 2 shows such a Dockerfile for a Python application. The Linux distribution Debian Buster is chosen as the starting point for the image, followed by instructions for installing all required dependencies and programs, including the linter `pep8`. When it comes to the source code, there are two possibilities for implementing the DevContainer concept. Optionally, the program code can be copied into the image so that the image already contains all the program components. The resulting fully self-contained image is the typical method in production operation. The disadvantage of this approach is that the program code can only be accessed within the container and is not directly present on the host system. While this is desirable in production, due to security and strict isolation, it may not be ideal for development purposes. Developers are limited by only using applications installed within the container to edit the code, and the CI system must create new images for every commit. Prematurely discarded containers can lead to the loss of changes that have not yet been published to the remote source code repository [Doc21b]. This would definitely impose restrictions on the developers' workflow.
Alternatively, copying the source code can be omitted and only the well-defined runtime environment is provided by the image. At runtime, the source code stored on the host is then mapped into the container via a bind-mount when the container is started. This way, the source code can be edited and used by any editor or program on the host. In the case of a discarded container, the changes made are still available on the host. Furthermore, a new image only needs to be created when a change in the Dockerfile or dependencies occurs, which saves significant computational effort in the CI system. This developed approach allows simultaneous use of local tools and the programs inside the container. The workflows of the developers are not restricted nor even significantly changed.
After the build process of an image is completed it is made available on a private or public image registry in order to be accessible. In the working directory of a project, the following commands are executed by the CI system in order to create, name and upload the image to a registry:

```
docker build -t my-python-app .
docker push my-python-app
```

4.3.2 Orchestrating the Application Containers

In order for the Python application above to work, it needs a database. Instead of each developer having to install and set up a database on their own, which may then shared between different projects, this can be done in isolation and automatically for each project by using virtualization and composition. Applications in a MSA have multiple services they depend on. Arranging all these programs in order to create one greater service is called orchestration. Well-known applications for container-based orchestration are *Docker-Compose* and *Kubernetes*. Since Kubernetes does not ship with Docker by default, is quite complex and production oriented, a performed comparison showed that Docker-Compose is the recommended choice for local orchestration.
Docker-Compose requires a configuration file, declared in the YAML format, which contains all the necessary information for orchestrating multiple applications. Each application is defined as a service with a unique name when using Docker-Compose. The `docker-compose.yml` file in Listing 3 defines the services `app` and `db`. Each service has an image, providing the initial state for each container. With Docker containers, it is common to use environment variables to influence and configure the application within the container. The credentials for the database and the Python app are set via environment variables. It should be noted that the BD host can just be the service name of the database server instead of an IP address. Within the Docker managed network, the integrated DNS server automatically resolves all service names to the appropriate IP addresses of each container. This virtual network allows containers to communicate with each other. In order access the containers functionality on the host, the network ports used must be explicitly exposed to the host. The Python application uses the alternative HTTP port `8080` and the database server uses the standard MySQL port. When starting the containers, the corresponding ports are allocated on the host system and local programs can access the defined services at `localhost` and the corresponding network port [Doc21b].
If the source code is not copied into the image, it must be mapped into the container using bind-mounts. Line 12 in Listing 3 binds the local `app-src` directory of the host to the container at the location `/workspace`. The database service, on the other hand, uses a volume mount, in which docker manages the allocated storage persistently itself. The configuration is completed with an entrypoint for the Python app that is executed when the container starts.
Docker-Compose ensures that all interdependent services are started in the correct order, that all persistent volumes are created, and that all containers can communicate within their network [ÖK20], [Doc21b]. Since all the necessary configurations are stored in an ordinary text file, the IaC principle applies here. The state of the `docker-compose.yml` file and all changes are tracked in VCS.

```yaml
services:
  app:
    image: my-python-app          #used base image
    entrypoint: python3 app.py    #the command to start
    environment                   #environment variables
      - DB_HOST=db                #for db connection
      - DB_PW=yes
      - DB_USER=root
    ports:                        #exposed port to host
      - 8080:8080
    volumes:
      ./app-src:/workspace        #bind-mounted directory

  db:
    image: mysql:8.0              #used DB image
    environment:
      - MYSQL_ROOT_PASSWORD=yes
    ports:
      - 3306:3306
    volumes:
      - sql_data:/var/lib/mysql #volume mount
volumes:
  sql_data:
```

Listing 3: Exemplary Python Project `docker-compose.yml`

The command `docker-compose up` is used in order to start all services specified in the `docker-compose.yml` file. In case the docker images are yet available on the host, they are automatically downloaded from image-registries. When all services are started, Docker-Compose attaches itself to all running containers and writes the color-coded output of the every program to the console.
Through this concept, developers have isolated program environments that are created quickly, easily and are producible. Multiple separated projects, no longer have to share a database and different language frameworks, interpreters and compilers can be tested independently. In case an additional service, like a fast cache-server is needed, another service can be created quickly in order to provide a *Redis* or *Memcached* server without the need of manually installing it to the host. Individual, manual and diverging configurations for each developer are eliminated. New environments that are similar to the production environment can be created quickly, independent of the hosts operating system. In a microservice architecture, developers have the opportunity to test the interaction of multiple applications before committing changes to the VCS. This way, potential errors are already detected before CI integration tests, which means that they can be corrected more quickly. These characteristics are promising in terms of solving the problems described in section

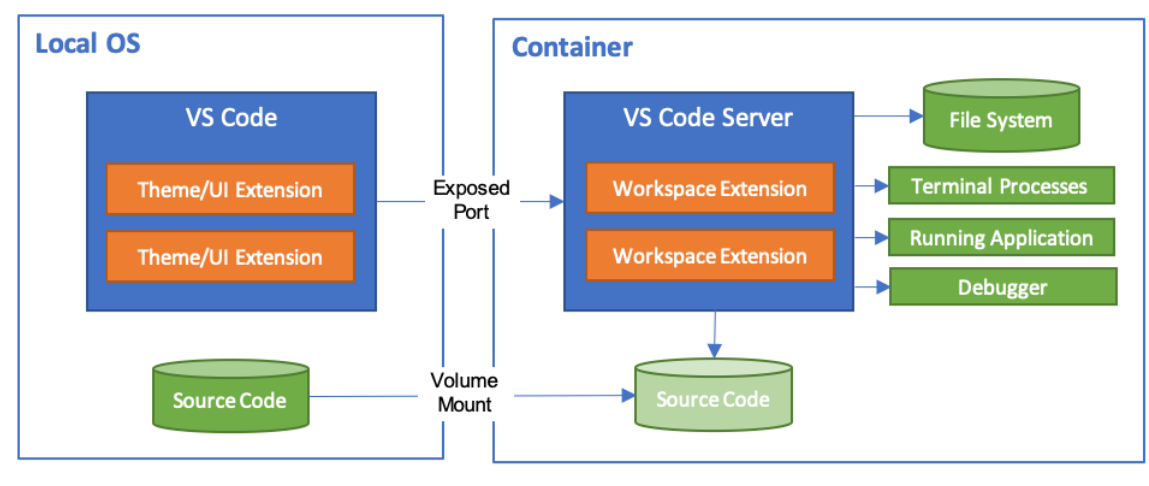

Figure 4: Architecture of Visual Studio Code (VSCode) Development Container Setup, *Source: [Mic21]*

3 regarding heterogeneity, lack of testing facilities, and the tedious configuration of the setup. In order for this concept to be adopted by developers, there must be an equally effective way for developers to interact with their applications within the DevContainer.

4.3.3 Interacting with DevContainers

The primary way developers interact with their code and the application is through their editor. Their variability and the number of different editors is great. *Visual Studio, XCode, Atom, Sublime, Eclipse, Emacs* and *VIM* are well-known general purpose editors. There is usually a distinction between simple text editors and integrated development environments. While text editors are quite simple and only provide basic functionalities like syntax highlighting, integrated development environments (IDE)s are much more comprehensive with powerful IntelliSense suggestions, built-in project management, a VCS, debugger, graphical visualization and build tools. The choice of the editor is a personal decision for most developers, and they are customized according to their preferences. For this reason, the proposed solution developed in this thesis does not require a specific editor for DevContainers, but gives a recommendation which will be used as a reference for the rest of the work. VSCode is a free and platform independent editor with extensive extendibility, which is used by over 70% of all developers accordingly to StackOverflow [Sta21b]. One of its beneficial features for virtualized workloads is its remote development functionally. This officially provided, extension allows developers to use the intuitive comfort of a graphical UI while running the code, the application and auxiliary processes like a debugger on another, remote machine. Figure 4 shows how the VSCode frontend connects to a remote machine or container and installs a server instance of the editor. The server side instance manages the access to the remote file system and the execution of processes while communicating with the local VSCode frontend instance for comfortable access to

these functions. This type of remote development works for Secure Shell Protocol (SSH) connections, the Windows Subsystem for Linux (WSL), and on Docker containers. WSL is a built-in Windows feature to provide a Linux environment on Windows hosts without the need of a separated VM. The implementation of Docker for Windows is build upon this feature [Mic21]. In the further course, only the remote functions for containers will be considered here.
In order t access a containerized application, the appropriate ports need to be exposed via Docker-Compose. Even though VSCode provides a similar feature through the remote extension, it was concluded that this function is unstable. It allows making remote processes available locally through port forwarding. VSCodes implementation can not expose ports for multiple containers simultaneously, making the feature only useful for a service that are currently worked on, or for services with dynamically changing network ports. However, this feature can be used to make a short-lived process available on the host quickly without having to modify the configuration in the `docker-compose.yml` file. However, Dockers port mapping functionality is still the primarily used way to interact with the containerized application.
In order for VSCode to connect to or start a DevContainer, a configuration file is required. This is provided by a `devcontainer.json` file. For each service developed, the path to the `docker-compose.yml` file is specified, as well as the service name VSCode should connect to. Listing 4 shows such a configuration file. It also specifies, which directory the VSCode server should open, which extensions will be installed and what happens when the container starts or stops. If this file exists, VSCode automatically offers to reload the local project using the DevContainer.

```
{
  "name": "MY-PYTHON-APP",

  // Path to the docker-compose.yml file.
  "dockerComposeFile": ["../../docker-compose.yml"],

  // The service property for the container that VSCode uses.
  "service": "app",

  // The project folder to be opened by VSCode when connected.
  "workspaceFolder": "/workspace",
  "shutdownAction": "none",

  // Extensions to be installed when the container is created.
  "extensions": [ "ms-python.python", "ms-python.vscode-pylance"],

  // Runs this command after the container is created
  "postCreateCommand": "/workspace/.devcontainer/kill_app.sh"
}
```

Listing 4: VSCodes Container Configuration File `devcontainer.json`

All auxiliary services are started automatically, directories are mounted, and the network ports are allocated accordingly. Thus, the development work is nearly identical to a local setup [Mic21].
All these functions can be archived without VSCode by using, (SSH) port-forwarding or terminal-based editors. Even without VSCode, any file change made by any editor on the host will take effect, since the source code directories are bind-mounted into the container.

