TECHNISCHE HOCHSCHULE MITTELHESSEN

CAMPUS GIESSEN

MNI
Mathematik, Naturwissenschaften und Informatik

Bachelor Thesis

Functional Package Management and Containerization Technologies in Development Environments and Software Deployment

for the Degree of

Bachelor of Science

submitted to the Department of Mathematics, Natural Sciences, and Computer Science at the Technische Hochschule Mittelhessen (University of Applied Sciences)

by

Clemens Horn

October 1, 2024

Referee: Prof. Dr. Dennis Priefer

Co-Referee: Kevin Linne, M.Sc.

Declaration of the use of Generative AI

In accordance with the recommendation of the German Research Foundation (DFG - Deutsche Forschungsgemeinschaft)[1] and that of the journal Theoretical Computer Science[2] I (the author) herby declare the use of generative AI.

During the preparation of this work I used ChatGPT 4o in order to improve readability and language, only. After using ChatGPT 4o, I reviewed and edited the content as needed and take full responsibility for the content of this thesis.

Declaration of Independence

I hereby declare that I have composed the present work independently and have not used any sources or aids other than those cited, and that all quotations have been clearly indicated.

Gießen, on October 1, 2024 Clemens Horn

1 DFG Formulates Guidelines for Dealing with Generative Models for Text and Image Creation: `https://www.dfg.de/en/news/news-topics/announcements-proposals/2023/info-wissenschaft-23-72`

2 Declaration of generative AI in scientific writing: `https://www.sciencedirect.com/journal/theoretical-computer-science/publish/guide-for-authors`

This thesis explores the comparative use of functional package management and containerization technologies, focusing on their applicability to development environments and software deployment. The motivation behind this research is the increasing need for reproducible, efficient, and scalable software environments in modern development workflows. As organizations scale and applications become more complex, choosing the right tools for managing development and deployment processes becomes critical.

The core research question investigated is how Nix, a functional package manager, and Docker, a containerization platform, compare in terms of efficiency, scalability, and reproducibility in different software development and deployment contexts. The thesis seeks to determine which tool is more suited for specific use cases, such as small-scale, self-hosted environments versus large-scale, high-traffic applications. Through an empirical evaluation of development workflows and deployment models, the research highlights key differences in how each tool handles package isolation, process isolation, scalability, and infrastructure management.

Methodologically, the research was conducted through hands-on experimentation with both Docker and Nix, comparing their performance in setting up reproducible development environments and deploying software. The study also analyzed the use of binary caches and declarative environments in Nix, as well as Docker's container-based isolation and scalability with orchestration tools like Kubernetes.

The findings suggest that Nix is highly effective for setting up reproducible development environments and is particularly beneficial for small-scale deployments where traffic is predictable and resources are limited. Docker, on the other hand, excels in large-scale deployments and high-traffic applications due to its scalability and flexibility in dynamic environments. Despite Docker's advantages in scaling, Nix's use for development environments remains valuable even in larger projects due to its time-saving benefits and reduced friction in managing dependencies.

The implications of this research extend to both practice and academic study. Practitioners can leverage the findings to better choose tools based on the specific needs of their projects, whether prioritizing reproducibility or scalability. From an academic perspective, this thesis contributes to the growing body of research on software deployment and environment management, suggesting areas where further integration between functional package management and containerization technologies could be explored.

Contents

1 Introduction and Motivation

In the rapidly evolving field of software development, managing development environments and ensuring consistent deployment has become increasingly challenging [Conb]. This thesis explores functional package management and containerization, two key concepts that address these challenges.

Functional package management creates reproducible and consistent software environments through a declarative approach. It ensures that software components and their dependencies are precisely defined, reducing conflicts and errors. This method is crucial for projects where reproducing the exact development environment is essential for debugging, testing, and deployment [Mal24].

Containerization, on the other hand, packages applications and their dependencies into portable containers. This ensures that applications run consistently across various environments, from development to production. Containers provide isolated environments for applications, simplifying deployment and enhancing scalability [Pah15].

The relevance of this thesis lies in examining these foundational concepts, which are vital in modern software development. As organizations adopt DevOps [Dev24] practices and CI/CD [CIC24] pipelines, the need for reliable tools to manage environments and deployments grows [Conb]. Understanding functional package management and containerization helps developers and organizations make informed decisions about their tools and methodologies.

The motivation for this thesis comes from the increasing complexity of software projects and the challenges of managing development environments. Issues like "dependency hell" [Dep24] and environment discrepancies can slow development, introduce bugs, and complicate deployment. This thesis aims to provide solutions to these problems by exploring functional package management and containerization.

Reproducibility is a significant driver for this study. In complex projects, reliably recreating the development environment is crucial. Functional package management ensures reproducibility, while containerization guarantees consistent and portable deployment, maintaining stability across different stages of the software lifecycle [Mal24].

This thesis offers a comprehensive understanding of functional package management and containerization, filling a gap in existing literature. The insights gained will help developers, system administrators, and organizations choose the best tools for their needs. As the software development landscape evolves, understanding these concepts will be essential for managing environments and deployments effectively.

In conclusion, this thesis addresses the challenges of modern software development by exploring functional package management and containerization. By enhancing the efficiency, reliability, and scalability of development and deployment processes, this work contributes valuable insights to both academic and industry audiences.

1.1 Objectives

The primary goal of this thesis is to provide a comprehensive comparison between functional package management and containerization technologies as development and deployment tools. The thesis seeks to analyze these two paradigms through empirical and practical exploration, focusing on key metrics such as build times, deployment speeds, package sizes, and the management of developer environments.

At the heart of this research is the central question: **How do functional package management and containerization technologies compare as development and deployment tools, and what recommendations can be derived for enterprises that use these technologies?** The purpose of this research is to offer insights that help developers and organizations make informed decisions about which technology to adopt depending on their specific needs.

In order to thoroughly explore this question, the thesis pursues several objectives. First, it aims to examine the philosophical underpinnings of both functional package management and containerization. Functional package management emphasizes immutability, reproducibility, and declarative configurations. Containerization technologies, on the other hand, center on process isolation, portability, and flexibility. These core principles are vital to understanding the broader context in which each paradigm excels. It is essential to investigate how these philosophies impact the setup and management of development environments, software reproducibility, and the overall software lifecycle.

The second objective is to conduct a technical analysis of the implementation details that differentiate these technologies. By evaluating their declarative configurations and lifecycle management, the thesis explores how these approaches provide strengths and expose limitations, particularly in areas like environment isolation, package dependencies, and configuration complexity. These technical distinctions reveal the potential benefits and constraints of using either functional package management or containerization technologies in different development contexts.

The third objective is to apply these technologies to a real-world project to provide empirical evidence of their effectiveness. A web server written in Rust serves as the test bed for demonstrating how functional package management and containerization behave under practical conditions, with a focus on dependency management, environment isolation, build reproducibility, and deployment. This case study illustrates how each technology can handle typical tasks faced by development teams and highlights specific scenarios where one approach may be more suitable than the other.

Finally, the thesis evaluates performance metrics such as build times, package sizes, and deployment scalability. Build times are crucial because they directly affect developer productivity and the continuous integration/continuous deployment (CI/CD) pipeline. By comparing containerization's layered caching approach with the derivation-based build system of functional package management, the research examines how quickly each system can produce artifacts under various conditions. Package size, another key metric, has significant implications for storage and network bandwidth, particularly in production environments. The streamlined derivations of functional package management, compared to the larger images produced by containerization technologies, highlight the potential advantages of functional package management in terms of efficiency. Furthermore, scalability and deployment speeds are assessed to determine how well these systems perform when scaled to meet increasing production demands.

Through these objectives, this thesis aims to offer concrete, data-driven recommendations to organizations and developers, helping them select the most suitable tools for their development and deployment processes. The goal is to enhance the efficiency, reliability, and scalability of software workflows while providing a clear comparison of functional package management and containerization technologies.

1.2 Approach

To systematically address the research question, this thesis adopts a comprehensive approach that integrates theoretical analysis, empirical evaluation, and practical exploration. The methodology begins with an extensive review of existing literature on functional package management and containerization technologies. This literature review encompasses academic papers, technical documentation, blog posts, and industry reports. The purpose of the review is to gain a thorough understanding of the theoretical foundations and practical uses of both functional package management and containerization technologies, which are widely used in modern software engineering. The literature review provides a foundation by examining the history, core principles, and technical developments behind these paradigms, highlighting how they have evolved to solve specific challenges in development and deployment. By synthesizing insights from a variety of sources, the literature review lays the groundwork for further theoretical and empirical analysis.

The theoretical analysis builds upon the literature by exploring the design philosophies and technical structures of both functional package management and containerization technologies. In this phase, core elements such as package management, dependency resolution, environment isolation, and reproducibility are closely examined. Functional package management emphasizes declarative configurations to build environments, ensuring that each build is consistent and reproducible. In contrast, containerization technologies encapsulate entire environments—including applications, dependencies, and the operating system—within isolated units. Particular attention is given to how these approaches handle immutability, caching, and dependency conflicts, as these are crucial factors in determining the reliability and reproducibility of development environments. This theoretical analysis helps establish a conceptual framework for understanding the unique strengths and limitations of both functional package management and containerization before moving into empirical testing.

The empirical evaluation is central to this research and involves setting up a series of controlled experiments to measure the performance, scalability, and efficiency of functional package management and containerization technologies in real-world development and deployment scenarios. These experiments are designed to evaluate how each technology performs under typical conditions faced by development teams. A key focus is placed on measuring build times, deployment speeds, and the size of deployment artifacts. For instance, build times are assessed under cold builds, warm builds, and fully cached builds to reflect various stages of a developer's workflow. Both technologies are tested using a web server project written in Rust, and the experiments are automated through GitHub Actions, ensuring consistent conditions for each test run. The layered caching model of containerization is compared to the derivation-based system of functional package management, with metrics such as time saved from caching, overall build speed, and the impact of caching on warm and cold builds closely monitored. Additionally, the final package sizes produced by both approaches are analyzed to determine which technology generates more efficient artifacts for deployment in terms of storage and bandwidth. By carefully controlling the experimental conditions, the empirical evaluation generates reliable, reproducible data that supports the theoretical analysis conducted earlier.

Finally, the findings from both the theoretical analysis and empirical evaluation are synthesized in a discussion that critically addresses the research question. This discussion not only compares the performance of functional package management and containerization technologies but also explores the practical implications for development workflows, deployment processes, and overall project management. The strengths, weaknesses, opportunities, and risks of each approach are analyzed, offering valuable insights into when it is most appropriate to adopt one technology over the other. For instance, while containerization excels in scalable, high-traffic environments, functional package management's reproducibility and lightweight approach to package management may be more suited for smaller teams or projects focused on deterministic builds. The discussion integrates these findings to provide practical recommendations for developers, teams, and organizations aiming to optimize their software development and deployment workflows.

This multifaceted approach aims to provide a thorough and nuanced exploration of the research question. By combining a detailed theoretical framework with rigorous empirical testing, the thesis offers a comprehensive comparison of functional package management and containerization technologies, highlighting their similarities, differences, and broader implications for modern software engineering practices.

1.3 Scope

This thesis provides a comprehensive analysis of functional package management and containerization technologies as they are applied throughout the software development pipeline, focusing on the key stages of development, testing, and deployment. The research encompasses both development and deployment tools, examining how these technologies integrate into existing workflows and enhance productivity and reproducibility. The scope of this study is broad, aiming to evaluate how functional package management and containerization contribute to software engineering practices, from initial coding to final production deployments. In the development phase, the thesis evaluates the role of functional package management and containerization in supporting developers during the coding process. A key aspect of this analysis is understanding how each approach manages dependencies and provides reproducible development environments. Functional package management offers developers a declarative way to define their environment, ensuring that every team member works within the same configuration, with immutability and reproducibility at the core of the approach. On the other hand, containerization technologies encapsulate the entire development environment within a container, providing process isolation and flexibility by packaging applications along with their dependencies and operating systems. The thesis also investigates how well these technologies integrate with popular development tools and frameworks, such as integrated development environments (IDEs). The ability of these technologies to seamlessly interface with widely used tools is crucial for their adoption in real-world projects, where productivity and workflow efficiency are paramount.

Moving into the deployment phase, the thesis explores how functional package management and containerization are employed to package, deploy, and manage applications in production environments. This includes assessing their ability to create reproducible deployment artifacts, which are vital for ensuring consistency between development, staging, and production environments. The research examines how containerization technologies, using layered images and orchestration capabilities, enable efficient scaling and management of applications at scale, particularly with orchestration tools like Kubernetes. Similarly, functional package management is analyzed for its ability to create reproducible deployment packages through declarative configuration, ensuring consistency across all stages of the deployment pipeline. The thesis also addresses the scalability of both technologies, exploring how well they handle growing demands in production environments, and how they automate deployment tasks to reduce operational overhead and improve efficiency.