4.3.4 Variations and Additional Supporting Tools

In addition to the tools mentioned above and in section 2.3, other auxiliary programs can be used. To simplify certain workflows and automate recurring tasks, scripts will be used. The scripting language Bash can be used natively on Linux and macOS, the installation of Git for Windows also brings Bash support to Windows. Accordingly, one uniform scripting language can be used to perform platform-independent operations and to simplify complex instructions. In order for developers not having to wait for the creation of the DevContainer images, these should be built automatically by a CI-tool and made available in a private container registry. Accordingly, uniform and up-to-date images are quickly available to all developers.
It is also possible to use other management tools via Docker. Database management tools, such as *phpMyAdmin*, can simplify administration by adding further services. Docker's capabilities allow multiple instances of these tools to run in isolation from each other in different versions without creating conflicts. Although Docker-Compose offers a color-coded log output, even this can become overwhelming if there are too many logs from multiple applications. Since the Docker stack is used anyway, enterprise log aggregators and analysis tools like *Grafana* or *Elastic-Search* can be used to get a persistent and searchable log dashboard. No program outputs will ever be lost and are easy to filter and search through. Developer teams can have a central log server that everyone can access. In case of a bug, team members can directly see the error messages from others and offer appropriate help. This developed concept even allows to mirror the entire production service structure in a local environment. The configuration of a Grafana dashboard or others will not be discussed any further in this paper, as this is beyond the scope.

4.4 Security Aspects of this Concept

The presented solution is oriented towards the development process and not for productive operation. For this reason, common security practices for container operation are not implemented. The images contain extra programs that are not mandatory, but convenient. The processes in the container run with super-user privileges to any avoid additional configuration effort and interruptions due to permission errors. In the `docker-compose.yml` file, passwords are defined in plain text in order to allow a consistent and easy setup process across all systems. Functions and data within the containers are made available through exposed ports on the host. Accordingly, it must be ensured that the host is not accessible

from the Internet. If certain services, such as databases or log dashboards, are shared between developers via a central server, it must be ensured that this server is only accessible within the company network in order to avoid the unintentional publication of confidential information. The same security measures must be applied for a non-containerized local development environments.

4.5 Strengths, Weaknesses and Limits

The software described in section 4.3 are already used in DevOps enabled teams. Accordingly, the entry barrier is small compared to new and unknown tools. DevContainer allow for a homogenization of development and production environments. The application runtime is identical, accordingly OS or runtime specific errors are prevented. Developers can use a ready-to-go development project setups without having to install and configure the application runtime themselves. Dependencies, keys, supporting tools like debuggers, and configurations can already be shipped within the container to enable a quick initial setup. Instead of examining the environment for a long time in the event of an error, it can quickly be torn down and recreated to a known good state. The possibility to start a large service, consisting of many applications, with one command improves the workflow and allows for much more extensive testing possibilities. These are the key features that DevContainer promise to provide, in addition to solutions to the problems described in section 3. They extend the existing toolset of agile and DevOps teams by another tool that allows developers to be more flexible and to focus more on coding.
It should be noted that DevContainers are not a perfect solution to all problems in the software development stack. Like any other tool, they come with their own set of quirks. Expertise for the use of Docker containers must be available, and the production architecture must be transformable to a local DevContainer-based configuration. This configuration must be kept up to date and maintained. Developers may need to adapt to minor adjustments in their workflow. The CI/CD solution used must always be available and provide up-to-date container images. Every virtualization approach creates an additional overhead that cannot be ignored, especially when using multiple containers on Windows based systems. Details on the exact effects are given in section 6.
As mentioned before, there are also projects, for which the use of DevContainers is not suitable. Heavy monolithic applications are not in the sense of containers and therefore not ideal for DevContainers. Graphical applications can run in containers, but require a graphical X-Server on the host, so that the resulting experience is not comparable to a native user experience (UX). Similar limitations apply to applications which require a Windows stack. Containers based on Windows do exist, but they require a Windows host, accordingly they are no longer platform-independent, require additional licenses and further adaptations. For embedded projects that require specific unconventional hardware, a virtualization approach may not feasible, since virtualization abstracts the hardware layer.
In order to demonstrate an appropriate use of DevContainers, the next bigger section describes how to migrate from a traditional development environment to a DevContainer-

based solution based on a real project. Even if this concept has not yet found much consideration in the scientific literature, there are efforts from the commercial sector, which have recognized similar problems as described in section 3 and try to offer solutions with their products. The growing number of these providers shows that there is economic interest in this area and that these firemen expect a growing market here. A selection of these commercial solutions is presented in the following.

4.6 Alternative Solutions

Apart from the concept presented here, there are already alternative solutions on the market which promise to solve problems similar to the ones described in section 3. A selection of these solutions is presented in the following section. These alternative development environments can roughly be classified into two categories: browser-based and container-based solutions. A comparison in terms of functionality of presented concepts will be in section 6.2.

Browser-Based Development Environments
Browser-based development solutions provide an editor that is entirely built upon web technologies like HTML, CSS and JavaScript (JS). Thereby, they can run on every device with a modern web browser. Their goal is to provide a quick functioning setup without any configuration.
The products CodeSandbox.io and Stackblitz implement this approach. Both solutions offer a VSCode-like editor in the browser and allows the development of NodeJS-based JavaScript projects. The functionality of codesandbox.io is provided by a client-server architecture where the browser-editor is the client and the server is managed by CodeSandbox itself. Therefore, Developers have little control over the actual runtime and always need an active internet connection. Stackblitz, on the other hand, can also be used without an Internet connection. Stackblitz is a Progressive WebApp (PWA), which brings the NodeJS runtime into the browser by using Webassembly. Accordingly, Stackblitz does not need a backend-server, because it is already running in the browser. While codesandbox.io does not provide any console at all and is completely managed by the service provider, Stackblitz provides a minimal shell for installing, copying and launching files. However, since both solutions run within the browser, regular TCP/UDP connections are restricted, due to browser security regulations. Network connections are only made available via HTTP or the HTTP-based WebSockets protocol [Cod21], [Sta21c].

Container-Based Development Environments
Similar to the solution presented in this paper, this approach uses containers in, in which everything can be installed theoretically. Compared to browser-based solutions, container-based development environments offer broader functionality and are not restricted to one specific programming language. The source code of an application and all the dependencies it takes to run it are bundled within one container. The best-known products that

implement this are Gitpod and GitHub's Codespaces, which was not released until August 2021. The infrastructure for the containers is offered by the provider, but Gitpod also offers a self-hosting option. There are no restrictions for users in the containers, additional software can be added as desired, and regular network ports can also be opened to the outside. Both solutions can be used in the browser as well as in a local VSCode installation via the remote development extension. An active internet connection is required in order to use these services. Both services are billed either on a monthly basis or on an hourly basis [Git21a], [Git21d].

5 Prototype Implementation

This section describes the implementation of the DevContainer concept in a real project that is currently developed by the Symbic GmbH. At the beginning, the initial state of the project is described as well as the goal to be achieved. Subsequently, the concept from section 4 is applied and an architecture for the DevContainer setup is designed. The following section describes the implementation process and puts special emphasis on potential errors and which catches there are to minimize them. Finally, the archived state is compared to the previously set goal.

5.1 Project Information and expected Target State

The project presented here is currently developed and maintained by Symbic GmbH. It is a system for deploying and managing Internet of Things (IoT) devices from the agricultural sector and is based on a microservice architecture. The IoT devices are used on agricultural machinery to assists vehicle operators in their work by providing an interface between the lead vehicle and trailed machinery. Vehicle and harvest information can be displayed and the position of the vehicle is tracked. New interfaces can be installed on the devices and existing ones can be adapted, while also providing a web interface to display the collected device data. Various APIs provide different functionalities, which are made available by multiple microservices. In order for the system to operate, various auxiliary services are required, including SSH-servers and MQTT-brokers. For the brevity and clarity of this paper, only a subsection of the system is presented in this thesis. This makes a further, more realistic implementation possible without going into an excessive description of the system which would have a detrimental effect on the concept of DevContainers. A detailed description of the projects architecture is given in the following section.
The goal is to test the concept described in section 4 for applicability on a real project, any encountered challenges are described below and possible solutions to them are given as well. In the end the resulting solution is compared to the initial expectations described in section 5.1.3.

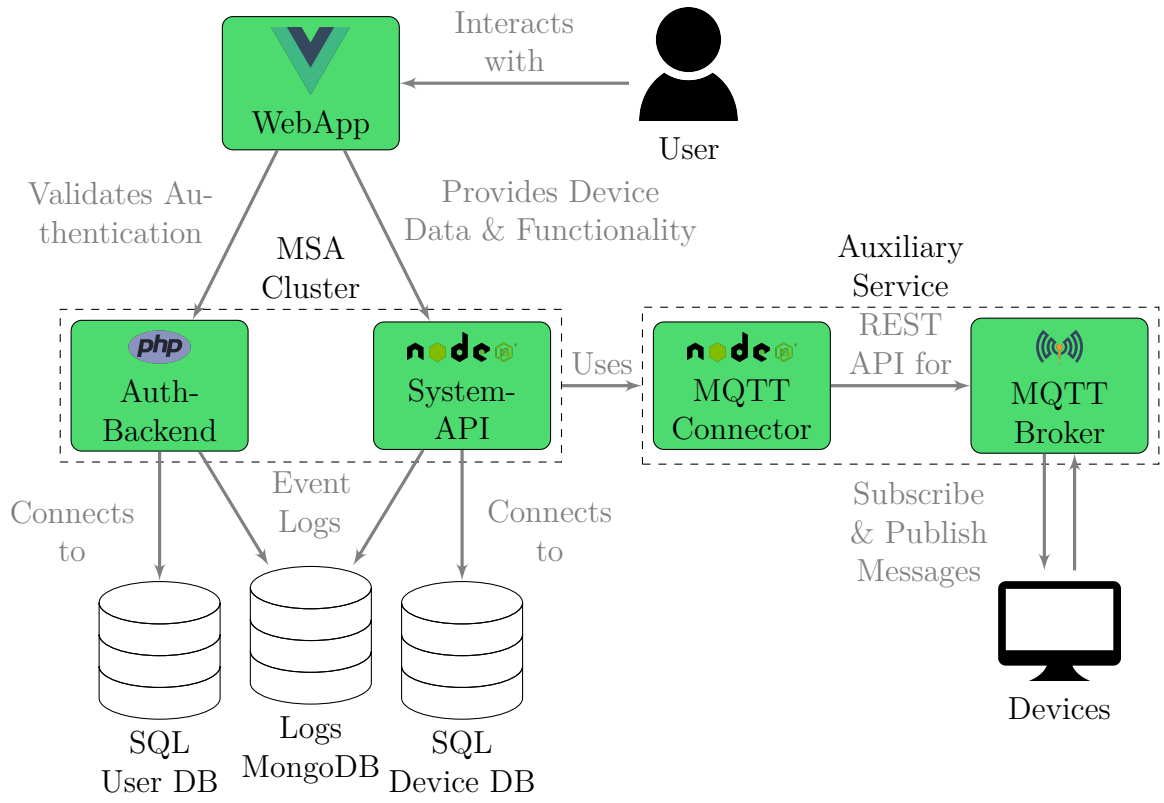

Figure 5: IoT Management Service Architecture

5.1.1 About the Prototype Project

As described above, the project builds upon a microservice architecture. This allows for the use of different technologies for individual applications. Communication between applications takes place via a platform-independent protocol. If the load on a part of the application gets critical, new instances of this application can be created with little effort. Figure 5 visualizes the architectural structure of the entire service. Each green box refers to a standalone application and contains an icon with the technology used. Users of the entire service only directly interact via the web interface (WebApp). The web interface is created with the JS framework *Vue.js*, which generates a static bundle of HTML, CSS and JS files, served by a static web-server. In order to access the system, users must authenticate themselves in the WebApp. This is done via the authentication backend (Auth-Backend), written in PHP. In addition to the authentication, this application also performs additional tasks, which will be neglected, for reasons of brevity. Provided the user is authenticated, the WebApp can display internal information and functions of all deployed IoT devices. These dynamic information are provided by the NodeJS-based System-API. All API endpoints are implemented under the compliance with the OpenAPI Specification (OAS). The OAS allows easy discoverability and understanding of the API for humans and computers. It also provides versioning support of APIs to avoid unexpected errors in other applica-

tions. These applications are part of the microservice cluster, delimited by the dashed frame, which is actively being developed by Symbic; in addition to the services listed here, the cluster consists of additional services.
Both, the Auth-Backend and the System-API, have their own SQL databases that persistently stores relevant information. However, both applications write user events to a shared event database. MongoDB is used to provide an schema flexible NoSQL database. In order to send specific instructions to, or receive information from a IoT device, the system API communicates with an MQTT-broker via the NodeJS-based MQTT-Connector. The MQTT-Connector provides an unified REST API for publishing and subscribing messages over the MQTT protocol. On a larger scale, the MQTT-Connector is consumed by several applications and is therefore a standalone application. Since only a part of this project is considered here, the System-API is the only application communicating with the MQTT-Connector. In addition to the MQTT-broker, other auxiliary services such as SSH servers are used, which are not discussed here for reasons of brevity. The MQTT-broker exposes a public endpoint offering various topics which devices can subscribe or publish to. Via this system the user can trigger a command in the web interface, which is then processed and logged by the system API. Sequentially it is made available on the MQTT-broker via the MQTT-Connector. IoT devices, that have subscribed to the corresponding topic, receive this command and execute it. The result is then published to another topic and can be accessed in the same way via the WebApp. Since the IoT devices are connected to the Internet via SIM cards, and are therefore on a restricted mobile carrier network, they are not directly accessible from the Internet. Through the described system, the IoT devices can execute commands from the outside without being directly accessible via the Internet.

5.1.2 Initial State of the Projects Development Environment

New developers joining the project have to clone all four repositories, install NodeJS 14, a specific version of the Apache, MariaDB, PHP, Perl (XAMPP) stack and must customize its configuration. Furthermore, the PHP project requires the package manager *composer*, which must be installed separately. The configuration details must be taken from the documentation or are provided by a collaborator. In order to test the whole, unified system, developers also need to install a MQTT-broker and the MongoDB database. Both applications need an initial administrator user, which needs to be created beforehand, and the corresponding credentials need to be accessible by each API, which is made possible through the creation of `.env` files - files, which is not tracked in the VCS. Then, a database and its schema needs to be created. Eventually network ports have to be adjusted and secrets for signing tokens have to be created. Thus, to set up a new environment, developers must install the all required frameworks, set up databases, customize ports, and enter connection information in the applications `.env` files. These steps are necessary to get the project running properly, whereas the setup of developer tools like debugger has not been considered yet. Even in a small project, this can take some time, especially if one is not yet familiar with the project.

Still, even experienced project members encounter challenges when working on such a project. Before the APIs can be started, it must be ensured that the databases and other auxiliary tools are already running. To start all four applications (WebApp, Auth-backend, System-API, MQTT-Connector) a developer needs four terminal sessions or a terminal multiplexer to operate all applications. The program's outputs (logs) are distributed over all four sessions and can neither be linked to each other directly, nor are they filterable, nor searchable. When everything is running, developers can contribute value to the project. The restrictions mentioned above do not generally hinder developers in their work unless an error occurs. A merge-request containing dependency or database schema updates conducted by a team member can cause the environment of others to break, resulting in a hard to debug error due to the lack of uniformity. The lack of consistent, reproducible setups is the cause for lasting troubleshooting sessions and the well-known iconic saying *"But it works on my machine"*. This mirrors the problems of heterogeneous systems mentioned in section 3. It does not matter whether the individual development environments differ from each other or the development environment differs to the production environment. The sheer fact that there are always differences is the main cause of before-mentioned problems. These selected problems emerged in just one project. When working on multiple projects, the problem scope increases accordingly. Different databases, runtimes and dependencies further contribute to a greater potential for conflicts and issues. The lack of project isolation may result in unintended side effects between projects that slow down the development process. To prevent these unnecessary slowdowns, the initial setup of an environment and its operation must be as homogeneous as possible.

5.1.3 Goal for the Target State

The goal is, by using DevContainers, a faster deployment of the project environments is archived, which is uniform among all developers and has greater similarity to the productive environment. This is made possible by the principle of virtualization, which provides a similar environment, regardless of the host. Accordingly, this should avoid system-dependent errors and eliminate the lack of reproducibility. Thus, any system can be used as host operating system, as long as it is supported by Docker. The DevContainer environment to be created is not intended to forcibly replace existing environments, but to provide an alternative solution that is also platform independent. New team members can quickly set up a working environment based on either Windows, Linux or macOS, while offering an alternative for developers with existing environments, which they can switch to gradually. Thus, a hybrid environment of the traditional development environments and DevContainer should be possible. Developers should also remain free in their choice of editors and utilities.
At any time, developers should be able to exercise control over the container runtime and the processes within the container. Thus, in the event of an error, they are independently able to replace a non-functioning container with a new one. The DevContainer environment should run locally and not require an Internet connection to external servers or services in order to be unaffected by Internet connection errors. To ease the transi-

tion to a DevContainer-based development environment, it must be possible to set up the environment with a single command, as well as to start all services with one command. Developers should not need to have explicit knowledge of how DevContainers work in order to use them.

5.2 Applying the DevContainer Approach

This section will apply the concept of DevContainer as described in section 4 to the Symbic project. First, the rough procedure of the process will be described, followed by the exact implementation and step-by-step descriptions. Problems, possible solutions and limitations of the concept will be described at the end.

5.2.1 Approach on the Project

Regarding the goal of a fully comprehensive development environment for all microservices of the Symbic project, each service will get its own DevContainer, as is intended in a microservice architecture. This allows granular control over the individual applications and makes it possible to implement a hybrid operation of conventional running applications and DevContainers. For this purpose, a `Dockerfile` file, containing all the necessary instructions for the runtime environments and other developer tools for a pre-built image, must be created for each service. The source code is not copied into the container, but later included via bind-mounts. This prevents a sudden or accidental loss of changes. Images have to be built less often and can be shared between applications with the same runtime. The orchestration of individual services is then done with Docker-Compose. Both, the configuration settings of all services, defined in a `docker-compose.yml` file, and their Dockerfiles are then stored in VCS according to the IaC principle. Using the remote extension for VSCode, developers can attach themselves to any container process and take full control over the system and the running application.
Windows is used as the starting point for the development of the setup, as this is the most widely-used operating system for development at Symbic. Once the setup is operational on Windows systems, it is tested on other operating systems, and adapted if necessary. After the Docker images have been created and tested, they will later be automatically built by the GitLab CI. Finally, scripts will be developed for the initial download of all repositories and for the database initialization. The developed interaction of the tools and the created attributes and scripts are described in detail in the following section.

5.2.2 The Implementation Process

The next paragraphs will provide a step-by-step description of the DevContainer architecture's design, starting with the Dockerfiles for the images, over the creation of the `docker-compose.yml` file, up to the usage of the environment via VSCode.

Creating the Dockerfiles

The microservices project described in section 5.1.1 is based on two technology stacks. The WebApp, the System-API and the MQTT-Connector are all based on the NodeJS runtime, whereas the authentication backend is written in PHP. Accordingly, two variants of Docker images are needed. As already described in section 2.3.5, Docker images are created from Dockerfiles. Each statement in a Dockerfile creates a new layer, which allows building upon of already existing images.

As a basis for the NodeJS DevContainer the official `node:16-bullseye` image is used as a starting point. This Debian-based images was chosen because it also used in the production environment, comes from a trusted source and is updated regularly. Accordingly, it serves as the appropriate basis for the installation of additional tools, such as git, an ssh server, build tools, and the terminal-based editors vim and emacs. As already stated, the source code is not written into the image, because it is mounted into the container via bind-mounts at runtime. The complete Dockerfile for the WebApp, the System-API and the MQTT-Connector can be found in Listing 5. The custom developed installation script used in the Dockerfile makes some recommended, low level adjustments to the system and can be found in the appendix.

```
1 # Node.js version: 16-bullseye, 14-bullseye, 12-bullseye
2 ARG VARIANT=16-bullseye
3 FROM node:${VARIANT}
4
5 # Install needed packages, yarn, nvm and other tools
6 COPY install-scripts/*.sh /tmp/install-scripts/
7 RUN apt-get update && bash /tmp/install-scripts/install.sh \
8     && apt-get -y install python3 make vim emacs git ssh \
9     && npm install -g eslint
```

Listing 5: NodeJS DevContainer Dockerfile

However, as for the PHP application, the Dockerfile is more complex, since besides the PHP runtime and a webserver several plugins are needed that rely on C/C++ libraries which are not installed by default. After its installation, the *Apache HTTP Server* must be configured accordingly to be able to use these plugins. The official PHP Docker image is used as a starting point. It is especially built for PHP applications and already contains the Apache-HTTP-Server and a package manager for installing PHP extensions. Line 2 in Listing 10 specifies the PHP version used, followed by the installation of several development tools. Subsequently, the PHP extensions used for the app and their dependencies are installed. Since this application also provides static assets, this time, the source code is copied into the image, followed by the installation of additional PHP packages with the help of the *PHP-Composer*. In the last step, the Apache HTTP Server is configured and file permissions are adjusted. The image contains all needed resources in order to start the application and also includes several development tools such as the preconfigured *Xdebug* debugger.

For both, the NodeJS image and the PHP image, the versions are passed to the Dockerfile as arguments. This makes it possible to influence the version of the used runtime when executing the build process. Thus, multiple images can be built from one Dockerfile without the need to adapt it and developers have the possibility to adapt new runtimes without much effort. During the process of creation, the images were created locally; whereas in productive use, the GitLab CI is responsible for creating regularly updated images. The job for creating the images is described in more detail at the end of this section. The images developed are very extensive in order to offer the developers the greatest possible comfort. DevContainers also work with much narrower images that can be adapted to the needs of the developers over time.