Throughout the analysis, several factors that influence the selection and application of functional package management and containerization in various software development contexts are considered. These factors include the specific requirements of the project, the expertise of the development team, and the constraints imposed by the organization. For instance, a team with more experience in container orchestration may prefer a containerization solution, while an organization prioritizing maximum reproducibility and environmental consistency might favor functional package management. By assessing the strengths and limitations of both technologies across different stages of the software development pipeline, the thesis aims to provide insights into their suitability for various use cases, ranging from small, self-contained projects to large-scale applications requiring complex orchestration and scalability.

It is important to note that the scope of this thesis does not extend to monitoring and observability aspects of software systems. While these components are crucial in modern production environments, they fall outside the focus of this study. Instead, the thesis maintains its emphasis on the development and deployment phases, focusing on how these tools impact software engineering practices in these areas. The goal is to provide a focused examination of how these tools shape the reproducibility, scalability, and efficiency of software environments without delving into post-deployment monitoring strategies.

By focusing on how these technologies manage environments, automate deployments, and scale applications, the thesis aims to deliver valuable insights for developers, organizations, and researchers seeking to leverage these technologies in their workflows.

2 Theoretical Foundations

This chapter delves into the theoretical foundations of two critical technologies in modern software development: functional package management and containerization. By examining the principles, architecture, and examples of these technologies, we aim to build a comprehensive understanding that will inform the comparative analysis and practical applications discussed in subsequent chapters.

2.1 Overview of Functional Package Management

This section introduces the fundamental concepts and scope of functional package management, laying the groundwork for a detailed exploration of its core concepts, architectural design, and practical examples.

2.1.1 Definition and Scope of Functional Package Management

Functional package management is a method for managing software packages by treating them as immutable entities, ensuring reproducibility, and using declarative configuration methods [Cou13, Section 2.1]. Unlike traditional package management systems, which allow in-place upgrades and manual dependency handling, functional package management guarantees that each package and its dependencies remain unchanged once created [Cou13, Section 2.2]. This immutability allows for the creation of reproducible environments, ensuring consistent configurations across different systems and times, thus mitigating the "it works on my machine" problem [Rah22, Section 2, Page 5].

The scope of functional package management includes immutability, reproducibility, and declarative configuration. Immutability means that any updates or changes to a package result in a new version, leaving the original version intact [Dol04, Page 84]. Reproducibility involves capturing the exact state of the build environment, including all dependencies and their versions, allowing for precise recreation of environments [Cou13, Section 2.1]. Declarative configuration involves describing the desired state of the system, with the package manager handling the necessary adjustments to achieve that state [Cou13, Section 2.1].

A core aspect of functional package management is *dependency closure*, which ensures that all dependencies required for a package are completely defined and isolated from the global system. This prevents dependency conflicts and ensures that different versions of a package can coexist [Dol06, Chapter 3.3]. Functional package management systems like Nix also guarantee strict reproducibility by hashing all inputs, including dependencies and system configurations, used in the build process. This ensures that package builds remain identical over time [Cou13, Section 2.2].

An important concept in functional package management is *dependency closure*, which ensures that different versions of dependencies can coexist without conflicts. This technique prevents dependency hell and ensures that packages are built in isolation, so different applications or parts of the system can operate independently [Dol06, Chapter 3.3]. By ensuring that packages include their full set of dependencies, Nix guarantees that the build process can reproduce identical results across different systems and times [Cou13, Section 2.1].

Functional package management excels in scenarios where *long-term stability* is crucial. Industries like scientific computing or regulated environments benefit from its guarantee that any system or environment configuration can be rebuilt, even years later, without deviation from the original setup [Rah22, Section 2, Page 5]. By maintaining immutable package versions and using exact versions for every dependency, these systems ensure that builds are not subject to drift over time. This stands in contrast to traditional systems, where updates to libraries and dependencies can introduce uncertainty into future builds [Dol06, Chapter 5.2].

Furthermore, functional package management reduces system pollution. Developers can use tools like `nix develop` or `nix-shell` to create isolated development environments without modifying the global system. This approach allows quick reinitialization by symlinking the required packages, significantly reducing initialization time compared to container-based solutions [Nixa, Nixb]. As a result, developers can rapidly switch between different project environments while ensuring that the global environment remains clean and unaffected.

2.1.2 Examples of Functional Package Management Systems

Several systems exemplify the principles and benefits of functional package management. These systems provide robust tools for managing software packages and their dependencies in a reproducible and declarative manner, making them suitable for complex development environments and deployment scenarios.

One prominent example is Nix, a functional package manager that employs a purely functional approach to package management. Nix ensures that each build is isolated and reproducible by using Nix expressions, which describe how packages are built. This approach guarantees that every dependency is explicitly defined and accounted for, allowing for precise replication of environments. Nix's unique features, such as the Nix store, where packages are stored in a way that ensures immutability, and its garbage collection mechanism, which automatically removes unused packages, make it a powerful tool for managing software in a consistent and reliable manner [Cou13, Section 2.2].

Another notable example is GNU Guix, which is built on top of the Nix package manager [Ack] and extends its functionality with a focus on free software principles. Guix uses Scheme, a dialect of Lisp [Lis24], for its configuration language, providing a high degree of flexibility and power in defining package configurations and system setups. Like Nix, Guix ensures reproducibility and immutability, making it an excellent choice for environments where these properties are critical.

Both Nix and Guix exemplify the core principles of functional package management. They provide the tools and frameworks necessary to create and maintain reproducible, immutable, and declaratively configured environments, offering significant benefits for both development and deployment of software.

2.1.3 Architecture and Key Principles of Functional Package Management

Functional package management systems are built on several key architectural components and principles that work together to ensure immutability, reproducibility, and predictability in software management. By understanding both the architecture and the underlying principles, we gain insight into how these systems achieve their goals.

At the core of functional package management is the purely functional approach, where operations like installing, updating, or removing a package do not have side effects. Instead of altering the global system state, these operations create a new system state, ensuring predictability and reversibility [Cou13, Section 2.1]. This concept directly supports the architecture by ensuring system consistency even as packages are added or modified.

A central component of the architecture is the Nix store, a centralized location where all package versions and their dependencies are stored. Each package is stored under a unique path based on its hash, ensuring that once a package is built, it cannot be modified, preserving its immutability [Dol06, Chapter 5.2]. This immutability guarantees system stability, as packages cannot be altered once installed. Updates or modifications result in new versions, which are stored alongside older versions, allowing for versioning and rollback functionality [Dol06, Chapter 1.5].

Another critical architectural component is the build system, which constructs packages from source code. The build system captures the complete environment required to build a package, including all dependencies such as compiler versions and libraries. This ensures reproducibility, meaning the same inputs will always produce the same outputs [Dol06, Section 2.6]. Reproducibility is crucial in ensuring that software environments can be exactly recreated across different systems, an essential feature for both development and production.

Functional package management also relies on declarative configuration, which simplifies system management by allowing users to specify the desired state of the system instead of providing step-by-step instructions on how to achieve it [Cou13, Section 2.2]. This declarative approach is facilitated by the configuration language, a domain-specific language or format used to declare package dependencies and system configurations. In Nix, this is achieved through Nix expressions, which provide a flexible and powerful way to define system configurations [Dol06, Chapter 2.2].

Dependency closure is another key concept underpinning functional package management. In traditional systems, dependencies can conflict, causing issues across applications. In functional systems, each package includes all its dependencies, ensuring independent functionality without conflicts [Dol06, Chapter 3.3]. This architectural decision ensures that different versions of the same dependency can coexist without interfering with each other, a crucial feature in complex environments.

Functional package management systems also employ garbage collection to manage disk space. Since packages are immutable, old versions remain in the system unless explicitly removed. The system automatically identifies and removes unused packages, ensuring that only the necessary packages are retained [Cou13, Section 2.2].

The combination of these architectural components and principles provides a robust and efficient framework for managing software. The isolation of packages, made possible through dependency closure and immutability, ensures that different packages do not interfere with each other. This is particularly important in multi-application environments, where different applications may require different versions of the same dependency [Dol06, Chapter 2.1].

By focusing on immutability, reproducibility, and declarative configuration, functional package management systems offer significant advantages over traditional methods. The architecture is designed to support these principles, providing predictability, reliability, and ease of use in managing complex software environments.

Versioning and rollback capabilities provide a safety net by allowing users to revert to previous system states, further enhancing system stability [Dol06, Chapter 1.5]. Each change in the system is transactional, ensuring that updates are consistent and can be reversed if necessary.

Together, these key principles and architectural components form the foundation of functional package management, ensuring that software environments are isolated, reproducible, and maintainable across different use cases.

2.2 Overview of Containerization Technologies

This section introduces the fundamental concepts, principles, and architectural components of containerization. Containerization technologies have revolutionized the way software is developed, shipped, and deployed. By encapsulating applications and their dependencies into isolated containers, these technologies ensure that software runs consistently across various environments [Mer14, Under the Hood].

2.2.1 Definition and Scope of Containerization

Containerization is a lightweight form of virtualization [Mer14, Containers vs. Other Types of Virtualization] that packages an application along with its dependencies, libraries, and configuration files into a single container. Unlike traditional virtual machines, containers share the host operating system's kernel but run as isolated processes [Mer14, Containers vs. Other Types of Virtualization]. This isolation provides a consistent runtime environment, ensuring that applications behave the same way regardless of where they are deployed. The scope of containerization encompasses several key aspects. First, isolation is a fundamental feature where containers provide process and resource isolation through mechanisms like namespaces [Mer14, Under the Hood] and control groups (cgroups) [Mer14, Under the Hood] in the Linux kernel. This ensures that containers do not interfere with each other, even though they share the same operating system kernel [Ope24].

Portability is another crucial aspect [Sof24]. Containers can run on any system that supports the container runtime, making them highly portable across different environments, such as development, testing, and production. Additionally, containerization offers significant efficiency advantages. Containers are more resource-efficient compared to traditional virtual machines because they do not require a full OS instance for each application [Mer14, Under the Hood]. Instead, they share the host system's kernel, which leads to lower overhead and faster start-up times.

Containerization has become an essential technology in modern software development [Mer14, Docker: a Little Background], enabling the development and deployment of applications in a consistent, repeatable manner.

2.2.2 Theoretical Concepts of Containerization

Containerization technologies, such as Docker and Podman, have revolutionized the software deployment landscape by offering lightweight, efficient methods for isolating applications and their dependencies. The core theoretical concepts underlying containerization are rooted in OS-level isolation and resource control mechanisms that provide environments capable of running applications consistently across various infrastructure environments. A central concept in containerization is process isolation, which is achieved through the use of namespaces. Namespaces ensure that applications running within a container are isolated from processes on the host or in other containers. By providing isolated views of the system, namespaces prevent containers from interfering with each other or accessing resources not explicitly shared [Cgr24]. This isolation allows containers to simulate separate operating systems while still sharing the host's kernel. Another critical concept is resource control, which is primarily implemented through control groups (cgroups). Cgroups allow the system to limit, prioritize, and isolate resource usage (e.g., CPU, memory, and network bandwidth) for processes within a container. This ensures that containers do not exceed their allocated resources, thus maintaining system stability even when multiple containers run concurrently [Mer14, Under the Hood]. Containers also rely on image layering, a technique that allows reusable and immutable layers of a container image to be shared across different containers. This reduces redundancy and improves efficiency, as containers only need to store and transfer the unique layers for each application [Mer14, Chapter 2.4]. Layers improve the overall speed of container deployment and resource usage, enabling faster initialization and scaling in dynamic environments. While containers offer significant benefits in terms of isolation and resource control, the concept of reproducibility remains a challenge. Unlike functional package management systems, containers rely on external base images (e.g., Ubuntu or Alpine), which can evolve over time, making it difficult to guarantee that a container built today will be identical in the future unless explicitly managed through version pinning or using image digests [Ino]. Addressing this issue requires additional effort in Docker-based deployments, where base images and package versions must be manually pinned. Containerization technologies are particularly effective in environments where efficiency and scalability are crucial, such as cloud-native applications and microservices architectures. Their ability to provide lightweight, isolated environments with minimal overhead makes them ideal for continuous integration and continuous delivery (CI/CD) pipelines and other dynamic infrastructures [Cgr24].

2.2.3 Examples of Containerization Systems

Several containerization systems exemplify the principles and benefits of this technology, providing robust tools for developing, shipping, and running applications in a consistent and isolated manner.

Docker [Doc22] is one of the most widely used container platforms. It offers a comprehensive suite of tools for building, sharing, and running containers. Docker simplifies the process of creating container images with Dockerfiles and provides a powerful runtime for managing containers [Doc00b]. Its popularity has led to a vast ecosystem of pre-built images available on Docker Hub, making it easy to find and use a wide variety of software [Doc].

Another notable example is Podman [Pod24], a daemonless container engine developed by Red Hat [Red]. Podman provides a similar user experience to Docker but does not require a background daemon, enhancing security and flexibility. It supports rootless containers, allowing users to run containers without elevated privileges, which improves security by reducing the attack surface [Pri21, 4.1].