Orchestrating the Microservices

The applications from the Symbic project illustrated in Figure 5 need auxiliary services. The project requires an SQL-Database, a MongoDB-Server and a MQTT-Broker. By choosing a MySQL-Server, the Eclipse-Mosquitto-Broker and the official MongoDB-Server, free pre-built, official Docker images are available for all these applications. Listing 6 shows how these services are configured in a `docker-compose.yml` file in order to be usable for other applications. Each application is defined as a service, containing their corresponding Docker image as an attribute. In order to make the applications usable outside the virtual docker network, the internal network ports are exposed to the host system (see line 7, 16, 22 in Listing 6). To ensure that the data stored in the container is not lost when it is renewed, volumes are used to store application data persistently on the host. For this purpose, the name of the volume followed by the mount point of the volume in the container is specified in the services volumes property. Typical for Docker, the initial credentials are configured by means of environment variables. In contrast, the Mosquitto-Server is configured by bind-mounting configuration files from the `manage-reop/config` folder to `mosquitto/config`.

```
1  services:
2    db:
3      image: mysql
4      environment:
5        - MYSQL_ROOT_PASSWORD=yes
6      ports:
7        - 3306:3306
8      volumes:
9        - sql_data:/var/lib/mysql
10   mongo:
11     image: mongo
12     environment:
13       - MONGO_INITDB_ROOT_USERNAME=root
14       - MONGO_INITDB_ROOT_PASSWORD=none
```

```
15      ports:
16        - 27017:27017
17      volumes:
18        - mongo_data:/data/db
19    mosquitto:
20      image: eclipse-mosquitto
21      ports:
22        - 1883:1883
23      volumes:
24        - mosquitto_data:/mosquitto/data
25        - ./manage-reop/config:/mosquitto/config
26  volumes:
27    sql_data:
28    mongo_data:
29    mosquitto_data:
```

Listing 6: Auxiliary Services `docker-compose.yml`

Without applying the concept of actual DevContainers, the configuration in developed Listing 6 already enriches the developer experience. Developers do not have to install and configure extra auxiliary programs, but can make them available quickly, uniformly and platform-independently with one command.
In order to use DevContainers, the individual applications must also be defined as services. Listing 7 demonstrates this for the System-API: the name of the DevContainer image is used as a starting point, just like in the previous section 4. Subsequently, the credentials for the auxiliary services are made available to the application via environment variables. Since the source code is not included in the image, it is mapped by a bind-mount into the container (see line 15). Now, all the necessary requirements for starting the application in the DevContainer are fulfilled.
Since the project is a NodeJS application, all dependencies are stored locally in the project directory under the `node_modules` folder. These dependencies are operating system specific, which is why it may happen that the NodeJS application does not start when developing on a Windows host while the DevContainer is based on Linux. Developers would have to delete and reinstall all dependencies whenever the DevContainer is not in use. However, this can be avoided by using the Docker overlay file system. A volume managed by Docker is created, which overlays the local dependencies mapped via a bind-mount (see line 16 in Listing 7). This makes all dependencies used within the DevContainer independent of those installed locally. For other programming languages not storing their dependencies in the project directory, this is not needed. A volume is also used for the extensions installed by VSCode, which avoids that extensions have to be reinstalled every time the image is updated.
When the service is started, its entrypoint is executed. For the NodeJS applications, this is a custom developed script similar to Listing 8, which installs all the necessary dependencies via npm, if not already present, and then starts the application in develop-

ment mode via nodemon. The main process of the container is an infinite loop in order to keep the container running even if the developed applications exits. The complete `docker-compose.yml` file with all application services and auxiliary services can be found in the appendix under Listing 11.

```yaml
services:                          # Exemplary service
  system-api:                      # configuration
    image: system-api/dev-container
    entrypoint: "/workspace/.devcontainer/entrypoint.sh"
    environment:                   # Configure settings
      - PORT=8090                  # for auxiliary services
      - MONGO_DB_HOST=mongo
      - MONGO_DB_USER=root
      - MONGO_DB_PW=none
      - SQL_HOST=db
      - SQL_USER=root
      - SQL_PASSWORD=yes
      - SQL_DB_NAME=local-device-db
    volumes:
      - ./system-api:/workspace
      - system_api_node_modules:/workspace/node_modules
      - vscode_extentions:/root/.vscode-server/extensions
    ports:
      - 8090:8090
volumes:                           # Persistent storage
  system_api_node_modules:         # managed by Docker
  vscode_extentions:
```

Listing 7: Auxiliary Services `docker-compose.yml`

```bash
#!/bin/bash
NODE_MODULES=/workspace/node_modules && cd /workspace

# Check if modules are present
if [ -z "$(ls -A ${NODE_MODULES})" ]; then npm install; fi

# Start the main process in background & remember the pid
./node_modules/.bin/nodemon index.js&
MAIN_PROCESS_PID=$!

echo -n $MAIN_PROCESS_PID > /tmp/pid.tmp
while sleep 1000; do :; done
```

Listing 8: DevContainer `entryscript.sh`

Enabling Development
Following the procedure above, all applications are started in development mode without the need to open an editor. Changes to the code, made by any editor on the host, directly take effect to all NodeJS applications, since *nodemon* enables hot-code-reloading. This may, however, not be possible in every programming language and is not certainly supported by compiled languages.
If the developer demands more control over the application, he can open it in the DevContainer via VSCode or start an interactive shell to the container via Docker. On startup VSCode terminates the, previously automatically started, process via its Process ID (PID) and the developer now has complete control over the application via the built-in terminal. However, in order for the container not to exit, as intended when the main container process terminates, a process must continue to run to keep the container alive. For this reason, the entry-script executes an infinite loop at the end. See line 12 in listing 8. To determine which container VSCode connects to, a `devcontainer.json` file is required. Listing 4 shows such a `devcontainer.json` file. It defines the service that the developer currently wants to work on. The overall development process in a DevContainer does not differ, compared to a local VSCode instance.

Combination of all Components
Docker provides the isolation for the application runtime, Docker-Compose orchestrates the individual services, enables the integration of local code and VSCode offers a comfortable user interface. In order for this to work, it is imperative to adhere to a certain project structure, that is the same for all developers. Only if this structure is respected, the relative bind-mounts can be applied automatically, otherwise each developer would have to adjust the paths in the `docker-compose.yml` file. For the Symbic project, the directory structure can be found in the appendix in figure 7. All projects are arranged in subfolders, the `docker-compose.yml` file is made available as a system-link from the management repository. To support the Linux architecture, it was necessary to ensure that all files are checked out with the correct line endings and that the appropriate Linux function is used to create a system-link. No further adjustments were necessary. The support of macOS could not be tested due to a lack of hardware. However, since macOS is Unix-based, it can be assumed that only minor or no adjustments would be necessary.
Configuration files which are project-specific, are stored in a management repository. It also contains handy scripts to perform operations on several repositories at once, set initial configuration and apply the database schema. The GitLab CI automatically creates all needed Docker images and provides them via a private registry. Listing 9 shows the developed a `.gitlab-ci.yml` file for CI configuration using the *kaniko* executor in order to build Docker images. The executor is given the repository context, the Dockerfile and which variant of the container should be built. The build process is executed on every push-event to the GitLab repository.

```
build_dev_container:
  image: gcr.io/kaniko-project/executor:debug
  variables:
    DOCKERFILE: $CI_PROJECT_DIR/.devcontainer/Dockerfile.dev
    BUILD_IMAGE_TAG: $CI_REGISTRY_IMAGE/dev-container:latest
    NODE_VARIANT: 14-buster-slim
  script:
    - /kaniko/executor --context $CI_PROJECT_DIR
    --build-arg NODE_VARIANT=$NODE_VARIANT
    --dockerfile $DOCKERFILE
    --destination $BUILD_IMAGE_TAG
  rules:
    - if: $CI_COMMIT_SOURCE == "push"
```

Listing 9: GitLab CI Build-File for Docker Images

By adding a logging parameter to the `docker-compose.yml` file in listing 11, all application logs can also be sent to a dedicated system to display them graphically. In the implemented project of Symbic this is achieved by a *Grafana* instance and the *Loki* log-shipper. Since the availability of a central, searchable log dashboard is not mandatory for the development with DevContainers, it will not be discussed further here.

5.2.3 Encountered Challenges and Limitations

During the creation of the DevContainer architecture some challenges arose. The biggest challenge consisted the differences between a Windows- and a Linux-based system. Since the source code is mapped into the container via bind-mounts and not bundled within the image, all mounted files have the `CR` line ending. Linux programs like Bash and SSH can not read this format, therefore their execution fails. Creating an `.gitattributes` file with the following content solves the problem.

```
* text=auto eol=lf
*.{cmd,[cC][mM][dD]} text eol=crlf
*.{bat,[bB][aA][tT]} text eol=crlf
```

All files (except `.bat`/`.cmd` scripts) will be checked out with the Linux `LF` line ending. These are supported by most editors including editors focused on Windows.
Another challenge was the lack of isolation of local NodeJS dependencies. Deleting the `node_modules` folder and reinstalling all dependencies, is not a reasonable solution in case developers switch between the local- or the DevContainer environment. The overlay file system can overshadow individual folders or files that are mounted via bind-mounts. This way, operating system-specific files are being isolated. However, the use of non-Linux focused file-systems create some problems. The NodeJS applications use nodemon to reload the application whenever file changes are made, in order to apply these changes

immediately. Nodemon monitors the file system for changes, just like other hot-reloading solutions. However, due to the fundamental differences between an NTFS file-system and the Docker overlay file-system, this monitoring does not work by default. To enable this feature, an additional parameter must be specified so that nodemon actively polls for changes. With this parameter, hot-reloading works, but requires more processing power for active file-polling.
To make the usage of DevContainer as comfortable as possible, all applications are started simultaneously, with the container start likewise. However, since Docker containers only run until their initial process is terminated, a wrapper had to be created, which starts the application process, but does not terminate when the developer takes control of the application. Listing 8 depicts a short Bash script as the workable solution. The application process is started in the background, while the main process is an infinite loop.
Depending on the project, different technologies with different requirements and different upcoming solutions are used. This can be seen when comparing the very simple Dockerfile for the NodeJS applications with the way more complex Dockerfile used for the PHP application. For similar applications, such as the various NodeJS applications above, a Docker image can be reused and used as a template. When using other technologies, the implementation will have to be adapted accordingly. If templates for common programming languages are already available, they can be used without much effort, only the mount points in the containers have to be adapted.
Although there have been challenges in the creation of the architecture and the individual services, these have been solved with minor adjustments. However, the additional system load on Windows due to containerization could not be solved. On a Windows based host, the entire Linux kernel has to be virtualized. For a convenient file accesses between Docker and Windows, further load applies, due to necessary conversion between filesystems. This is even amplified by the active searching of file changes by nodemon. I/O heavy tasks, such as the installation of many `node_modules` cause a noticeable overhead on the system. More applications and more demanding the applications, increase the overhead and the load on the system. The impact of this drawback is further described in section 6.
During the development, bugs have also been encountered when using Docker Desktop on Windows. A memory leak caused the system memory to reach capacity when Docker images where created repeatedly. Although this can be prevented by modifying a specific setting on the WSL system, it is not a permanent solution to the problem. Apart from that, Docker Inc. changed their licensing model for Docker Desktop during the writing of this thesis, so that, now, its only free of charge by companies with less than 250 employees and less than $10 million in revenue [Doc21a]. However, this does not apply to Docker Community Edition.

5.3 Final State

The newly created development environment provides a uniform environment across multiple systems regarding the operating system used. Thanks to virtualization, the great-

est possible similarity between the proposed environment and productive environment is achieved. Auxiliary processes no longer have to be created and configured individually as their preconfigured definitions are available in the repository. Through the use of Docker, software, which were only available on Linux beforehand, is now also available on Windows and macOS. By uniformizing the environment through virtualization, configuration-related errors become reproducible, traceable and can be avoided.
Developers are not compelled to only use the new DevContainer environment, as a hybrid use of local applications and container applications, has been achieved. The functions of container applications are available through exposed ports on host. Figure 6 shows several containers, each running one application, that can communicate with each other within their docker network. The source code for these applications is stored on the host system and is mapped into the containers by the Docker-Engine via bind-mounts. The network ports provided by the applications are forwarded by the Docker-Engine to the host system. This way, developers can use locally installed programs, in order to access the network functions of the applications. The source code can be modified by using any editor on the host, thanks to the bind-mounts. Using the VSCode editor, developers can even connect to a container, as shown in Figure 4, and thus use all the additional programs available within the container. This allows hybrid operation of applications using local and virtualized environments. Gradually, the DevContainer principle can be applied to one application after the other. The command `docker-compose up service_1 service_2 ... service_n` only starts the explicitly specified applications as a container, while all other applications can still be executed locally. Explicit changes to the applications in order to use DevContainers were not necessary.

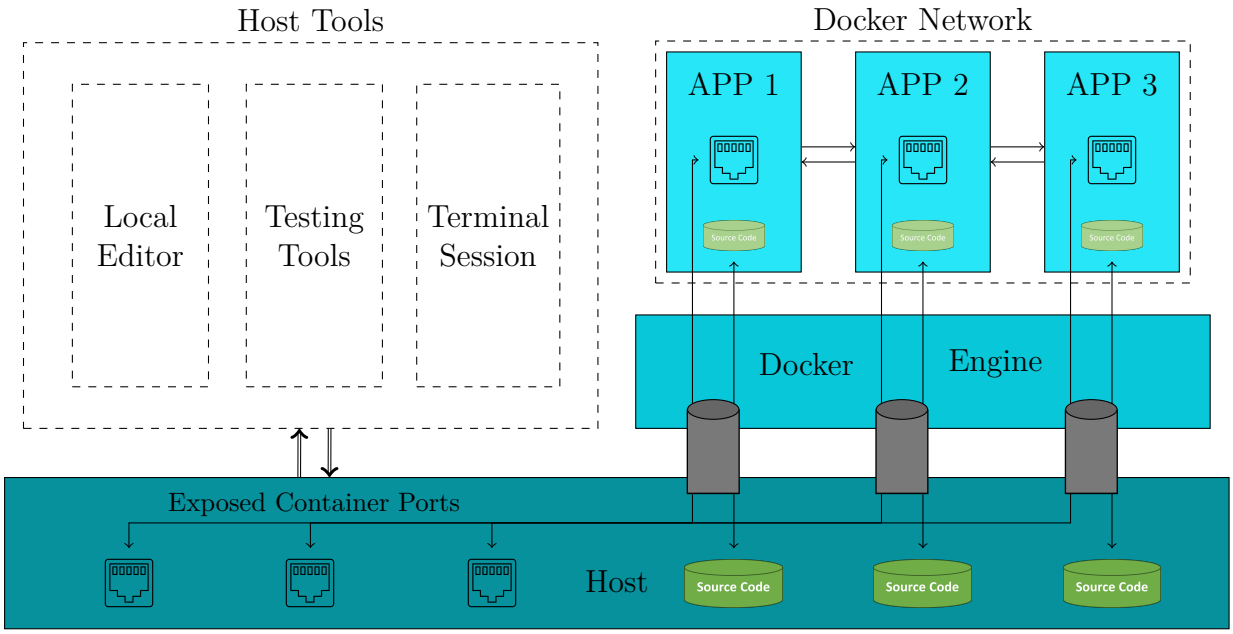

Figure 6: Development Architecture with DevContainers

However, an overhead due to the use of the DevContainer was noticed during use, especially on a Windows based system. The effects of this additional load are discussed in the next section along with the presented advantages of the DevContainer environment.

6 DevContainer Analysis and Evaluation

The following section discusses the usability of DevContainers and compares them to alternative development solutions. The metrics considered for this comparison are described below. These are used in order to evaluate the usability and to determine the success of the applied concept.

6.1 Considered Metrics

The evaluation is divided into two sections. First, it is examined whether DevContainers were able to solve the problems described in section 3. Second, DevContainers are compared to alternative solutions, some of which have only recently been released. For this purpose, the alternatives described in section 4.6 are taken into account.
For the evaluation of the Symbic DevContainer prototype, several metrics are taken into consideration. One of the first things to look at is the number of steps needed in order to perform a full deployment of all necessary components to set up a developers machine for development. This approximates the expected effort for new developers to get a working project setup. Doing so, platform-specific steps are considered as well, so is the choice of available (development) software. Subsequently, the interaction with the environment is analyzed. The usability of the environments is considered as well, the decisive factor is the free choice of editors and interaction possibilities with the application to be developed. The system-load analysis clarifies whether the properties of the DevContainers have a significant impact on the development process. Finally, the error behavior, (dependency) configuration setup, reproducibility capabilities and available testing possibilities are considered.
The function rage and control capabilities represent the main points of comparison when contrasting DevContainers with the alternative development environments mentioned in section 4.6. This includes several aspects, such as the control over the environment, supported programming languages, the type of service provision and the pricing model.

6.2 Evaluation and Results

This section examines the points mentioned above concerning the metrics and data collection. This examination is divided into two sections. First, the findings for the prototype implementation of the DevContainer concept is evaluated. The native development environment is directly contrasted with the DevContainer environment. Then, the DevContainer environment is, albeit only argumentatively, compared to, both browser-based and container-based, alternative solutions. A direct comparison between DevContainers and

the alternatives was not possible, due to lack of access to the full function range of the alternatives and agreement enclose for the project source code.

6.2.1 Evaluation for the Prototype Implementation

The following section is an evaluation of the applied DevContainer concept with respect to the problems described in section 3.

The Initial Setup Process
The initial setup process is significantly simplified using DevContainer. Databases and auxiliary services are provided by isolated containers, so there is no need to install them manually. Language runtimes, compilers and dependencies are provided with the DevContainers images. The specific versions of every software used are stored as code, so that, similar to the PaaS principle, the entire development environment is made available as a uniform platform. There is only one uniform way for software installation.
Table 3 shows a comparison of all the steps, which need to be performed by new developers in order to create a working environment. The initial cloning process of the repositories and installation of an editor is necessary in both approaches, but while the native development approach requires all program components to be installed individually, the DevContainer environment only requires Docker to be installed. Accordingly, in this example less than half of the steps are necessary in order to set up a development environment. Table 3 also takes into account whether the respective steps are independent of the operating system. All (dark) green highlighted steps are independent of the operating system used. The steps displayed with a gray background are operating system specific and are therefore potentially different for each OS and, additionally, more prone to errors.
Thus, DevContainers make it easier and more efficient to set up a new working environment. The runtime and settings of all applications are in a defined good state, independent of the host system and without conflicts to other projects. After installing Docker, multiple applications can be started simultaneously by executing `docker-compose up`.

Dependency Management
Due to the firmly defined Dockerfiles, the version of the programming languages used is always consistent among all developers. Errors caused by different library versions are thus avoided. If a new major program version is to be tested, developers can simply set the `VARIANT` variable accordingly and restart the DevContainer with the `--build` flag of Docker Compose (`docker-compose up --build`). A new image is crated automatically and the application is launched with the new runtime without affecting other applications, projects or the host. This strict isolation of individual software versions greatly simplifies the simultaneous operation of multiple versions and prevents errors. Working on different projects with conflicting dependencies is thus made possible.
In case an error occurs anyway, it can be fixed quickly by discarding the existing DevContainer, reverting all changes and recreating a new container from a known good

Step	Native Development	DevContainer
1	Initial checkout of the project	Initial checkout of the project
2	Install Editor of choice	Install Editor of choice (VSCode preferred)
3	Install NodeJS	Install Docker
4	Install the XAMPP stack	Start all services with docker-compose
5	Install PHP Composer	
6	Install MQTT Mosquitto	
7	Configure the XAMPP stack	
8	Configure MQTT Mosquitto	
9	Configure each microservice	
10	Start each services with npm	

Table 3: Steps to Initialize the Symbic Project

image. Changes to the source code are not lost, since these are stored on the host and are made available by bind-mount in the container. Since all relevant configurations are tracked in VCS, changes can be quickly detected and be undone.
The number of possible problems is significantly reduced by the use of DevContainers. Furthermore, the DevContainer environment is quite similar to the configuration in productive operation. Operating system specific errors, such as different file system separators, line endings and libraries that may behave differently between Linux and Windows, are avoided. Developers can focus on new features for an application and do not have to consider as many edge cases when running and testing their application.

Lack of Testing Options
A major challenge of a microservice architecture is the increased configuration overhead between applications and the associated lack of testing possibilities for the entire service. Apart from unit tests, developers can not test the successful integration of individual applications due to the immense configuration effort. This is why this job is often performed by CI systems. However, these systems can only detect errors after the code has been pushed to the VCS system and all, possibly lengthy, tests have been completed. This late detection of errors slows down the development process and increases its cost.
However, thanks to DevContainers and the orchestration of individual services by Docker Compose, all relevant applications can be launched simultaneously, and are preconfigured to successfully interact with each other. Within a short time, developers can test whether cross-application communication is successful and can reproduce user processes in concrete terms. Thus, end-to-end tests can already be performed on the developers' machines. Errors can be detected before the code is checked into the VCS and can hence be fixed earlier. Through the use of containers, even more comprehensive testing tools are available. Yet, the concept of DevContainers do not replace testing conducted by the CI. Load tests and acceptance tests cannot be carried out on the developer's systems, since its results have no relevant significance.

Performance Evaluation

Although containers have a significantly lower performance overhead than VMs [MKK15], a performance loss is still noticeable when using the implemented prototype from above. To quantify this, performance metrics of the individual applications from the prototype were collected and the difference between direct execution on the host and execution in the container were considered.