These examples illustrate the diverse range of containerization systems available, each with its unique features and strengths. Whether it's Docker's ease of use or Podman's security features, these systems demonstrate the versatility and effectiveness of containerization in modern software development and deployment.

2.2.4 Architecture and Key Concepts

Containerization systems are built on several key architectural components and principles that work together to provide isolated, portable, and efficient environments for applications. Understanding these components and their underlying mechanisms highlights how containerization achieves its goals.

At the core of containerization is the concept of container images [Sys24]. A container image is a lightweight, self-contained software package that includes everything needed to run an application: code, runtime, libraries, and environment settings. These images are built from declarative scripts, such as Dockerfiles, and use Docker's layering system to improve efficiency [Und00]. Each instruction in the Dockerfile creates a new layer, and Docker employs caching mechanisms to reuse unchanged layers during builds, significantly reducing image build times [Cac00].

Once built, these images are immutable, ensuring consistent and reproducible environments across various systems. When executed, images are instantiated as containers by a container runtime [Cona]. The runtime is responsible for managing the container lifecycle—creation, execution, and termination. Common container runtimes include the Docker Engine [Doc00a], containerd [Conc], and CRI-O [Cri24]. These runtimes leverage key features of the underlying operating system, particularly namespaces and control groups (cgroups), to provide isolation and resource control.

Namespaces are a Linux kernel feature that isolates different resources at the kernel level. By using different types of namespaces, such as process ID (PID) [Pro24], network, and file system namespaces, containers have their own isolated views of these system resources. For example, the PID namespace ensures that each container has its own process tree, separate from others, while the network namespace gives each container its own network stack [Lin24]. Cgroups are another fundamental Linux kernel feature that ensures efficient resource allocation for containers. Cgroups limit the amount of CPU [Cen24], memory, disk I/O [Inp24], and other resources a container can use, preventing any container from monopolizing system resources. This fine-grained control helps maintain system stability and balance performance across containers [Cgr24]. Linux Containers (LXC) represent an earlier container technology that also relies on namespaces and cgroups. Linux containers run a full Linux distribution within the container but require more manual configuration than Docker containers. Docker containers, by contrast, emphasize ease of use and portability by packaging applications and their dependencies in a standardized image format, simplifying deployment across different environments [Lin]. Another critical component is container networking. Containers must communicate with each other and with external systems. Containerization technologies provide several networking models, such as bridge networks for inter-container communication on the same host, and overlay networks for distributed applications across multiple hosts. Host networking allows containers to share the host's network stack directly [Net00]. Storage is also vital in containerization. While containers are designed to be ephemeral, many applications require persistent data. Containerization technologies offer solutions such as volume mounts for persistent data storage, as well as networked or cloud-based storage systems to ensure data integrity and persistence across container restarts and migrations [Per00].

By combining these architectural elements—container images, runtimes, namespaces, cgroups, networking, and storage—containerization systems create a robust platform for running applications in an isolated, portable, and efficient manner. The isolation provided by namespaces and cgroups ensures that applications can run securely without interfering with each other, while the immutability and layering of container images enhance portability and efficiency. Together, these technologies have made containerization a cornerstone of modern software development and deployment.

3 Practical Application

This chapter presents the practical application of functional package management and containerization technologies through the development and deployment of a stateless web server written in the Rust programming language [Rus]. The primary objective is to demonstrate the effectiveness, challenges, and benefits of using functional package management and containerization technologies in a real-world project. While Nix and Docker are the specific tools selected for this implementation, the emphasis is on the general principles and methodologies of functional package management and containerization technologies, rather than the tools themselves.

The web server in question serves both static and dynamic content. Although the server's functionality is relatively straightforward, it is well-suited for illustrating modern practices in environment management and deployment workflows. The server fetches data from an API and returns web content in HTML, utilizing HTMX for dynamic interaction [Htm] and Tailwind CSS for styling [Tai20]. This project aims to highlight the integration of functional package management and containerization technologies, focusing on how they manage dependencies, ensure reproducible builds, and facilitate the deployment of the application in an isolated environment. The complete source code is publicly available on GitHub [Hor24], allowing for replication of the setup.

The practical importance of this project lies not in the complexity of the web server itself, but in the techniques used to manage the server's development and deployment environments. One of the primary challenges addressed in this project is the management of specific compile-time dependencies, such as OpenSSL, which are required for building the server. These dependencies provide an ideal scenario for demonstrating how functional package management handles complex build environments while ensuring reproducibility. Additionally, this project enables a comparison of how functional package management and containerization technologies contribute to modern development workflows, with a focus on key aspects such as ease of use, build times, environment isolation, and reproducibility.

This chapter is structured progressively, beginning with detailed instructions on setting up the development environments using both functional package management and containerization technologies. This includes installing and configuring Nix for functional package management and Docker for containerization, with specific attention to how these tools are used on various platforms, such as Linux or Windows. Following the installation process, the chapter covers how to integrate these technologies into IDEs like Visual Studio Code, which improves productivity by allowing the developer to manage dependencies, run builds, and isolate environments directly from the IDE. Next, the chapter demonstrates how each technology was utilized to build and run the web server using best practices. Nix was employed for managing dependencies and ensuring reproducibility in development environments, while Docker was used to package the application into containers for deployment. Code snippets are provided to show how the technologies were configured, and best practices are highlighted to ensure that the development and deployment processes are efficient and reliable. The chapter concludes with a section on CI/CD pipelines, configured using GitHub Actions. Separate workflows were created for functional package management and containerization technologies, allowing for a comparison of their performance in automated testing and deployment environments. Each workflow is explained in detail, showing how both approaches can be integrated into modern CI/CD pipelines.

This chapter serves as the foundation for the analysis presented in the next chapter, which will delve deeper into the comparative evaluation of functional package management and containerization technologies. Here, the focus remains on the practical steps taken, while the analysis of their respective strengths, weaknesses, and impacts on development workflows will follow.

3.1 Setup

Setting up a reproducible development environment is crucial for ensuring consistency and reliability in software development, especially when working across various platforms and with diverse team members. In this section, we focus on two specific tools: Nix and Docker, which serve as examples of functional package management and containerization technologies, respectively. Both tools play a vital role in modern development workflows, ensuring that development environments can be recreated accurately, no matter where they are deployed.

The following command installs Nix as a multi-user system on Unix-based systems such as Linux and macOS.

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Listing 3.1: Installing Nix on Linux and macOS

After installation, Nix must be integrated into the user's shell environment to make its commands available. This is done by sourcing the Nix configuration file, which is automatically generated during installation. This step ensures that the ‘nix‘ command and its associated functionalities are accessible within the current shell session.

```
source /nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh
```

Listing 3.2: Configuring Shell for Nix

For Windows users, Nix is not natively supported but can be run using the Windows Subsystem for Linux (WSL). WSL provides a Unix-like environment on Windows systems, allowing users to install and run Linux applications natively. The same installation procedure used on Linux applies within the WSL environment, making it possible for Windows users to leverage the functional package management capabilities of Nix.

Once installed and configured, Nix allows users to define their entire development environment in a single configuration file (e.g., ‘default.nix‘). This file specifies the exact versions of all required dependencies, ensuring that the environment can be reproduced across different machines. The ability to roll back environments to previous states also helps mitigate issues that might arise during software updates.

To install Docker on Linux, users can utilize their system’s package manager. For Debian-based systems, the following commands will install Docker and set up permissions so that users can run Docker without needing superuser privileges:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo usermod -aG docker $USER
newgrp docker
```

Listing 3.3: Installing Docker on Linux

For Windows users, Docker Desktop is available, which uses WSL2 as the backend to run containers. After installing Docker Desktop, users can interact with Docker via its GUI or command-line tools. Docker Desktop also handles system-level configurations for networking and storage, making it easier to manage containers on Windows systems.

In summary, the setup of development environments using Nix and Docker ensures that both package-level and runtime dependencies can be managed in a reproducible and isolated manner. These tools form the foundation of reproducible development workflows, ensuring that development environments can be easily recreated across different systems and stages of software development.

3.2 Development Environment Configuration

In software development, creating a reproducible and isolated development environment is essential for ensuring consistency across different systems and developers. This section discusses how to utilize two key technologies for setting up such environments: functional package management and containerization. The goal is to create development environments that are isolated from the host system, ensuring consistency in the development process regardless of the developer's operating system or machine configuration.

The following subsections will discuss each approach in detail, starting with functional package management and its role in setting up development environments.

3.2.1 Functional Package Management for Development Environments

In a functional package management system, such as Nix, developers can define their entire development environment, including compilers, libraries, tools, and other dependencies, in a single configuration file. This file describes the precise version of every component needed for development and testing.

Below is an example of how this can be achieved using a configuration file. While the following implementation is specific to Nix, the concepts are general and can be applied to other functional package management systems. The file defines inputs, which include external resources like package repositories and overlays for specific languages or tools.

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
    flake-utils.url = "github:numtide/flake-utils";
    nix-filter.url = "github:numtide/nix-filter";
    rust-overlay = {
      url = "github:oxalica/rust-overlay";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };
```

Listing 3.4: Input section for functional package management

In this example, the inputs define the external sources required to build the environment. The main idea is that the package repository (here represented by nixpkgs) and overlays (such as rust-overlay) provide packages in a purely functional manner, ensuring that different versions of packages can coexist without conflicts. The configuration guarantees that the exact same packages will be used by all developers working on the project.

The outputs section defines what is produced by the functional package management system, including development shells. A development shell is an isolated environment that includes all the tools and dependencies needed for development. By defining such environments declaratively, functional package managers ensure that all developers have access to the same setup.

```
outputs = inputs:
  with inputs;
    flake-utils.lib.eachDefaultSystem (
      system: let
        inherit (nixpkgs) lib;
        overlays = [(import rust-overlay)];
        filter = nix-filter.lib;
        pkgs = import nixpkgs { inherit system lib overlays; };
        bin = pkgs.pkgsBuildHost.rust-bin;
        rustToolchain = bin.fromRustupToolchainFile ./rust-toolchain.toml;
```

Listing 3.5: Outputs section for functional package management

The outputs are built in a way that respects the functional programming paradigm. This means that each output is a pure function of its inputs, and any change in the input will result in a predictable change in the output. For example, a change in the rust-overlay version will result in a new environment that reflects that specific version of Rust, without affecting other environments that rely on a different version.

Functional package management systems also allow the definition of development shells, which are isolated environments that provide all the tools and dependencies needed for development. These shells are constructed based on the inputs and outputs defined in the configuration file.

```
          devShells = {
            default = mkShell {
              buildInputs = [pkg-config];
              nativeBuildInputs = [
                extendedRustToolchain
                rust-analyzer
                openssl
                proto
                moon
              ];
              RUST_SRC_PATH =
              "${rust.packages.stable.rustPlatform.rustLibSrc}";
              RUST_BACKTRACE = 1;
              shellHook = ''echo "Entering the development shell..."'';
            };
          };
```

Listing 3.6: Development shell for functional package management

The development shell isolates the environment, ensuring that any tools or libraries installed in the shell do not interfere with the global system. Furthermore, each shell is defined declaratively, meaning that it can be shared among developers, ensuring that everyone works in the same environment.

In addition to specifying the dependencies, functional package management allows the definition of environment variables. These variables can be defined inside the development shell configuration and will be automatically set whenever the shell is entered. In the example above, the `RUST_SRC_PATH` variable is set to point to the Rust standard library source, and the `RUST_BACKTRACE` variable is enabled to assist with debugging by providing detailed backtrace information.

Another powerful feature provided by functional package management systems like Nix is the ability to define *shell hooks*. A shell hook is a script that runs as soon as the development shell is entered. This can be used to run initialization scripts, set up additional configuration, or prepare the environment for specific tasks.

In this example, a shell hook is defined, which prints a message and sets a custom environment variable. Shell hooks can be a powerful tool for automating setup tasks and ensuring the environment is properly initialized for development.

Functional package management systems also support binary caching, allowing developers to reuse pre-built binaries across different environments and machines. This significantly speeds up the process of setting up a development environment, as pre-built binaries can be fetched from a cache instead of being built locally.

```
nixConfig = {
  extra-substituters = [
    "https://nix-community.cachix.org"
    "https://clemenscodes.cachix.org"
  ];
  extra-trusted-public-keys = [
    "nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRCYg3Fs="
    "clemenscodes.cachix.org-1:yEwW1YgttL2xdsyfFDz/vv8zZRhRGMeDQsKKmtV1N18="
  ];
};
```

Listing 3.7: Cachix configuration for functional package management

By incorporating binary caching into the functional package management process, developers can avoid the time-consuming task of building every package from source, while still ensuring that the environment remains reproducible and isolated.

In summary, functional package management provides a powerful and flexible way to define development environments. By isolating dependencies and ensuring that environments are fully reproducible, developers can avoid conflicts and inconsistencies across different machines, ensuring that the development process remains smooth and predictable.