The evaluation was performed on a laptop with an Intel Core i5-10210U CPU, 16 GB of RAM and an NVMe SSD. Windows 10 Pro 21H1 and Debian 11 (Bullseye) were used as the testing operating systems. On Windows, the running processes were written to a log file with the command line tool `Get-Process` every 10 seconds for a duration of 30 minutes. When using Docker Desktop (Version 20.10.8), the tool `docker stats` was also used to obtain the resources used per container in addition to `Get-Process`. On Linux, the `top` program was used to monitor the processes for the same duration, and `docker stats` was used again when running the applications inside `docker-ce` (Version 20.10.9). The link to the entire evaluation script can be found in the appendix. The results were then processed and the arithmetic averages are shown in Table 4 and 5.

Considering the results for Windows (Table 4), the sum of the individual process load of all applications running directly on the host (NodeJS, MariaDB, Apache) is larger than the load of the Docker process when running the same applications as containers (25% vs 15%). The approximately 13% CPU usage from the combined containers matches the 15% usage of the Docker process except for a few percent overhead. Contrary to this, two of the three containerized applications consume more RAM than when running directly on the host. Docker's usage of 4 GB of RAM can be explained by a setting in WSL. Since Windows has to virtualize the Linux kernel, the micro VM uses all the available memory allocated to it, even if the individual containers, with a commutated demand of 300 MB, do not actively use all the available RAM.

The performance results on Linux barely show any differences wherever the application is directly executed on the host and or inside a container. The CPU utilization is exactly the same and even the RAM usage is quiet similar. The lower RAM usage of the Apache webserver within the container might be caused because the container image has fewer plugins enabled by default. The overhead of the docker engine is less than 200 MB RAM and about 1% CPU utilization. When comparing the results from Windows and Linux, it is noticeable that not only the overhead of the Docker-Engine is significantly lower, but the all container processes also use significantly fewer CPU resources under Linux. The usage of DevContainer under Linux was indistinguishable from the native execution of the processes. The main reason for this is probably the specific optimization of Docker for Linux and, in contrast to Windows, the omission of the kernel to be emulated.

In addition to the CPU and RAM usage, the read and write speed of the disk was also considered. While the installation of npm packages on Linux was almost identical when and when not using docker, there are significant differences under Windows. The direct installation of npm packages on the host was twice as fast as the installation in the container using bind-mounts. Using volumes for the packages, as suggested in section 5, the installation was still 50% slower than without using DevContainers. It is the poor I/O performance

	Win_RAM	Win_CPU	Win_Docker_RAM	Win_Docker_CPU
nodeJS	108.5 MB	16%	92.2 MB	11%
MariaDB	29.8 MB	<1%	111.9 MB	<1%
Apache	19.9 MB	8%	27.2 MB	<1%
Docker	4078 MB	15%	-	-

Table 4: CPU and RAM usage of Processes on Windows with and without using Docker

	Linux_RAM	Linux_CPU	Linux_Docker_RAM	Linux_Docker_CPU
nodeJS	65.2 MB	1%	65.0 MB	1%
MariaDB	100.2 MB	<1%	106.0 MB	<1%
Apache	15.7 MB	<1%	10.6 MB	<1%
Docker	185.6 MB	1%	-	-

Table 5: CPU and RAM usage of Processes on Linux with and without using Docker

when using many small files, such as npm packages, that is the biggest limitation of using DevContainers on Windows. The usage of 20 containers, on the hardware described above, CPU and RAM resources have not been the limiting factor under Windows. Only the poor NTFS file system performance in combination with the Docker overlay-filesystem and the aggressive scanning behavior of the Windows integrated virus scanner made for a slower user experience in practice. On Linux, no limitations could be found in terms of performance.

6.2.2 Comparison to Alternative Development Solutions

While the concept of a local containerized development setup is new, the existence of alternative approaches is not. The solution presented in this paper is compared to browser-based and container-based development environments of various vendors. The results of the following comparison is summarized in Table 6 at the end of this section. The evaluation is based on the available documentation and experiences made while testing these solutions at the time of writing this thesis.

Comparison to browser-based Development Solutions
The major distinction between the solutions presented in this thesis and browser-based development environments is the scope of supported programming languages. The alternatives Codesandbox.io and Stackblitz, mentioned in section 4.6, only support NodeJS projects based on JavaScript or its superset TypeScript. Accordingly, the choice of programming languages is very limited, as is the ability to run multiple services in parallel. Both services lack the possibility of orchestration and the possibility to install any additional utilities, which greatly limits the control and therefore the features of these services.

Furthermore, not all functions of NodeJS are supported. External packages based on native OS libraries or binaries, like the popular task-runner `grunt`, often do not work. This is especially true for packages based on encryption libraries, like the SSH-client `ssh2`. Control over created network sockets is either missing or limited by the browser runtime. A command-line interface is also missing (Codesandbox.io) or very limited (Stackblitz). Both solutions do not support debugging. Due to the nature of a browser window, the available space and input methods to the editor are limited, due to an omnipresent preview window and, in addition to that, common key combinations, such as `F1` collide with the browsers keyboard shortcuts.
The advantage of this solution is that users have a working setup within seconds, due to the variety of available templates. Users can edit code from any device without the need to install any additional programs, because these alternatives run in the browser. This makes this solution attractive for teaching purposes and fast prototyping of frontend projects.
Since Codesandbox.io, unlike StackBlitz, runs the code on their servers, Codesandbox.io has access to the entire source code and network traffic, which may violate confidentiality clauses. Their free pricing model only allows for publicly accessible development sessions and does not support private `npm` package repositories. Paid sessions vary from $9 to $39 for Stackblitz and $30 to $50 for Codesandbox.io per user per month [Sta21c], [Cod21].
These solutions also try to solve problems in modern development environments, but limit themselves to a specific scope. The limitations listed here show that only small NodeJS projects or purely frontend projects can be implemented with Codesandbox.io and Stackblitz. This is supported by the fact that the majority of templates are built for frontend frameworks. The narrower support for backend features and the missing support for auxiliary services make these alternatives not suitable for professional development.

Comparison to container-based Development Solutions
The alternative container-based development environments Codespaces and Gitpod show distinct similarities to the solution presented in this thesis. Both solutions rely on the isolated provisioning of predefined runtime environments via container-based virtualization. These environments are likewise built upon a Linux kernel and can thus provide a development environment for any programming language supported on Linux. At the time of writing, projects requiring a Windows or macOS host are not supported.
Gitpod is integrated in various VCS-providers and offers the possibility to create a VSCode instance within the browser. A local VSCode instance can also be used in order to connect to a running container, which bypasses the limited keyboard shortcut support of a browser. Codespaces offers the same possibility, except that this service only works with GitHub.com. Within a container, developers have full control over the system. Additional applications can be installed, debugging is possible and ordinary TCP/UDP ports can be exposed to the internet or to a connected VSCode client. Codespaces, Gitpod and the DevContainers require that the container environment must be manually defined and optimized once. The time required is determined by the size and complexity of the project.
However, Codespaces and Gitpod differ in the way individual applications are organized.

Both alternatives focus on the provision of one development environment for one application, while the DevContainer concept specializes in the simultaneous usage and interaction of multiple applications. While one container can contain any number of services, this makes the container bulkier, more resources hungry, and less reusable. Generally, containers are intended to run one application only, additional applications should be available via APIs, this makes containers lightweight and flexible. While Codespaces can enable a development environment for massive applications like the monolithically structured GitHub.com service (13 GB repository, built-in DB and many dependencies) when given 32 CPU cores and 64 GB RAM [Git21b], this is not the most efficient and flexible solution since this approach requires many rebuilds if a container changes. Often developers only need to work on one part of a project, so that the massive compute resources are not needed. In particular, GitHub struggled with these massive containers primarily because of the deployment time of their containers, yet, through intense and extensive optimization, their developers were able to reduce the Codespace creation time from 45 minutes to less than a minute. This was only possible because their CI constantly builds new, updated container images so that there is always a buffer with ready to deploy images. These immense computing resources are not available in smaller companies. The provisioning time of the DevContainer solution is about 10 seconds, provided that the images are already downloaded once. Another difference is that the solution presented here allows a hybrid operation between local applications and DevContainer applications. This is not supported by Codespaces and Gitpod. Companies are confronted with the major decision on whether to invest heavily into one service and lack the option to switch gradually to a new development strategy.
Furthermore, the presented solutions differ in respect of the choice of the available editors. Although the use of VSCode is recommended for DevContainers, the use of any other editor is possible, unlike with Codespaces and Gitpod. With Codespaces and Gitpod, the container infrastructure is provided by the respective providers, which is why a permanent Internet connection is required and the compliance with confidentiality agreements must be fulfilled. Both services offer options for self-hosting, but these are only included in higher-priced subscriptions and still do not run on the developers machines. Free versions of these services either offer only limited computing time, such as 50 hours per month at Gitpod, or the computing capacity used is billed per minute. The paid offerings range per month and user from $9 to $39 for Gitpod and from $4 to $21 for Codespaces. The DevContainer solution presented here is free on Linux and free on Windows and macOS, as long as the company size does not exceed 250 employees or a turnover of $10 million when using Docker Desktop. Beyond these numbers, Docker Desktop prices vary between $5 and $21 per month and per user [Git21d], [Git21b].
Structurally, all three solutions are similar in various points, the usability of Codespaces is superior due to its direct integration in GitHub.com and the resulting comfort. The massive hardware resources of Codespaces and Gitpod are not comparable to those of a local workstation. The biggest disadvantage of these solutions is the dependence on external service providers and the associated significantly higher costs. The existence of these solutions and the increasing number of providers conclude the potential of efficient and

uniform development environments. Codespaces has only been officially presented during the preparation of this work (August 2021) and is still at an early stage. The choice for the most suitable solution for a specific project depends on the respective requirements. Since none of the three solutions relies on proprietary configuration files, the same Dockerfiles can be used for DevContainer, Codespaces and Gitpod. This enables easy migration between these solutions.

6.3 Summarized Evaluation

As seen above, DevContainers offer a tremendous opportunity to solve problems during the development process. Development environments become more isolated, uniform and much easier to set up and maintain. Commercial solutions such as Codespaces and Gitpod show that these companies see potential in using pre-defined development environments.
The solution presented in this thesis is able to solve the problems described in section 3 to a large extent and therefore can be classified as successful. Although the creation of a DevContainer environment increases the configuration effort in a short term, in the long term fewer different configurations setups need to be supported. The performance evaluation has shown that Docker introduces a performance overhead on Windows, nevertheless applications, executed on Linux often require less CPU capacity compared to Windows. On modern computers with sufficient memory, running multiple applications on Windows is possible without any real limitations. Only the I/O performance of the storage-device is a limiting factor, which is why fast SSDs should be used. On Linux, no limitations were found in terms of performance.
The possibilities of virtualization in development are enormous. Suddenly, a much larger selection of programs, which are already preconfigured, is available. Even if DevContainers are not used for the applications developed, the platform independent, simple provisioning of databases and auxiliary services is an immense advantage.