In the next subsection, we will discuss how containerization, particularly with Docker Devcontainers, can be used to achieve similar isolation and reproducibility.

3.2.2 Containerization for Development Environments

Containerization is an important concept in modern software development, enabling the creation of isolated, reproducible environments. The idea behind containerization is to package not only the application code but also its dependencies, runtime, and system libraries into a single image. This ensures that the environment will behave consistently, regardless of where it is deployed. Docker is one of the most widely used technologies for implementing containerization. Docker containers encapsulate everything needed to run an application, isolating it from the host system and other running applications.

In the context of development environments, containerization is particularly useful because it guarantees that developers work with the same tools, configurations, and dependencies, eliminating issues that may arise from differences between local setups. The concept of immutability is central to containerization: containers are built from immutable images, ensuring that the environment does not change unexpectedly once it is created.

The Visual Studio Code Dev Containers extension allows developers to use a container as a fully-featured development environment. By isolating the development environment inside a container, it ensures that the tools, libraries, and configurations remain consistent across different systems. This is particularly useful in collaborative projects or when working with complex dependencies. A `devcontainer.json` file is used to define how Visual Studio Code accesses or creates a development container with the necessary toolchain and runtime stack. The container can be used to run the application, compile libraries, or provide separate runtimes needed for interacting with the codebase.

Workspace files are either mounted from the local file system into the container or cloned directly into it. This ensures that developers can seamlessly interact with the containerized environment while retaining access to the entire workspace from the host system. Extensions are installed and executed inside the container, providing full access to the tools, platform, and file system inside the container.

As a result, developers can switch between development environments by simply connecting to a different container, without having to worry about reconfiguring tools or dependencies locally.

The diagram below illustrates the interaction between the local operating system, Visual Studio Code, and the containerized environment:

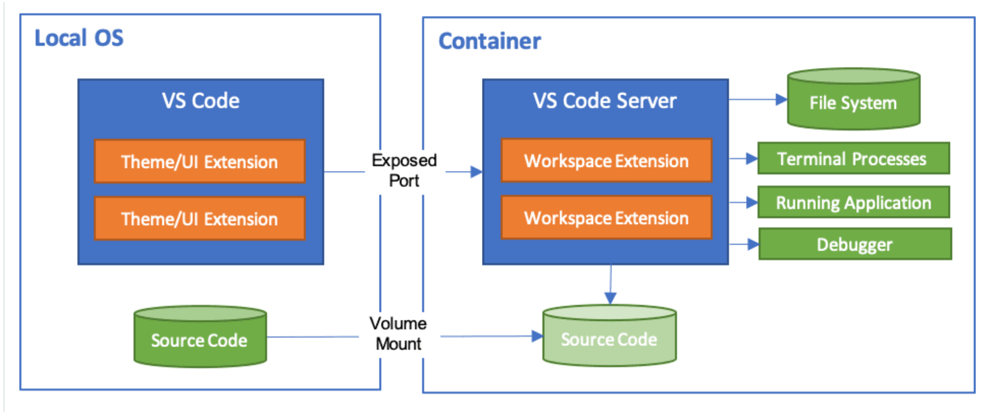

Figure 3.1: Devcontainer Architecture: VS Code and Docker Integration [Dev]

In this architecture, Visual Studio Code, running on the local machine, provides the interface for the developer to interact with the container. The local machine runs UI-related extensions such as themes, while the container hosts the VS Code server and manages all the runtime tasks. Inside the container, developers have access to the file system, terminal processes, application runtimes, and debugging capabilities. The workspace files, including the source code, are mounted from the local system into the container using a volume mount, allowing seamless interaction with the code from within the container. Exposed ports allow services running inside the container, such as web servers, to be accessed from the local machine.

This setup ensures a local-quality development experience while providing all the benefits of containerized isolation. Developers can perform tasks such as code navigation, IntelliSense, and debugging without worrying about conflicts with local environments. Visual Studio Code can run in two primary modes with Dev Containers: as a full-time development environment inside the container, or by attaching to a pre-existing running container for inspection or debugging purposes.

By leveraging Dev Containers, developers can ensure a consistent, isolated development workflow while benefiting from the reproducibility and portability that Docker containers offer. This integration streamlines the process of setting up a development environment, making it easier to manage complex dependencies and configurations in a unified, containerized workspace.

Devcontainer Setup for Development Environments

The Devcontainer setup used for this application highlights the power of containerization technologies in creating a consistent and isolated development environment. The setup consists of three key files: `devcontainer.json`, `Dockerfile`, and `.vscode/extensions.json`. Together, these files configure the containerized development environment, providing developers with all the necessary tools and dependencies without affecting the host system. By using a container, the development environment remains reproducible across different machines, ensuring consistency throughout the project lifecycle. Although this example uses Visual Studio Code Dev Containers, the underlying principles of containerization—such as environment isolation and dependency management—remain relevant across different tools and technologies.

The `devcontainer.json` file is the cornerstone of this setup. It defines how the development container is built and managed, specifying important parameters such as the Dockerfile location, port forwarding, and workspace mounting. The following is the configuration used for the Rust project:

```
{
  "name": "Rust Development Container",
  "build": {
    "dockerfile": "Dockerfile",
    "context": ".."
  },
  "forwardPorts": [
    8000
  ],
  "mounts": [
    "source=${localWorkspaceFolder},target=/app,type=bind"
  ]
}
```

Listing 3.8: devcontainer.json Configuration

This file defines several key aspects of the containerized environment. The `dockerfile` attribute specifies the Dockerfile used to build the container, while the `context` is set to the parent directory. This allows Docker to access all the files needed to construct the environment. The `forwardPorts` section ensures that port 8000 from within the container is accessible on the host system, facilitating local testing of web applications. Finally, the `mounts` section binds the local workspace folder to the `/app` directory within the container, ensuring that changes made on the host machine are immediately reflected inside the container. In this setup, the local workspace directory, represented by the `$localWorkspaceFolder` variable, is bound to the `/app` directory inside the container

The `Dockerfile` is used to create the containerized environment. In this case, the Dockerfile is designed specifically for development purposes, optimizing for flexibility and ease of use rather than production deployment. It defines the base environment, installs necessary tools, and configures additional settings that are critical for Rust development. Below is the Dockerfile used in this project:

```
FROM rust:1.80.1-slim-bullseye

ENV DEBIAN_FRONTEND=noninteractive
ENV SHELL=/bin/bash
ENV PATH="/root/.proto/bin:$PATH"

RUN apt-get update && \
  apt-get install -y --no-install-recommends \
  git=1:2.30.2-1* \
  gzip=1.10-4* \
  unzip=6.0-26* \
  xz-utils=5.2.5-2.1* \
  curl=7.74.0-1.3* \
  pkg-config=0.29.2-1* \
  openssl=1.1.1* \
  libssl-dev=1.1.1* \
  musl-tools=1.2.2-1* \
  make=4.3-4.1* \
  && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*

SHELL ["/bin/bash", "-o", "pipefail", "-c"]

RUN curl -fsSL https://moonrepo.dev/install/proto.sh | \
    bash -s -- 0.40.4 --yes && \
    proto plugin add moon \
    "https://raw.githubusercontent.com/moonrepo/moon/master/proto-plugin.
        toml" && \
    proto install moon

WORKDIR /app

COPY .moon .moon
COPY .prototools .prototools
COPY dockerManifest.json dockerManifest.json
COPY moon.yml moon.yml

RUN moon docker setup && \
    cargo install cargo-watch --locked
```

Listing 3.9: Development Dockerfile for Rust Project

The Dockerfile begins by using the official Rust image `rust:1.80.1-slim-bullseye`, which ensures that the Rust toolchain is available within the container. Key environment variables are defined to streamline the package installation process and configure the shell. A series of packages are installed using `apt-get`, including Git, OpenSSL, and `pkg-config`, which are necessary for building and testing Rust applications.

Next, the `moonrepo` tool is installed via the Proto installer, a tool used to manage project workflows and tasks. This enables the environment to manage project tasks efficiently. The Dockerfile also sets the working directory to `/app`, where the local workspace is mounted. Various configuration files, such as `moon.yml` and `dockerManifest.json`, are copied into the container. Lastly, the `moon` tool is configured, and `cargo-watch` is installed to monitor file changes, providing a productive development experience with real-time feedback.

The `.vscode/extensions.json` file specifies the recommended editor extensions to install in Visual Studio Code. This ensures that all developers have the required tools within the IDE to interact with the containerized environment effectively. Below is the configuration used for this project:

```
{
    "recommendations": [
        "rust-lang.rust-analyzer",
        "ms-azuretools.vscode-docker",
        "ms-vscode-remote.remote-containers",
        "moonrepo.moon-console"
    ]
}
```

Listing 3.10: Recommended VS Code Extensions

This file recommends several key extensions for Rust development and for Docker management, the `remote-containers` extension is necessary to enable Visual Studio Code to interact with the container. The `moon-console` extension integrates with the Moonrepo tool, allowing developers to manage project workflows directly within the editor.

Together, these files create a complete, containerized development environment that ensures consistency across different machines. The environment is isolated from the host system, yet remains fully integrated with the developer's workflow through Visual Studio Code. The setup provides a robust solution for managing dependencies, building and testing Rust applications, and handling project tasks within a containerized environment.

3.3 Reproducible Builds

In this section, we explore how reproducible builds are achieved using functional package management through a Nix-based approach, focusing on concepts such as dependency isolation, deterministic builds, and environment specification. Reproducible builds are an essential concept in software development that ensures anyone can build the exact same binary from the same source code, across different machines and environments. The idea is to eliminate any potential for variance between builds, such as those caused by differing system configurations, dependencies, or build tools. In a reproducible build, the input always results in the same output. This ensures that developers can trust the integrity and consistency of their builds, regardless of where or when the build process takes place.

3.3.1 Reproducible Builds with Functional Package Management

Functional package management systems like Nix introduce immutability and isolation into the package management process. Every environment is built from a set of declarative rules, ensuring that the same rules always yield the same result. The key to reproducible builds in functional package management lies in the declarative definition of dependencies and build processes, which eliminates the impact of external factors like system state or environment configuration. The build process in Nix relies on expressions, such as those written in the `default.nix` file, which define how the software should be built and what dependencies are required. These expressions are purely functional, meaning that their output is determined entirely by the input. Let us explore how the Nix expression for building a Rust application ensures reproducibility. The `default.nix` file begins by importing and filtering the source files, which is crucial for maintaining a well-defined input set. The source code and other relevant files are explicitly listed, ensuring that only these files are used during the build process. This isolates the build from any extraneous files in the directory:

```nix
src = filter {
  root = ./.;
  include = [
    ./src
    ./styles
    ./templates
    ./assets
    ./Cargo.lock
    ./Cargo.toml
  ];
};
```

Listing 3.11: Source filtering in Nix

This approach ensures that only the specified files—such as the `src` directory, stylesheets, templates, and the `Cargo.toml` and `Cargo.lock` files—are included in the build. By filtering the input, Nix guarantees that no unintended files or external dependencies are introduced into the build process, contributing to the reproducibility of the final output. It also ensures that a new derivation is only built if the source files for the application changed.

Another key element of reproducible builds is dependency management. In Rust projects, the `Cargo.lock` file locks the exact versions of dependencies, ensuring that the same versions are used every time the project is built. Nix leverages this concept by importing and hashing the dependencies from the `Cargo.lock` file:

```
cargoDeps = rustPlatform.importCargoLock {
  lockFile = ./Cargo.lock;
};
cargoHash = "sha256-EYTuVD1SSk3q4UWBo+736Mby4nFZWFCim3MS9YBsrLc=";
```

Listing 3.12: Rust dependency management in Nix

Here, the `importCargoLock` function imports the exact dependency versions defined in `Cargo.lock`. Additionally, the `cargoHash` attribute ensures that the same dependencies are used by verifying the integrity of the `Cargo.lock` file. This hashing mechanism ensures that even if the dependencies were to change in the source repository, the build process would fail unless the `Cargo.lock` file remains unchanged, thereby guaranteeing reproducibility.

In functional package management systems, builds are isolated from the host system, and all dependencies must be explicitly defined. This prevents any contamination from system libraries or tools, which can lead to non-reproducible builds. In the Nix expression, the `nativeBuildInputs` and `buildInputs` attributes explicitly define the dependencies needed for building the application:

```
nativeBuildInputs = [pkg-config];
buildInputs = [openssl];
```

Listing 3.13: Defining build dependencies in Nix

By listing the necessary build inputs (such as `pkg-config` and `openssl`), Nix ensures that the build process does not rely on any system-wide installed tools or libraries. Instead, all dependencies are fetched from the Nix package store, where they are versioned and managed immutably. This further strengthens the reproducibility of the build, as the same versions of these dependencies will be used regardless of the machine or environment in which the build is performed.