Possible Improvements
Docker enables access to software, which were previously exclusively available in the server world. This software allows to further improve the concept presented above. Instead of directly exposing application ports, a reverse proxy can be placed in front of every application. Only the port of the proxy is then exposed to the host. This allows developers to access their frontend by requesting `frontend.myapp`. The proxy than automatically forwards the requests to the appropriate application. Accordingly, Developers do not need to remember application ports or have to pay attention to port collisions on `localhost`. The reverse proxy routes all requests thought the internal docker network to the appropriate application. This is another strategy that is often used in a production environment, making the DevContainer environment even more similar to the production one. Furthermore, to minimize the load on the developer's machine, static services can be made available on a shared company-internal servers. The MQTT-broker or the SSH server may not be needed on every machine and can be shared.

7 Conclusion

In the course of this thesis, basic definitions and practices of modern programming methodologies were described. Concepts form the DevOps culture were highlighted and the microservice architecture was described in more detail. By taking a closer look at the workflows for developing such a microservice or comparable architectures, various problems were identified that slow down the development speed and cause an increased error potential. The origin of these problems is mostly the heterogeneity of different systems and their manual configuration. Especially with regard to cloud native microservice applications, it has become clear that production great environments differs significantly from commonly used development setups. Differing operating systems, configurations and a limited testing setup are responsible for unexpected errors.
This thesis has highlighted these limitations in current development environments and pointed out that there is potential for improvement. With the use of software from the DevOps and server world, a concept was created that addresses these shortcomings in local development environments. By abstracting the operating system through container technology, it can be ensured that applications can be reliably executed in well-defined consistent environments. This eliminates platform specific errors and enables a homogeneous development and production environment. The widely used container tool Docker was used implement this. To avoid configuration errors and manual tasks, all configuration-specific settings have been standardized and are stored in a VCS via a common file format. This allows multiple applications to be orchestrated reliably and seamlessly without any manual effort. Several tools have been used in order to provide developers with a fast, error-resistant and familiar development setups, which also allows the use of local, non-container based, programs.
The implementation of the concept to a prototype has shown that it is quite applicable in the real world. Depending on the application, adjustments to the settings of the developer tools may be necessary in order to use functions such as hot-reloading. A measurable overhead of this concept could also be determined in the performance evaluation when the concept is applied to Windows hosts. However, the time saved by the automatic management of the application processes and their environment, as well as the significantly reduced potential for errors and the greater choice of software, demonstrate the success of this concept.
The evaluation has further shown that the concept presented here is quite relevant for large software companies. The comparison with the, still quite new, commercial alternatives Codespaces and Gitpod shows that these companies see a market for platform-independent, pre-configured development setups. This concept allows to actively develop software with any internet capable device, regardless of the computing capacity, instead of focusing on the configuration and maintenance of one's setup.

Further Potential
All solutions presented here are still in a very early state (of their development). Gitpod was announced in 2019, while Codespaces was released in 2021, during the writing process

of this thesis. Like DevContainers, these services promise to provide a uniformly managed development infrastructure. This concept intends to allow developers to focus on contributing to the project and is meant to prevent interruptions due to maintenance and errors in their own working environments. Through the power of cloud computing, computing capacity can be booked at will in order to provide products and services independent of the underlying hardware. In the event of an application instance failure, it is simply discarded and rebuilt the instance. Now this is also possible with containerized development environments. In the case of Codespaces and Gitpod, the available computing capacity can even be increased without having to purchase new devices for developers. This kind of external hosting allows the usage of any device for development. The increasing number of mobile devices now can connect to powerful development servers. This offers new possibilities for mobile working setups. DevContainers are also quite suitable for other development platforms. With the tool `docker buildx`, developers can compile code for other processor architectures such as ARM on their primary x86 bases system. Developers in the embedded world can develop and test their software with only one device without the need to manually copying data to an external additional device.
The concept of DevContainers offers considerable opportunities for opensource projects that rely on contributions of voluntary developers. Projects with extensive and complicated setup requirements can be discouraged and thus miss out on valuable contributors and employees. For such projects, DevContainers, with a predefined environment which works independently of the host, are an easy entry point in order to offer volunteers a working project environment. While the limitations of browser-based development environments can be too restrictive for professional developers, they still offer potential for the use in education. Pupils or students can have a working environment within seconds, in which they can gain hands-on experience with tutors, regardless of used device.
Based on this work, the question arises for a comprehensive study of the usability of these container-based development techniques. This study should particularly take into account a possible gain in productivity and the different cost models of all solutions.

References

[BBVB+01] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.* "Manifesto for agile software development." *Agile Manifesto*. 2001. Available: https://agilemanifesto.org

[BvdG20] H. Been and M. van der Gaag. *Implementing Azure DevOps Solutions: Learn about Azure DevOps Services to successfully apply DevOps strategies*. Packt Publishing, 2020. Vol. 1. ISBN: 978-1-78961-969-0.

[BWZ15] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015. Vol. 1. ISBN: 978-0-1340-4988-5.

[Che17] G. Chen. "The Critical Role of Virtualization in Modern Datacenters, Hybrid Clouds, and Containers." *IDC White Paper*. July 2017. Doc-id:#US42903817.

[Cod21] CodeSandbox BV. (2021) *CodeSandbox Docs*. Visited on: 2021-08-20. Available: https://codesandbox.io/docs

[DD16] J. Davis and R. Daniels. *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale*. O'Reilly Media, Inc., 2016.

[Doc21a] Docker, Inc. (2021) *Docker Blog - Updating Product Subscriptions*. Visited on: 2021-09-10. Available: https://www.docker.com/blog/updating-product-subscriptions/

[Doc21b] Docker, Inc. (2021) *Docker DOCS Reference*. Visited on: 2021-05-16. Available: https://docs.docker.com/reference

[DZo19] DZone. (2019) *Microservice Deathstar Created when Configuration Management Shifts to Runtime*. Visited on: 2021-10-11. Available: https://dzone.com/articles/navigating-the-microservice-deathstar-with-deployh

[Git21a] GitHub Inc. (2021) *GitHub Docs*. Visited on: 2021-08-11. Available: https://docs.github.com/en/codespaces

[Git21b] GitHub Inc. (2021) *GitHub's Engineering Team has moved to Codespaces*. Visited on: 2021-08-11. Available: https://github.blog/2021-08-11-githubs-engineering-team-moved-codespaces

[Git21c] GitLab Inc. (2021) *GitLab DOCS Reference*. Visited on: 2021-09-03. CommitRef: 28c1cf25. Available: https://docs.gitlab.com/ee/ci/

[Git21d] Gitpod GmbH. (2021) *Gitpod Docs*. Visited on: 2021-08-20. Available: https://www.gitpod.io/docs

[Inc19] A. S. Inc. “Developer Survey - Open Source Runtime Pains 2019.” Open Source Languages Company. Tech. Rep. 04 2019. Visited on: 2021-08-27. Available: https://www.activestate.com/resources/white-papers/developer-survey-2019-open-source-runtime-pains

[IS18] K. Indrasiri and P. Siriwardena. *Microservices for the enterprise*. Springer, 2018. ISBN: 978-1-4842-3858-5.

[Joy15] A. M. Joy. “Performance comparison between linux containers and virtual machines.” in *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE, 2015. pp. 342–346.

[KLT16] H. Kang, M. Le, and S. Tao. “Container and microservice driven design for cloud infrastructure devops.” in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016. pp. 202–211.

[Kou18] P. Koutoupis. “Everything You Need to Know about Linux Containers, Part I: Linux Control Groups and Process Isolation.” *Linux Journal*. August 2018. Available: https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process

[Mic21] Microsoft Corp. (2021) *Create a development container*. Visited on: 2021-05-16. Available: https://code.visualstudio.com/docs/remote/create-dev-container

[Mie20] A. Miesen. *Ansible: Das Praxisbuch für Administratoren und DevOps-Teams*. ser. Rheinwerk Computing. Rheinwerk Verlag GmbH, 2020. Vol. 1. ISBN: 978-3-8362-7660-3.

[MKK15] R. Morabito, J. Kjällman, and M. Komu. “Hypervisors vs. lightweight virtualization: a performance comparison.” in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015. pp. 386–393.

[ÖK20] B. Öggl and M. Kofler. *Docker–Das Praxisbuch für Entwickler und DevOps-Teams*. ser. Rheinwerk Computing. Rheinwerk Verlag GmbH, 2020. Vol. 2. ISBN: 978-3-8362-7226-1.

[Por12] M. Portnoy. *Virtualization essentials*. John Wiley & Sons, 2012. Vol. 19.

[Pyt21] Python Software Foundation. (2021) *Virtual Environments and Packages*. Visited on: 2021-08-12. Available: https://docs.python.org/3/tutorial/venv.html

[Red21a] RedHat Inc. (2021) *Containers vs VMs*. Visited on: 2021-06-08. Available: https://www.redhat.com/en/topics/containers/containers-vs-vms

[Red21b] RedHat Inc. (2021) *What are microservices?* Visited on: 2021-06-10. Available: https://www.redhat.com/en/topics/microservices/what-are-microservices

[Red21c] RedHat Inc. (2021) *What is virtualization?* Visited on: 2021-06-08. Available: https://www.redhat.com/en/topics/virtualization/what-is-virtualization

[Ric18] C. Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018. Vol. 1. ISBN: 978-1-6172-9454-9.

[SB08] M. Sliger and S. Broderick. *The software project manager's bridge to agility*. Addison-wesley professional, 2008. ISBN: 978-0-321-50275-9.

[Sli21] Slintel Inc. (2021) *Docker Market Share*. Visited on: 2021-08-04. Available: https://www.slintel.com/tech/containerization/docker-market-share

[Sta21a] Stack-Exchange Inc. (2021) *Stack overflow developer survey 2018*. Visited on: 2021-08-06. Available: https://insights.stackoverflow.com/survey/2018

[Sta21b] Stack-Exchange Inc. (2021) *Stack overflow developer survey 2021*. Visited on: 2021-08-06. Available: https://insights.stackoverflow.com/survey/2021

[Sta21c] Stackblitz VC. (2021) *Stackblitz Docs*. Visited on: 2021-08-20. Available: https://developer.stackblitz.com/docs/platform/

[Sta21d] Statista Inc. (2021) *Revenue of the software market worldwide from 2016 to 2025*. Visited on: 2021-07-29. Available: https://www.statista.com/forecasts/954176/global-software-revenue-by-segment

[TSM19] M. Z. Toh, S. Sahibuddin, and M. N. Mahrin. "Adoption issues in devops from the perspective of continuous delivery pipeline." in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*. ser. ICSCA '19. Association for Computing Machinery, 2019. p. 173–177.

[Ver21] C. VersionOne. "15th annual state of agile report." *collab. net*. 2021. Available: https://stateofagile.com/#ufh-i-661275008-15th-state-of-agile-report/7027494

[W3T21] W3Techs. (2021) *Usage statistics of operating systems for websites*. Visited on: 2021-08-16. Available: https://w3techs.com/technologies/overview/operating_system

[WSCC17] B. Wang, Y. Song, X. Cui, and J. Cao. "Performance comparison between hypervisor- and container-based virtualizations for cloud users." in *2017 4th International Conference on Systems and Informatics (ICSAI)*. 2017. pp. 684–689.