Once the application is built, a script can be defined to run the application with the necessary environment variables. This is achieved using the `writeShellScriptBin` function, which ensures that the correct paths are set for the application's runtime assets:

```
writeShellScriptBin pname ''
  WEBSERVER_ASSETS=${assets}/assets ${unwrapped}/bin/webserver
''
```

Listing 3.14: Generating the run script in Nix

This script ensures that the `assets` directory and other runtime resources are correctly set up when the application is executed. Since all dependencies, build tools, and runtime configurations are explicitly defined within the Nix expression, the build and execution of the application remain reproducible regardless of the underlying environment.

3.3.2 Reproducible Builds Using Containerization

Containerization technologies, such as Docker, are key to achieving reproducible builds by isolating the entire build environment from the host system. A Dockerfile defines a controlled environment where the exact dependencies, system libraries, and tools are specified, ensuring consistency across different machines and platforms. By following best practices, Docker enables reproducible builds that maintain the integrity of the software, regardless of external factors such as system configuration or host environment changes.

The Dockerfile used in this project adheres to several best practices, including multi-stage builds, dependency version pinning, and build optimization. Each section of the Dockerfile is crafted to ensure that the resulting binary is reproducible and that the build process remains efficient.

The Dockerfile begins by specifying a base image, `rust:1.80.1-slim-bullseye`, which provides a minimal Debian environment with the Rust toolchain pre-installed. Pinning the image version ensures that the same version of Rust is used in every build, eliminating variability that could arise from different versions of the language or runtime. This helps achieve reproducibility since it guarantees that the environment remains the same over time.

```
FROM rust:1.80.1-slim-bullseye AS base

ENV DEBIAN_FRONTEND=noninteractive
ENV SHELL=/bin/bash
ENV PATH="/root/.proto/bin:$PATH"
```

Listing 3.15: Base Image and Environment Setup

```
RUN apt-get update && \
  apt-get install -y --no-install-recommends \
  git=1:2.30.2-1* \
  gzip=1.10-4* \
  unzip=6.0-26* \
  xz-utils=5.2.5-2.1* \
  curl=7.74.0-1.3* \
  pkg-config=0.29.2-1* \
  openssl=1.1.1* \
  libssl-dev=1.1.1* \
  musl-tools=1.2.2-1* \
  make=4.3-4.1* \
  && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*
```

Listing 3.16: Installing Dependencies in Docker

In this section, system dependencies are installed using the `apt-get` package manager. Each package is pinned to a specific version, which ensures that future builds use the exact same versions, preventing inconsistencies caused by upstream changes. The `-no-install-recommends` option is used to avoid installing unnecessary packages, keeping the image as small as possible.

After installation, `apt-get clean` is run to free up disk space, reducing the size of the Docker image. Additionally, the `SHELL` directive with `-o pipefail` ensures that if any command in a pipeline fails, the entire build will fail, which prevents silent failures during the build process.

The Dockerfile also includes the use of multi-stage builds to keep the final image small and efficient. Multi-stage builds allow us to use a larger image with necessary build tools for compiling the application, and then copy only the compiled binary and essential files into a smaller runtime image.

```
FROM base AS build

COPY Cargo.toml Cargo.lock ./
RUN rustup target add x86_64-unknown-linux-musl && \
    cargo build --release --target=x86_64-unknown-linux-musl

FROM alpine:3.20.2 AS start

COPY --from=build /app/webserver /usr/local/bin/webserver
```

Listing 3.17: Multi-Stage Build Setup

The first stage, labeled `build`, compiles the Rust application. The `Cargo.toml` and `Cargo.lock` files are copied into the container to resolve dependencies. By copying these files early, Docker can cache the dependency resolution step, improving build times by ensuring that dependencies are only recompiled if they change. The Rust target is set to `x86_64-unknown-linux-musl`, ensuring that the binary is statically linked, making it more portable across different environments.

The final stage, labeled `start`, uses the minimal `alpine:3.20.2` image. Only the compiled binary and necessary runtime files are copied from the `build` stage into this smaller image. This keeps the final image lightweight by excluding the build tools and unnecessary libraries from the runtime environment. This separation of build and runtime environments is a best practice that results in a secure and efficient Docker image.

To ensure the Rust binary is fully self-contained and does not rely on dynamic libraries, the Dockerfile compiles OpenSSL against the MUSL C library, resulting in a statically linked binary. This approach improves portability, as the binary can be deployed on any system without requiring shared libraries such as OpenSSL.

```
RUN ln -s /usr/include/x86_64-linux-gnu/asm /usr/include/x86_64-linux-musl/
    asm && \
    ln -s /usr/include/asm-generic /usr/include/x86_64-linux-musl/asm-
        generic && \
    ln -s /usr/include/linux /usr/include/x86_64-linux-musl/linux && \
    mkdir /musl && \
    curl -LO https://github.com/openssl/openssl/archive/OpenSSL_1_1_1f.tar.
        gz && \
    tar zxvf OpenSSL_1_1_1f.tar.gz

WORKDIR /openssl/openssl-OpenSSL_1_1_1f/
RUN CC="musl-gcc -fPIE -pie" ./Configure no-shared no-async --prefix=/musl \
    --openssldir=/musl/ssl linux-x86_64 && \
    make depend && \
    make -j"$(nproc)" && \
    make install
```

Listing 3.18: OpenSSL Compilation with MUSL

This process statically links OpenSSL with the Rust binary, enabling the Rust application to handle HTTPS connections without needing shared OpenSSL libraries at runtime. By statically linking critical libraries, the resulting binary becomes fully portable and can be executed in a minimal container, such as the Alpine image, without additional dependencies. Security best practices are also implemented by running the application as a non-root user, which reduces the risk of privilege escalation attacks. In the final stage, a non-root user named `webserver` is created, and the ownership of the binary is transferred to this user.

```
RUN addgroup -g 1000 webserver && \
    adduser -D -s /bin/sh -u 1000 -G webserver webserver && \
    chown webserver:webserver /usr/local/bin/webserver

USER webserver

CMD ["webserver"]
```

Listing 3.19: Running the Application as a Non-Root User

Running the application as a non-root user is a Docker best practice, as it limits the potential impact of any security vulnerabilities within the application. The `CMD` directive ensures that the container will execute the `webserver` binary when it starts.

This Dockerfile follows several best practices to ensure a reproducible, secure, and efficient build of the Rust web server. By using multi-stage builds, pinning dependency versions, and statically linking critical libraries, the Dockerfile guarantees that the build is consistent and portable across different systems. These practices also ensure that the final image is as small as possible, minimizing the attack surface and reducing resource usage. The use of a non-root user and the minimal Alpine image further enhance the security and efficiency of the resulting container, making it well-suited for deployment in production environments.

3.4 Continuous Integration/Continuous Deployment (CI/CD)

Continuous Integration and Continuous Deployment are key practices in modern software engineering. These practices automate the processes of building, testing, and deploying applications, ensuring consistent, reliable workflows. For the application developed in this thesis, GitHub Actions was used to orchestrate the CI/CD pipeline. GitHub Actions is particularly suited for CI/CD pipelines because of its seamless integration with GitHub repositories, especially for projects hosted on GitHub.

One of the main benefits of GitHub Actions is its ability to trigger workflows automatically based on events like `push` or `pull_request`. This automation ensures that new code is built, tested, and deployed immediately, without manual intervention. Moreover, GitHub Actions is free for public repositories, which makes it a cost-effective solution for open-source projects and smaller teams. In this project, GitHub Actions handles the entire CI/CD process without additional infrastructure costs.

GitHub Actions also provides a variety of pre-built actions and plugins that make it easy to set up and manage complex workflows. These actions can handle tasks such as checking out the repository, logging into container registries, setting up Docker environments, and even managing multi-platform builds. In this application, several pre-built actions are used to automate the CI/CD process, reducing the need for custom scripts. This increases both the speed and reliability of the pipeline, since these actions are maintained by a broader community and official providers.

A key feature of GitHub Actions is its built-in caching mechanism, which helps improve performance for containerized applications. For projects using Docker, caching can significantly reduce build times by avoiding the need to rebuild all layers of a Docker image from scratch. Instead, unchanged layers from previous builds are reused. This optimization is especially useful in large projects where Docker images may consist of multiple layers. In the application, Docker layer caching is employed extensively to speed up the CI/CD pipeline, ensuring efficient and rapid builds. The workflow builds and pushes Docker images to the GitHub Container Registry (GHCR). GHCR is tightly integrated with GitHub, offering a secure and convenient place to store Docker images. GitHub Actions can log into GHCR using tokens and permissions, avoiding the need for external credentials. This integration ensures secure image management, reduces friction when working with container registries, and enhances the overall security model by avoiding the need to manage sensitive credentials manually.

The CI/CD pipeline is composed of several steps, each designed to handle specific parts of the process. Below are the key sections of the workflow file, along with explanations of the important actions.

The first step defines the events that trigger the pipeline. In this case, the workflow is triggered when code is pushed to the `main` branch or when a pull request is created or updated.

```yaml
name: Docker GHA Cache
on:
  push:
    branches: [main]
  pull_request:
    types: [opened, synchronize]
```

Listing 3.20: Triggering Events

This setup ensures that any code changes pushed to the `main` branch, or any modifications to an open pull request, will trigger the CI/CD pipeline. This helps maintain continuous feedback and ensures that code is always tested and built consistently.

Concurrency control is also defined to prevent multiple workflows from running simultaneously for the same branch or pull request. This avoids redundant builds and ensures that only the most recent changes are processed.

```yaml
concurrency:
  group: ${{ github.workflow }}-${{ github.event.number || github.ref }}
  cancel-in-progress: true
```

Listing 3.21: Concurrency Settings

Environment variables are used to define key settings like the Docker registry and image name. By using environment variables, the workflow becomes more flexible and easier to maintain.

```yaml
env:
  DOCKER_REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}
```

Listing 3.22: Environment Variables

This setup ensures that the Docker image is built with the correct registry and repository name, avoiding the need to hardcode these values in the workflow.

The next part of the workflow defines the job that will run on an `ubuntu-latest` runner. The repository is checked out using the `actions/checkout@v4` action to ensure that the latest code is available for the build process.

```yaml
jobs:
  docker:
    runs-on: ubuntu-latest
    permissions:
      id-token: write
      contents: read
      packages: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
```

Listing 3.23: Job Setup and Repository Checkout

Once the code has been checked out, the workflow logs into the GitHub Container Registry using the `docker/login-action@v3`. This action allows GitHub Actions to securely authenticate and push Docker images to the registry using `GITHUB_TOKEN`.

```yaml
- name: Log in to the Container registry
  uses: docker/login-action@v3
  with:
    registry: ${{ env.DOCKER_REGISTRY }}
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
```

Listing 3.24: Logging into the GitHub Container Registry

Using the `docker/setup-buildx-action@v3` action, Docker Buildx is set up. Buildx provides advanced features such as multi-platform builds and more efficient caching.

```
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3
```

Listing 3.25: Setting up Docker Buildx

Metadata for the Docker image is extracted using the `docker/metadata-action@v5`. This ensures that the image is properly tagged and labeled for version management.

```
- name: Extract metadata (tags, labels) for Docker
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: ${{ env.DOCKER_REGISTRY }}/${{ env.IMAGE_NAME }}
```

Listing 3.26: Extracting Metadata for Docker Image

Finally, the Docker image is built and pushed to the GitHub Container Registry using the `docker/build-push-action@v6`. Caching is enabled to reuse unchanged layers from previous builds, significantly reducing build times.

```
- name: Build and push Docker image
  id: push
  uses: docker/build-push-action@v6
  with:
    context: .
    push: true
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
    cache-from: type=gha
    cache-to: type=gha,mode=max
```

Listing 3.27: Building and Pushing the Docker Image

This caching mechanism is vital for optimizing the CI/CD pipeline, as it speeds up subsequent builds by avoiding the need to rebuild Docker layers that have not changed. This is particularly useful for larger projects where Docker images consist of multiple layers.

For the Nix-based project components, two different CI pipelines are used: one that leverages GitHub Actions' built-in caching and another that uses Cachix, a service designed to cache Nix build artifacts across different environments. Both pipelines help ensure reproducibility by caching build results and avoiding unnecessary rebuilds.

The first workflow uses the `magic-nix-cache` action to store Nix build artifacts within GitHub Actions' cache. This setup is fully integrated within the GitHub environment, making it easy to use. The job runs on an `ubuntu-latest` runner, and Nix is installed using the `DeterminateSystems/nix-installer-action@main`. The code is checked out, and the build process is initiated using the Nix flake configuration.

```yaml
jobs:
  magic-nix-cache:
    runs-on: ubuntu-latest
    permissions:
      id-token: "write"
      contents: "read"
    steps:
      - uses: actions/checkout@v4
      - uses: DeterminateSystems/nix-installer-action@main
      - uses: DeterminateSystems/magic-nix-cache-action@main
```

Listing 3.28: GitHub Runner and Nix Installation

Cachix is used in the second pipeline to provide caching that can be shared between CI pipelines and local developer environments. This allows developers to reuse build results across environments, improving productivity. The key element in the Nix CI pipeline using Cachix is that it allows caching of Nix flake inputs, development shells, and runtime closures. By pushing these caches to the Cachix service, developers can reuse previously built derivations, improving both CI and local development build times.

```yaml
      - name: Cache flake inputs
        run: |
          nix flake archive --json \
            | jq -r '.path,(.inputs|to_entries[].value.path)' \
            | cachix push ${{ env.NIX_CACHE }}
```

Listing 3.29: Caching Nix Flake Inputs

The ‘nix flake archive‘ command is used to archive Nix flake inputs (dependencies defined in the Nix configuration), and the results are pushed to the Cachix cache to be reused in future builds.

The next step caches the development shell, enabling developers to pull a cached shell environment, ensuring that their local environment is identical to the CI environment.

```yaml
      - name: Cache development shell
        run: |
          nix develop --accept-flake-config --profile \
            ${{ env.NIX_DEV_PROFILE }} -c true
          cachix push ${{ env.NIX_CACHE }} ${{ env.NIX_DEV_PROFILE }}
```

Listing 3.30: Caching the Development Shell

Finally, the runtime closures (outputs of the Nix build) are cached. This ensures that the project can be rebuilt without needing to recompile unchanged parts of the project, saving time.

```
      - name: Cache runtime closures
        run: |
          nix build --accept-flake-config --json \
            | jq -r '.[].outputs | to_entries[].value' \
            | cachix push ${{ env.NIX_CACHE }}
```

Listing 3.31: Caching Runtime Closures

By caching both the flake inputs and the development shell, the Nix CI pipeline significantly reduces build times for both CI and local environments, making it a robust and efficient system for handling functional package management in the project.

This chapter has provided an in-depth exploration of the CI/CD pipeline setup for the application, focusing on two key technologies: containerization using Docker and functional package management through Nix. The workflows designed for the project not only ensure reproducibility and consistency but also streamline the development process by automating key tasks such as building, testing, and deploying the application. These workflows, powered by GitHub Actions, emphasize best practices for modern software engineering, such as dependency version pinning, caching, and secure image management.

The use of Docker and Nix in the CI/CD pipeline demonstrates how containerization and functional package management can be employed to maintain a stable development environment, reducing the risk of inconsistencies between development, testing, and production environments. Caching mechanisms, both for Docker images and Nix derivations, play a crucial role in speeding up the build process, making the entire CI/CD pipeline more efficient.

The workflow designs provided here are not specific to any particular software stack or industry but represent best practices that can be adopted across the board to ensure consistency, security, and performance in development operations.

It is important to note that everything described in this chapter serves as both a guideline and a proof of concept for how companies can implement reproducible, efficient CI/CD pipelines utilizing functional package management or containerization technologies in their own environments. The tools and concepts discussed here are widely applicable and can be adapted to fit various types of projects and software architectures.

4 Analysis

This chapter presents an in-depth analysis of the performance metrics collected during experiments involving Docker and Nix. The goal of this analysis is to compare key metrics such as build times and package sizes, which are critical in evaluating the efficiency of each technology for creating reproducible environments and deploying software. The experiments were designed to be reproducible and transparent, with all tests automated via GitHub Actions in a continuous integration (CI) environment. By automating the process, consistent conditions were maintained throughout, allowing for accurate comparisons between Docker and Nix. The results for these tests are publicly accessible on GitHub.

4.1 Build Times

Build times represent the total duration from the start of a build process to its completion. For both Docker and Nix, the build times were captured during the CI pipeline. Build time is a critical metric as it directly impacts the efficiency of the development process and the feedback loop in continuous integration and continuous deployment (CI/CD) environments.

For Docker, the build times were collected by running the CI process and exporting an artifact generated during the build. This artifact, named `.dockerbuild`, contains detailed information about the build process, including timestamps. The CI process initiated a Docker build for the application without any prior caching, ensuring that the entire build process, including dependency resolution and layer creation, was performed. Once the build completed, the CI system generated the `.dockerbuild` artifact, which contains the build details such as the start and end times for each build layer. This artifact was then imported into Docker Desktop on Windows, where the build times for each layer were inspected. The total build time reported corresponds to the duration indicated in Docker Desktop after importing the build artifact from the CI system. This method ensures that the Docker build times reflect real-world builds where developers would build the application from scratch or with varying levels of cache utilization.

For Nix, the build times were measured by capturing the duration of the specific CI step responsible for building the application. The Nix build process in the CI pipeline was configured to produce a reproducible environment using Nix derivations, ensuring that the dependency tree and all necessary packages were included. The CI pipeline executed a Nix build of the application, fetching the required dependencies and building the software according to the provided Nix expression. The time taken for the Nix build step was measured in seconds, from the start of the build to its successful completion. The final build time for Nix is based on the total duration of this CI step, which directly reflects the time required to build the application.

Both Docker and Nix build times were captured under three different scenarios: cold builds, warm builds, and fully cached builds. Cold builds represent a build performed without any prior caching, where all dependencies must be resolved and built from scratch. Warm builds involve the use of caching mechanisms, such as Docker's layer caching system or reuse of artifacts from the Nix store, and include source code changes that invalidate some parts of the cache but not all. This means that some dependencies or components must be rebuilt, but parts of the build can still take advantage of previously cached results. Fully cached builds, in contrast, are builds where no new source code changes have been introduced, and in theory, everything should be cached. In these scenarios, both Docker and Nix skip all build steps that can be reused from the cache, resulting in the fastest possible build times.

The following table summarizes the build times for Docker and Nix across these three build types:

Table 4.1: Build Time Comparison between Docker and Nix (in seconds)

Build Type	**Docker**	**Nix GHA Cache**	**Nix Cachix**
Cold Build	244	168	144
Warm Build	44	152	144
Cached Build	9	14	3

The build time metric is crucial in evaluating the efficiency of Docker and Nix, particularly when looking at how each system handles different types of builds: cold, warm, and fully cached builds. The performance differences observed in Table 4.1 highlight how each technology handles caching and dependency resolution.

For cold builds, Nix outperforms Docker due to its lightweight and modular approach to handling dependencies. In Docker, a cold build involves pulling a base image, which typically includes an entire operating system. This process adds significant overhead, as large base images must be downloaded and unpacked before the actual build process can begin. Nix, on the other hand, does not rely on full base images. Instead, Nix can directly pull the required build-time dependencies from pre-existing caches (such as

Cachix) or, if necessary, build dependencies from source in a fully reproducible manner. This modularity and avoidance of OS-level image downloads allow Nix to complete cold builds faster than Docker. This advantage is especially noticeable when the project has many external dependencies that can be retrieved from binary caches, significantly reducing the need for source compilation.

However, the performance of Nix in warm builds shows almost no improvement over cold builds. This is because Nix derivations are built in isolated, immutable environments, which ensures that each build is reproducible and free from side effects. While this immutability is one of the core strengths of Nix, it also means that Nix cannot reuse intermediate build states in the same way Docker can. During the Nix warm build, even if some components are cached, the isolation enforced by the derivation system prevents significant reuse of build outputs, leading to similar build times as observed in cold builds. This immutability guarantees reproducibility but at the cost of speed in scenarios where only minor changes are made to the source code.

In contrast, Docker excels in warm builds, as it allows for optimization through selective caching and mutation of the build process. By leveraging Docker's layer-based caching system, it is possible to separate different stages of the build into distinct layers, each of which can be cached independently. For instance, in the Rust project used for these tests, I was able to optimize the warm build by mutating the build process. Specifically, I replaced the `main.rs` file with a dummy file during an early build layer, caching all dependencies and external crates in that layer. Then, in a subsequent layer, I inserted the actual source code, ensuring that only the source code itself needed to be compiled during rebuilds, while the dependencies remained cached. This approach allows Docker to cache the heaviest parts of the build (dependencies and libraries) while minimizing the amount of work required during warm builds, where only the application code changes. This level of optimization is not possible with Nix, as derivations are immutable and cannot be modified after being built, which is why Nix warm builds show no significant speed advantage compared to cold builds.

When comparing fully cached builds, both Docker and Nix show very fast build times, with Nix being slightly faster overall. Fully cached builds represent a scenario where no new source code changes have been introduced, meaning that the entire build can be reused from previous runs. In this case, the build process simply retrieves all cached results without performing any additional compilation or dependency resolution. Nix's minimal overhead in fetching cached derivations, particularly from services like Cachix, gives it a slight edge in speed over Docker, which must still handle its layer-based caching system. However, the difference is minimal since both systems are designed to avoid redundant work in fully cached builds.

These build time results demonstrate the strengths and weaknesses of each technology in different scenarios. Nix's approach to cold builds is highly efficient, but its immutability limits performance improvements in warm builds. Docker, while slower for cold builds due to its reliance on large base images, excels in warm builds through flexible caching and build optimization. Fully cached builds, which minimize overhead in both systems, showcase the efficiency of caching mechanisms when no changes are introduced.

4.2 Package Sizes

The size of the final deployment artifact is an important factor when considering the efficiency and practicality of using Docker or Nix for software deployment. Smaller packages generally lead to faster installations, reduced storage requirements, and lower bandwidth usage, which can be critical in environments where resources are constrained.

For Docker, the package size was measured by examining the size of the Docker image after it was built and inspected in Docker Desktop. After building the Docker image in the CI pipeline, the image was exported and analyzed locally in Docker Desktop to obtain the final size. This method ensures that the reported size reflects the total disk space consumed by the Docker image, which includes the base operating system, application code, and all associated dependencies.

In contrast, the size of the Nix package was determined by examining the size of the resulting Nix derivation, which was stored in Cachix, a binary cache service. This size reflects only the necessary runtime components of the application and does not include any build-time dependencies, as those can be garbage collected by the Nix system. By checking the cache size in Cachix, I was able to measure the actual size of the derivation after it had been built and uploaded. Importantly, the size of the derivation represents only the essential components required to run the application, without any extraneous system layers or unused dependencies.

Table 4.2 compares the sizes of the Docker image and the Nix derivation:

Table 4.2: Package Size Comparison between Docker and Nix (in MB)

Artifact Type	Docker Image	Nix Derivation
Size	31.82	2.78

As illustrated in Table 4.2, the Nix derivation is significantly smaller than the Docker image. This difference in size is due to the fundamental architectural differences between Docker and Nix. Docker images must include the entire operating system, even when utilizing optimized, lightweight base images such as those built on Alpine Linux. Despite these optimizations, there remains a considerable overhead associated with container technology because the entire environment, including all libraries and tools required by the OS, must be bundled within the image. This results in a larger package size.

In contrast, Nix derivations are inherently smaller because Nix does not bundle an entire operating system or redundant components within its package. Instead, Nix focuses on providing only the necessary runtime dependencies required to run the application. Since Nix derivations are built in an isolated, reproducible environment, any build-time dependencies are discarded and can be garbage collected, ensuring that the final package contains only the minimal set of dependencies needed to execute the software. This lean approach to package management allows Nix to produce much smaller artifacts compared to Docker, where the inclusion of a full OS is unavoidable in containerized environments.

Additionally, the smaller package size of Nix derivations translates into faster deployment times, as fewer bytes need to be transferred when installing or updating the application. In scenarios where bandwidth is limited, or when applications are deployed across multiple machines, the reduced size of Nix derivations offers significant advantages. Faster installation times also contribute to a more responsive and efficient deployment process, particularly in CI/CD pipelines where minimizing downtime and resource usage is crucial.

In summary, while the Docker image was optimized by using lightweight Alpine base images, it still requires additional layers to include the operating system and system libraries, leading to a larger overall size. Nix, by avoiding these redundancies and focusing only on the essential runtime dependencies, produces significantly smaller artifacts that are both faster to install and more space-efficient.

4.3 Developer Environments and Process Management

Both Docker and Nix are commonly used to create isolated, reproducible environments that simplify the management of dependencies and tools in software development. However, the way each system handles environment configuration, workflow integration, and compatibility with different operating systems differs significantly.

Docker uses containers to encapsulate the development environment, including the operating system, dependencies, and tools. This ensures that the environment inside the container closely resembles production systems, allowing developers to create an image that contains the exact tools and dependencies needed for the project. Developers access the environment by running the container, which is isolated from the host system. As a result, any tools, shell configurations, or personal customizations that developers may have in their native environment do not automatically apply within the container. Developers need to manually install and configure the necessary tools inside the container. This can involve additional steps such as connecting to the container and setting up the development environment from scratch.

When using Docker on Windows, a full Linux VM must run in the background to enable containerization, since Docker relies on the Linux kernel. This VM introduces additional overhead, as developers on Windows must manage not only the container itself but also the underlying virtual machine. This approach can affect resource usage and performance, especially when multiple containers are in use. File system synchronization between the host machine and the Linux container can also be slow, with Windows unable to automatically notify the container of file changes. This often requires manual container refreshes, slowing down the iteration process.