Appendix

A Acronyms

API	Application programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CD	Continuous Deployment
DB	Database
DNS	Domain Name System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IaC	Infrastructure-as-Code
IDC	International Data Corporation
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IPC	Interprocess Communication
ISP	Internet Service Provider
JS	JavaScript
LTS	Long-Term Support
MSA	Microservice Architecture
NVM	NodeJS Version Manager
OS	Operating System
OAS	OpenAPI Specification
PaaS	Platform as a Service
PID	Process ID
PWA	Progressive WebApp
QA	Quality Assurance
SDLC	Software Development Life Cycle
REST	Representational State Transfer
RSA	Rivest-Shamir-Adleman
SSH	Secure Shell Protocol
TDD	Test-Driven Development
UI	User Interface
UX	User Experience
VCS	Version Control System
VM	Virtual Machines
VPN	Virtual Private Network
VSCode	Visual Studio Code
WSL	Windows Subsystem for Linux
XAMPP	Apache, MariaDB, PHP, Perl
XP	Extreme Programming
YAML	YAML Ain't Markup Language

B Code Listings

```
# PHP version: 8-apache-bullseye, 7-apache-bullseye, and more
ARG VARIANT=7.3-apache-bullseye
FROM php:${VARIANT}

# Copy scripts and install tools
COPY install-scripts/*.sh /tmp/install-scripts/
RUN apt-get update && bash /tmp/install-scripts/install.sh \
    && apt-get -y install lynx libmemcached-dev zip \
    zlib1g-dev libicu-dev libxml2-dev libssl-dev \
    htop less curl unzip git iputils-ping unzip cron \
    zlib1g-dev libpng-dev libjpeg-dev zlib1g-dev libzip-dev \
    && docker-php-ext-install bcmath \
    && docker-php-ext-install sockets \
    && docker-php-ext-install pdo pdo_mysql\
    && docker-php-ext-install mysqli \
    && docker-php-ext-install mbstring \
    && docker-php-ext-install simplexml \
    && docker-php-ext-install gd \
    && docker-php-ext-install zip \
    && a2enmod rewrite \
    && usermod -aG www-data ${USERNAME} \
    && sed -i -e "s/Listen 80/Listen 80\\nListen 8080/g" \
    /etc/apache2/ports.conf

# Install xdebug
RUN yes | pecl install xdebug \
    && echo "zend_extension=$(find /usr/local/lib/php/
       extensions/ -name xdebug.so)" > /usr/local/etc/php/conf
       .d/xdebug.ini \
    && echo "xdebug.mode = debug" >> /usr/local/etc/php/conf.d
       /xdebug.ini \
    && echo "xdebug.start_with_request = yes" >> /usr/local/
       etc/php/conf.d/xdebug.ini \
    && echo "xdebug.client_port = 9000" >> /usr/local/etc/php/
       conf.d/xdebug.ini \
    && rm -rf /tmp/pear

# Install composer
RUN curl -sSL https://getcomposer.org/installer | php \
    && chmod +x composer.phar \
    && mv composer.phar /usr/local/bin/composer
```

```
WORKDIR /var/www/html
COPY . /var/www/html
RUN composer install --no-interaction --no-plugins \
    --no-scripts --prefer-dist && a2enmod rewrite

# Fix permission and set config files
COPY contrib/docker-apache.conf /etc/apache2/sites-enabled/
RUN ln -s /usr/local/bin/php /usr/sbin/php \
    && chown -R www-data:www-data /var/www/html/var/log \
    && chown -R www-data:www-data /var/www/html/var/cache \
    && chown -R www-data:www-data /var/www/html/var/openvpn \
    && mv docker-entryscript.sh /docker-entryscript.sh \
    && chmod +x /docker-entryscript.sh \
    && mv application/configs/application.docker.php \
    application/configs/application.php
```

Listing 10: PHP DevContainer Dockerfile

```
services:
  webapp:
    image: webapp/dev-container
    entrypoint: "/workspace/.devcontainer/entrypoint.sh"
    environment:
      - PORT=3000
      - BASE_URL=http://localhost:3000
      - AUTH_BACKEND_URL=http://localhost:80
      - SYSTEM_BACKEND_URL=http://localhost:8090
    volumes:
      - ./portal-webapp:/workspace
      - portal_node_modules:/workspace/node_modules
      - vscode_extentions:/root/.vscode-server/extensions
    ports:
      - 3000:3000

  system-api:
    image: system-api/dev-container
    entrypoint: "/workspace/.devcontainer/entrypoint.sh"
    environment:
      - PORT=8090
      - MONGO_DB_HOST=mongo
      - MONGO_DB_USER=root
      - MONGO_DB_PW=none
      - MONGO_DB_NAME=local-log-db
```

```
26        - SQL_HOST=db
27        - SQL_USER=root
28        - SQL_PASSWORD=yes
29        - SQL_DB_NAME=local-device-db
30      volumes:
31        - ./system-api:/workspace
32        - system_api_node_modules:/workspace/node_modules
33        - vscode_extentions:/root/.vscode-server/extensions
34      ports:
35        - 8090:8090
36
37    mqtt-endpoint:
38    image: mqtt-endpoint/dev-container
39    entrypoint: "/workspace/.devcontainer/entrypoint.sh"
40    environment:
41      - MQTT_CLIENT_ID=mqtt-connector
42      - MQTT_HOST=mosquitto
43      - MQTT_USERNAME=mqtt_user
44      - MQTT_PASSWORD=mqtt_pw
45      - HTTP_PORT=8080
46    volumes:
47      - ./mqtt-endpoint:/workspace
48      - mqtt_endpoint_api_node_modules:/workspace/node_modules
49      - vscode_extentions:/root/.vscode-server/extensions
50    ports:
51      - 8080:8080
52
53    auth-backend:
54      image: auth-backend/dev-container
55      volumes:
56        - ./auth-backend/src:/var/www/html/src
57      ports:
58        - 80:80
59
60  ####### Auxiliary services #######
61    db:
62      image: mysql
63      environment:
64        - MYSQL_ROOT_PASSWORD=yes
65      ports:
66        - 3306:3306
67      volumes:
68        - sql_data:/var/lib/mysql
```

```
69    mongo:
70      image: mongo
71      environment:
72        - MONGO_INITDB_ROOT_USERNAME=root
73        - MONGO_INITDB_ROOT_PASSWORD=none
74      ports:
75        - 27017:27017
76      volumes:
77        - mongo_data:/data/db
78    mosquitto:
79      image: eclipse-mosquitto
80      ports:
81        - 1883:1883
82      volumes:
83          - mosquitto_data:/mosquitto/data
84          - ./manage-reop/config:/mosquitto/config
85
86  ####### Database managment services #######
87    myadmin:
88      image: phpmyadmin
89      environment:
90        - PMA_HOST=db
91        - PMA_USER=root
92        - PMA_PASSWORD=yes
93      ports:
94        - 8888:80
95  volumes:
96    vscode_extentions:
97    webapp_node_modules:
98    system_api_node_modules:
99    sql_data:
100   mongo_data:
101   mosquitto_data:
```

Listing 11: All Symbic Services in one `docker-compose.yml`

C Additional Figures & Tables

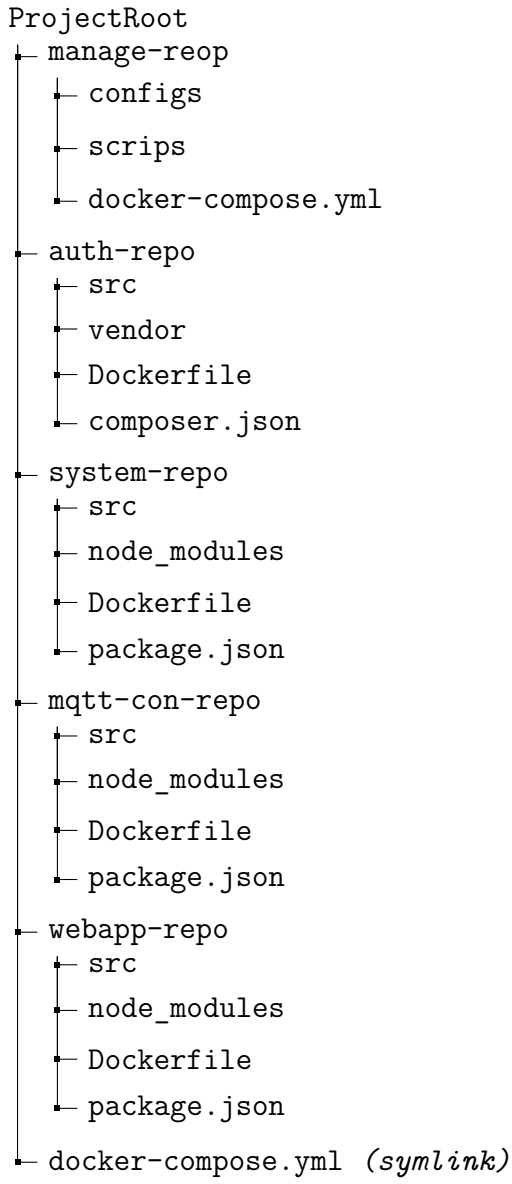

Figure 7: Directory Structure of the IoT-Project

	Orchestration Support	Language Support	Server Hosted/ Local Hosted	Network Functionality	Package Support	Pricing
Codesandbox.io	no	NodeJS	yes/no	HTTP - accessible from the web	restricted for native packages	Free, 30$, 56$
Stackblitz	no	NodeJS	no/yes	WebSockets - accessible from the browser	no support for native packages	Free, 9$, 39$
Gitpod	within containers	any	yes/yes	Full TCP/UDP	any	Free, 9$, 25$, 39$
Codespaces	within containers	any	yes/yes	Full TCP/UDP	any	Free, 4$, 21$
DevContainer	within containers & between containers	any	yes/yes	Full TCP/UDP	any	Free, 5*$, 7*$, 21*$

Table 6: Comparison of Different Development Solutions

List of Figures

List of Tables

Listings

Links to Extensive Code Snippets

- DevContainer Nodejs Image `install.sh`
  https://gist.github.com/hegerdes/310618ffff2af2c9d72033dd9a1a996c.
- DevContainer PHP Image `install.sh`
  https://gist.github.com/hegerdes/9bdacad696d9f8d86acaa0d9666c813a.
- Performance Evaluation Script `run-eval.sh`
  https://gist.github.com/hegerdes/cfbf2b2065eccad725bcb4af6cb0491d.

Declaration of Authorship

I hereby declare that the paper submitted is my own unaided work. I assure that I wrote this paper without using any other means and sources than those specified. As well as the sources used literally or analogously taken from the sources identified as such.

Signature (Henrik Gerdes)

Osnabrück, the November 1, 2021