Nix, by contrast, provides a declarative and reproducible approach to managing development environments through `nix-shell`. A Nix shell is defined by a `shell.nix` or `default.nix` file that lists the exact dependencies and tools required for the project. When the developer enters the Nix shell, the environment is set up in the developer's existing shell session without running a separate container or virtual machine. This allows developers to use their personal shell configurations, such as aliases and custom profiles, while working in the Nix-provided environment.

A key feature of Nix is its ability to pull pre-built packages from binary caches such as **Cachix**. This allows developers to set up fully functional development environments without having to build dependencies from source. Companies can host and manage their own Nix binary caches, providing employees with ready-to-use, pre-built environments. This ensures that developers, even if they have never worked on the project before, can quickly install all necessary dependencies without rebuilding them, significantly reducing the friction typically involved in setting up new environments. In addition, environments can be delivered on a per-branch basis, allowing developers to switch between branches and automatically load the appropriate environment for each branch without additional setup. This is especially useful in projects with varying dependencies across branches or when specific development tools are required for different stages of the project.

Managing multiple services becomes critical as applications grow in complexity, especially when databases, message brokers, or other auxiliary services are involved. Docker typically uses **Docker Compose** to orchestrate these services in development environments, allowing developers to run multiple containers for services concurrently. This approach ensures that each service runs in its own isolated container, simulating a production-like environment while keeping processes separated from each other.

In the Nix ecosystem, a similar goal can be achieved without containers using a tool called **services-flake**. It provides a declarative way to define and orchestrate services within a project, replacing Docker Compose by running services natively on the host system. Built on top of **process-compose**, **services-flake** allows multiple services to be run concurrently and isolated from one another, without the overhead of containers or virtual machines. It integrates directly with the project's `flake.nix` file, ensuring that services are managed in a reproducible and composable way, similar to how Nix handles package management.

It also supports running multiple instances of services (e.g., multiple databases) and provides a lightweight, platform-agnostic solution that works across macOS, Linux, and other environments where Nix is available. The services are run natively on the host system, and data is stored in a project-specific directory. Additionally, **services-flake** provides a terminal-based user interface for monitoring and managing the services, similar to how Docker Compose allows developers to manage and control containers.

By utilizing **services-flake**, developers can orchestrate complex development environments that include multiple services, such as databases, caches, and application components, all while avoiding the overhead and complexity associated with container technology. The native approach taken by **services-flake** ensures that services remain lightweight and easily manageable, with all configurations handled declaratively through Nix expressions.

In contrast to Docker's approach, which focuses on process isolation within containers, Nix primarily focuses on package isolation. However, with tools like **services-flake**, process isolation in Nix is also achievable, allowing developers to manage multiple services in a reproducible, container-less manner, using the same declarative configuration language that Nix is known for.

4.4 Software Deployments

This section presents the observed results from evaluating Docker and Nix in the context of software deployment, focusing on scalability, serverless capabilities, and infrastructure requirements.

Docker provides a method for deploying software in containerized environments. Docker images package applications with their dependencies and operating system, allowing them to be deployed across platforms that support Docker. In serverless deployments, Docker integrates with orchestration tools like Kubernetes, which manages the scaling of containers. Kubernetes can spin up additional containers when traffic increases and shut them down when traffic decreases. The deployment results indicate that

Docker's container-based packaging supports portability, with Docker images able to run independently of the host system. Docker's integration with orchestration tools like Kubernetes allows for automated scaling of containers based on traffic levels, and the containers can be rapidly started and stopped.

Nix takes a different approach by focusing on package management and reproducibility. Nix packages require deployment on systems with the Nix package manager installed, such as servers running NixOS. This approach ensures that deployments are reproducible and isolated at the package level. However, scaling Nix deployments involves provisioning servers with the Nix package manager, which is more manual compared to Docker's automated container orchestration. The results show that Nix deployments offer reliable reproducibility across environments. Nix deployments also require a fully functional server, and scaling must be achieved by adding additional servers manually or using infrastructure automation tools compatible with Nix.

In terms of scalability, Docker's containerization model, combined with Kubernetes, demonstrates rapid horizontal scaling capabilities. Docker containers can be deployed quickly, making the system suitable for environments with varying traffic levels. In contrast, Nix deployments are more static, with scaling requiring server provisioning and management. There is no built-in automated scaling equivalent to the container-based scaling observed with Docker. These results highlight differences in how Docker and Nix handle scalability and deployment automation.

5 Conclusion

This chapter provides a comprehensive reflection on the findings of this thesis, which explored the comparative use of functional package management and containerization technologies in development environments and software deployment. By systematically analyzing and evaluating the performance of Nix and Docker, this thesis has identified key strengths and limitations of each technology, offering practical recommendations for their usage in modern software development workflows. Additionally, the chapter presents suggestions for future work and discusses the broader implications of these findings for both practice and research.

5.1 Summary of Findings

The primary research question of this thesis aimed to compare Nix and Docker as tools for managing development environments and software deployment. Through detailed analysis and empirical evaluation, it became evident that both tools excel in different areas and cater to distinct use cases.

In development environments, Nix provides a highly effective solution for setting up and maintaining development environments, offering reproducibility and dependency management that is unmatched by Docker. By using `nix-shell`, developers can instantly create isolated development environments without the need for containers or virtual machines. This reduces the overhead associated with container-based environments, especially on non-Linux systems where Docker requires a virtual machine to emulate Linux. Nix also benefits from binary caches, allowing developers to pull pre-built packages and environments from shared caches, significantly speeding up setup times. This makes Nix ideal for both individual developers and teams, as the time saved by eliminating lengthy build processes leads to greater productivity.

The findings suggest that Nix should be recommended for any development environment, regardless of scale. For smaller teams or startups, the ability to provide quick, consistent, and reproducible environments is particularly beneficial, reducing the friction typically associated with onboarding new developers or working across different machines. The cost-efficiency and ease of use further make Nix an attractive option for companies that do not require the overhead of managing containers for development.

In the context of software deployment, the choice between Nix and Docker depends largely on the scale and traffic demands of the application. Nix is well-suited for smaller-scale, self-hosted solutions where traffic is relatively stable and does not fluctuate significantly. For startups or small companies that may not expect major traffic spikes, deploying applications on rented virtual private servers using Nix can be a highly cost-effective solution. The ability to manage the entire environment declaratively with Nix, combined with its package isolation, ensures a stable and reproducible deployment process. However, Nix's requirement for a fully functioning server with the Nix package manager can introduce complexity for larger-scale applications that require more flexibility and scalability.

For high-traffic applications and large-scale companies, Docker's containerization capabilities offer clear advantages. Docker excels in dynamic, scalable environments, where applications must be able to handle fluctuating traffic demands. Docker's integration with orchestration platforms like Kubernetes allows for automated scaling, making it the preferred choice for microservices architectures and serverless deployments. Docker containers are lightweight, portable, and can be easily managed across different infrastructure platforms, giving organizations the ability to scale services up or down with minimal effort.

Despite Docker's superiority in handling large-scale deployments, the use of Nix for development in such environments should not be overlooked. Even in large-scale organizations, utilizing Nix for development environments can provide substantial benefits. Nix's ability to maintain reproducible development environments, combined with its efficient caching mechanisms, ensures that developers spend less time setting up environments and more time coding. By caching developer shells and reusing dependencies, Nix significantly reduces friction in the development process, particularly when iterating on complex codebases. Therefore, even for large-scale projects that use Docker for deployment, Nix remains a highly valuable tool for development, offering time-saving benefits that improve overall productivity.

5.2 Recommendations

Based on the findings of this research, several practical recommendations can be made for developers and organizations when selecting between Docker and Nix for different use cases. For small companies and startups, particularly those that are self-hosting or working with rented virtual private servers (VPS), adopting Nix for both development and deployment is highly recommended. Nix provides a lightweight and reproducible approach to managing environments, reducing the operational overhead often associated with containerization technologies. Since Nix declaratively manages package dependencies and builds reproducible environments, it is particularly suitable for teams that do not need dynamic scaling but still require consistency across development and production environments. This approach works well for environments where traffic is stable, and scaling requirements are minimal. Nix allows smaller organizations to focus on efficient use of resources and maintain a streamlined development process without dealing with the complexity of container orchestration or dynamic scaling.

For companies working in environments with more dynamic traffic or high scalability requirements, Docker remains the most practical option for handling deployments. Docker's containerization technology, especially when paired with orchestration tools like Kubernetes, allows applications to scale seamlessly based on demand. Containers are lightweight, portable, and can be spun up and shut down quickly, providing resource efficiency and reducing downtime in high-traffic applications. The flexibility Docker offers in terms of scaling services independently is crucial for organizations that operate at larger scales or employ microservice architectures. The ability to efficiently manage application scaling through tools like Kubernetes is particularly important for large-scale companies that need to dynamically allocate resources to different services as traffic changes.

Despite Docker's advantages in production deployment for larger applications, Nix remains a powerful tool for managing development environments, even in large organizations. By leveraging Nix to manage development environments, teams can benefit from fast, reproducible setups that eliminate the friction often associated with configuring environments manually or dealing with dependency conflicts. Nix's ability to cache environments and reuse previously built dependencies means that developers spend less time setting up environments and more time coding. This is particularly valuable in larger teams where consistent environments are required across many developers. The combination of Nix for development and Docker for deployment provides a hybrid solution where Nix guarantees reproducibility during development while Docker offers scalability and flexibility in production.

5.3 Future Work

The research conducted in this thesis opens up several avenues for further investigation and development. One promising area of future research could focus on optimizing the integration between Nix and container-based deployment models. While Docker is highly effective at handling process isolation and scalability, Nix's functional package management could be combined with containerization to create hybrid solutions that leverage the strengths of both tools. For instance, a future direction might explore how Nix environments can be deployed within containers to ensure both reproducibility and scalability. This would allow organizations to benefit from the rapid environment setup Nix offers, while also taking advantage of Docker's scalability for production deployments.

Another area worth exploring is improving the scalability of Nix in high-traffic environments. Currently, while Nix excels at package management and reproducibility, it lacks the dynamic scaling capabilities offered by Docker and Kubernetes. Further research could investigate how Nix deployments might be adapted to scale more effectively, potentially through the use of orchestration tools specifically tailored to Nix's package management model. Additionally, developing tools or extensions that make Nix more suitable for dynamic traffic environments could make it a viable option for larger-scale deployments that require the flexibility to adjust resource usage in real-time.

Another avenue for future work could explore improving developer experience and onboarding with Nix in larger organizations. Although tools like **services-flake** show promise, further work could be done to simplify process orchestration and make it easier for larger teams to adopt Nix. By enhancing the usability of Nix in more complex environments, it could become an even stronger competitor to Docker for managing both development and production environments.

5.4 Implications for Practice and Research

The findings of this thesis have significant implications for both practical applications and academic research. From a practical perspective, organizations now have clearer guidelines for when to use Nix or Docker based on their project requirements. Small companies and startups that focus on maintaining consistency across environments without needing to scale dynamically will find Nix highly effective for both development and deployment. By contrast, organizations that require rapid scaling and process isolation will benefit more from Docker's containerization model, especially when deploying high-traffic applications using orchestration tools like Kubernetes.

The research also highlights the potential value of using Nix for development, even in environments where Docker is preferred for production. The use of Nix in development environments can reduce the overhead and friction associated with setting up and maintaining development environments, particularly in large organizations. This hybrid approach, where Nix is used for development and Docker for deployment, ensures that teams can maintain reproducibility and efficiency across the software lifecycle, balancing the strengths of both technologies.

For academic research, this thesis contributes to the growing body of knowledge on software deployment and package management. It provides a comparative analysis of functional package management and containerization, identifying where each approach excels and where improvements can be made. Future research could build on these findings by exploring how Nix and Docker can be integrated more effectively or how the scalability challenges of Nix could be addressed. There is also an opportunity to study how the principles of functional package management could be applied to other areas of software development and infrastructure management, potentially leading to new hybrid models that balance the benefits of reproducibility, scalability, and flexibility.

Bibliography

[Ack] Acknowledgments (GNU Guix Reference Manual), URL https://guix.gnu.org/manual/en/html_node/Acknowledgments.html

[Cac00] Cache (12:14:28 +0200 +0200), URL https://docs.docker.com/build/cache/

[Cen24] Central Processing Unit. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=Central_processing_unit&oldid=1243367426

[Cgr24] Cgroups. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=Cgroups&oldid=1225572362

[CIC24] CI/CD. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=CI/CD&oldid=1245821946

[Cona] Container Runtimes, URL https://kubernetes.io/docs/setup/production-environment/container-runtimes/

[Conb] Container Technology, URL https://www.statista.com/study/80311/container-technology/

[Conc] Containerd, URL https://containerd.io/

[Cou13] Courtès, Ludovic: Functional Package Management with Guix (2013), URL http://arxiv.org/abs/1305.4584

[Cri24] Cri-o (2024), URL https://cri-o.io/

[Dep24] Dependency Hell. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=Dependency_hell&oldid=1214980629

[Dev] Developing inside a Container Using Visual Studio Code Remote Development, URL https://code.visualstudio.com/docs/devcontainers/containers

[Dev24] DevOps. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=DevOps&oldid=1245953239

[Doc] Docker Hub Container Image Library | App Containerization, URL https://hub.docker.com

[Doc00a] Docker Engine (12:14:28 +0200 +0200), URL https://docs.docker.com/engine/

[Doc00b] Dockerfile Reference (21:17:19 +0200 +0200), URL https://docs.docker.com/reference/dockerfile/

[Doc22] Docker Homepage (2022), URL https://www.docker.com/

[Dol04] Dolstra, Eelco; de Jonge, Merijn und Visser, Eelco: Nix: A Safe and Policy-Free System for Software Deployment (2004)

[Dol06] Dolstra, E.: The Purely Functional Software Deployment Model (2006), URL `https://dspace.library.uu.nl/handle/1874/7540`

[Hor24] Horn, Clemens: Clemenscodes/Webserver (2024), URL `https://github.com/clemenscodes/webserver`

[Htm] </> Htmx - High Power Tools for Html, URL `https://htmx.org/`

[Ino] Inotify on Shared Drives Does Not Work · Issue #56 · Docker/for-Win, URL `https://github.com/docker/for-win/issues/56`

[Inp24] Input/Output. *Wikipedia* (2024), URL `https://en.wikipedia.org/w/index.php?title=Input/output&oldid=1238506785`

[Lin] Linux Containers, URL `https://linuxcontainers.org/`

[Lin24] Linux Namespaces. *Wikipedia* (2024), URL `https://en.wikipedia.org/w/index.php?title=Linux_namespaces&oldid=1243965513`

[Lis24] Lisp (Programming Language). *Wikipedia* (2024), URL `https://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=1237652289`

[Mal24] Malka, Julien: Increasing Trust in the Open Source Supply Chain with Reproducible Builds and Functional Package Management, in: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, Association for Computing Machinery, New York, NY, USA, S. 185–186, URL `https://doi.org/10.1145/3639478.3639806`

[Mer14] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (2014), URL `https://www.semanticscholar.org/paper/Docker%3A-lightweight-Linux-containers-for-consistent-Merkel/875d90d4f66b07f90687b27ab304e04a3f666fc2`

[Net00] Networking (12:14:28 +0200 +0200), URL `https://docs.docker.com/engine/network/`

[Nixa] Nix Develop - Nix Reference Manual, URL `https://nix.dev/manual/nix/2.18/command-ref/new-cli/nix3-develop`

[Nixb] Nix-Shell - Nix Reference Manual, URL `https://nix.dev/manual/nix/2.18/command-ref/nix-shell`

[Ope24] Operating System. *Wikipedia* (2024), URL `https://en.wikipedia.org/w/index.php?title=Operating_system&oldid=1245759229`

[Pah15] Pahl, Claus: Containerization and the PaaS Cloud. *IEEE Cloud Computing* (2015), Bd. 2(3): S. 24–31, URL `https://ieeexplore.ieee.org/document/7158965/?arnumber=7158965`

[Per00] Persisting Container Data (12:14:28 +0200 +0200), URL `https://docs.docker.com/get-started/docker-concepts/`

running-containers/persisting-container-data/

[Pod24] Podman (2024), URL https://podman.io/

[Pri21] Priedhorsky, Reid; Canon, R. Shane; Randles, Timothy und Younge, Andrew J.: Minimizing Privilege for Building HPC Containers, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, St. Louis Missouri, S. 1–14, URL https://dl.acm.org/doi/10.1145/3458817.3476187

[Pro24] Process Identifier. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=Process_identifier&oldid=1229884810

[Rah22] Rahman, Mohammad M.; Khomh, Foutse und Castelluccio, Marco: Works for Me! Cannot Reproduce – A Large Scale Empirical Study of Non-reproducible Bugs. *Empirical Software Engineering* (2022), Bd. 27(5): S. 111, URL https://doi.org/10.1007/s10664-022-10153-2

[Red] Red Hat - We Make Open Source Technologies for the Enterprise, URL https://www.redhat.com/en

[Rus] Rust Programming Language, URL https://www.rust-lang.org/

[Sof24] Software Portability. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=Software_portability&oldid=1229915379

[Sys24] System Image. *Wikipedia* (2024), URL https://en.wikipedia.org/w/index.php?title=System_image&oldid=1220330204#Process_images

[Tai20] Tailwind CSS - Rapidly Build Modern Websites without Ever Leaving Your HTML. (2020), URL https://tailwindcss.com/

[Und00] Understanding the Image Layers (13:57:13 +0200 +0200), URL https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

List of Abbreviations

API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration and Continuous Deployment
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CT	Containerization Technologies
DevOps	Software Development and IT operations
FPM	Functional Package Management
GHA	GitHub Actions
GHC	Glorious Haskell Compiler
GHCR	GitHub Container Registry
GNU	GNU's not Unix
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure

I/O	Input/Output
ID	Identifier
IDE	Integrated Development Environment
IT	Information Technology
LXC	Linux Containers
OpenSSL	Open Secure Socket Layer
OS	Operating System
PID	Process Identifier
SSL	Secure Socket Layer
TOML	Tom’s Obvious Minimal Language
TUI	Terminal User Interface
VM	Virtual Machine
VS	Visual Studio
VSCode	Visual Studio Code
WSL	Windows Subsystem for Linux

List of Figures

List of Tables

Listings

A Nix Code Listings

A.1 Nix Package Definition

This listing defines the Nix package for the Rust web server, including dependencies and build processes.

```
{
  pkgs,
  filter,
}:
with pkgs; let
  manifest = (lib.importTOML ./Cargo.toml).package;
  inherit (manifest) name version;
  pname = name;
  src = filter {
    root = ./.;
    include = [
      ./src
      ./styles
      ./templates
      ./assets
      ./Cargo.lock
      ./Cargo.toml
    ];
  };
  assets = stdenv.mkDerivation {
    inherit src version;
    pname = "${pname}-assets";
    buildPhase = ''
      ${tailwindcss}/bin/tailwindcss -i styles/tailwind.css -o assets/main.
          css
    '';
    installPhase = ''
      mkdir -p $out
      mv assets $out/assets
    '';
  };
  unwrapped = rustPlatform.buildRustPackage {
    inherit src version;
    pname = "${pname}-unwrapped";
    cargoDeps = rustPlatform.importCargoLock {
      lockFile = ./Cargo.lock;
    };
    cargoHash = "sha256-EYTuVD1SSk3q4UWBo+736Mby4nFZWFCim3MS9YBsrLc=";
    nativeBuildInputs = [pkg-config];
    buildInputs = [openssl];
  };
in
  writeShellScriptBin pname ''
    WEBSERVER_ASSETS=${assets}/assets ${unwrapped}/bin/webserver
  ''
```

Listing A.1: Nix Package Definition for Rust Web Server

A.2 Nix Configuration

The following ‘flake.nix‘ defines the configuration for the Rust web server application, including toolchains and dependencies.

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
    flake-utils.url = "github:numtide/flake-utils";
    nix-filter.url = "github:numtide/nix-filter";
    rust-overlay = {
      url = "github:oxalica/rust-overlay";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    with inputs;
      flake-utils.lib.eachDefaultSystem (
        system: let
          inherit (nixpkgs) lib;
          overlays = [(import rust-overlay)];
          filter = nix-filter.lib;
          pkgs = import nixpkgs {inherit system lib overlays;};
          bin = pkgs.pkgsBuildHost.rust-bin;
          rustToolchain = bin.fromRustupToolchainFile ./rust-toolchain.toml;
          extendedRustToolchain = rustToolchain.override {
            extensions = [
              "rust-src"
              "clippy"
              "llvm-tools"
            ];
          };
        in
```

```
          with pkgs; {
            packages = {
              default = import ./default.nix {inherit pkgs filter;};
            };
            devShells = {
              default = mkShell {
                buildInputs = [pkg-config];
                nativeBuildInputs = [
                  extendedRustToolchain
                  rust-analyzer
                  openssl
                ];
                RUST_SRC_PATH = "${rust.packages.stable.rustPlatform.rustLibSrc}";
                RUST_BACKTRACE = 1;
              };
            };
          }
      );

  nixConfig = {
    extra-substituters = [
      "https://nix-community.cachix.org"
      "https://clemenscodes.cachix.org"
    ];
    extra-trusted-public-keys = [
      "nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRCYg3Fs="
      "clemenscodes.cachix.org-1:yEwW1YgttL2xdsyfFDz/vv8zZRhRGMeDQsKKmtV1N18="
    ];
  };
}
```

Listing A.2: flake.nix for the application

B Docker Code Listings

B.1 Full Dockerfile Listing for Rust Web Server

The following is the full Dockerfile used to build and run the Rust web server application.

```
FROM rust:1.80.1-slim-bullseye AS base

ENV DEBIAN_FRONTEND=noninteractive
ENV SHELL=/bin/bash
ENV PATH="/root/.proto/bin:$PATH"

RUN apt-get update && \
  apt-get install -y --no-install-recommends \
  git=1:2.30.2-1* \
  gzip=1.10-4* \
  unzip=6.0-26* \
  xz-utils=5.2.5-2.1* \
  curl=7.74.0-1.3* \
  pkg-config=0.29.2-1* \
  openssl=1.1.1* \
  libssl-dev=1.1.1* \
  musl-tools=1.2.2-1* \
  make=4.3-4.1* \
  && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*

SHELL ["/bin/bash", "-o", "pipefail", "-c"]

RUN curl -fsSL https://moonrepo.dev/install/proto.sh | \
    bash -s -- 0.40.4 --yes && \
    proto plugin add moon "https://raw.githubusercontent.com/moonrepo/moon/
       master/proto-plugin.toml" && \
    proto install moon

WORKDIR /openssl

RUN ln -s /usr/include/x86_64-linux-gnu/asm /usr/include/x86_64-linux-musl/
    asm && \
    ln -s /usr/include/asm-generic /usr/include/x86_64-linux-musl/asm-generic && \
    ln -s /usr/include/linux /usr/include/x86_64-linux-musl/linux && \
    mkdir /musl && \
    curl -LO https://github.com/openssl/openssl/archive/OpenSSL_1_1_1f.tar.gz && \
    tar zxvf OpenSSL_1_1_1f.tar.gz

WORKDIR /openssl/openssl-OpenSSL_1_1_1f/

RUN CC="musl-gcc -fPIE -pie" ./Configure no-shared no-async \
    --prefix=/musl --openssldir=/musl/ssl linux-x86_64 && \
    make depend && \
    make -j"$(nproc)" && \
    make install

WORKDIR /app

FROM base AS build

COPY Cargo.toml Cargo.toml
COPY Cargo.lock Cargo.lock
COPY .moon .moon
COPY dockerManifest.json dockerManifest.json
COPY --from=base /musl /musl

ENV PKG_CONFIG_ALLOW_CROSS=1
ENV OPENSSL_STATIC=true
ENV OPENSSL_DIR=/musl

RUN rm .moon/toolchain.yml && \
    mv .moon/docker.toolchain.yml .moon/toolchain.yml && \
    echo "id: webserver" > moon.yml && \
    echo "project:" >> moon.yml && \
    echo "  name: webserver" >> moon.yml && \
    echo "  description: webserver" >> moon.yml && \
    moon docker setup && \
    mkdir src/ && \
    echo "fn main() {println!(\"if you see this, the build broke\")}" > src/main.rs && \
    rustup target add x86_64-unknown-linux-musl && \
    cargo build --release --target=x86_64-unknown-linux-musl && \
    rm -rf target/x86_64-unknown-linux-musl/release/deps/webserver*

COPY tailwind.config.js tailwind.config.js
COPY moon.yml moon.yml
COPY styles styles
COPY assets assets
COPY templates templates
COPY src src
```

```

RUN moon run webserver:styles && \
    cargo build --release --target=x86_64-unknown-linux-musl && \
    mv target/x86_64-unknown-linux-musl/release/webserver . && \
    moon docker prune

FROM alpine:3.20.2 AS start

WORKDIR /app

COPY --from=build /app/webserver /usr/local/bin/webserver
COPY --from=build /app/assets /app/assets

ENV webserver_ASSETS=/app/assets

RUN addgroup -g 1000 webserver && \
    adduser -D -s /bin/sh -u 1000 -G webserver webserver && \
    chown webserver:webserver /usr/local/bin/webserver

USER webserver

CMD ["webserver"]
```

Listing B.1: Dockerfile for the Rust Web Server

B.2 Docker Compose File

The following Docker Compose file configures the application to expose port 8000 on the host system and ensures that the application restarts automatically unless manually stopped.

```
services:
  app:
    build: .
    ports:
      - "8000:8000"
    restart: unless-stopped
```

Listing B.2: Docker Compose File for the Application