Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Boost flexibility in coding exercises by building containers at runtime

Bachelor Thesis

R. Stoll

July 10, 2022

Advisors: Dr. D. Sichau (co-head), Dr. M. Dahinden (co-head), Prof. Dr. B. Gärtner

Department of Computer Science, ETH Zürich

Abstract

At ETH Zürich, lecturers can use a software platform called CodeExpert to set up environments for coding exercises every student can access using a browser. These development environments are commonly run inside *containers* on virtual machines. They are currently prebuilt in a long and manual process, which impedes configuration flexibility for the lecturer and makes maintainability a burden for the developer.

The aim is to find and evaluate feasible approaches that solve the challenges of the current approach by creating flexible *environments* at runtime. The new approach should limit security vulnerabilities and improve the developer experience. The potential build time overhead should not poorly impact the experience of students and lecturers. Furthermore, lecturers should be able to efficiently set up a development environment that includes any package and language.

This thesis proposes two approaches relying on the Nix build system that allows building environments at runtime. A prototype was built and evaluated for each approach, following the design and objectives advocated in this thesis. It is shown how the prototypes improve the user and developer experience compared to the current approach.

It is argued that one of our approaches provides the basis for the next version of environments at CodeExpert that will deliver better maintainability, offer more configuration flexibility and have a comparable or better build performance.

Contents

Chapter 1

Introduction

Cloud-based IDEs are becoming increasingly popular in the education and tech industries. They allow users unlimited mobility to develop a programming project in a standardized environment solely using a browser. Users can escape troublesome local installation and configuration without worrying about the runtime and hardware requirements. Cloud-based IDEs (e.g., Github Codespaces) promote collaborative work and foster exchange between the users (e.g., developers, students) involved [49][46]. They often execute the users' code in *containers* to achieve a controlled environment and scalable system. This thesis contributes to CodeExpert, which is a cloud-based IDE developed at ETH Zürich.

Before the objectives and problems are introduced, some essential concepts need to be discussed.

Definition 1.1 (*Execution environment)* An execution environment defines a (virtual) platform on which e.g. a process or kernel executes. In a process, the execution environment could be the virtual address space and available system calls, while in a kernel, it is the machine hardware.

Definition 1.2 (*Container image)* A container image has a standardized format which defines it as a collection of multiple *layers* (stored as files) and a *configuration file*. These files are read-only, and the layers store the image's content. In contrast, the configuration file contains image metadata (providing extra information about the layers) and an ordered list of references to layers used in this image [22][7].

In this thesis, the terms container image and image are used interchangeably. The term *runtime* is used to refer to a program (i.e., a container) that is either executing (in a running state) or has finished the "build" phase (life-cycle phases such as compile and distribute phase) and can be instantly executed on the central processing unit (CPU).

Definition 1.3 (*Container)* A container has two states (resting and running), similar to the states of a process. In the rest state, it is a set of files and metadata (i.e., the container image) saved on a disk. A container can be started from a container image by unpacking the set of files and metadata and making a system call to the kernel to start a process. This system call initiates isolation (for security and performance) from other containers and mounts (i.e., makes available to users) a copy of the container image files. Now the container is in the running state as a process [22][16]. Recall from def. 1.2 that the layers of an image are immutable, which means that a container cannot modify a file from the image file system. This restriction has the advantage that the state of a new container is *reproducible*, meaning that any number of identical containers can be started from the same image [7][18]. Containers are used, among others, to isolate applications from each other running on the same hardware [24].

Users are separated into students who execute code inside the environment and lecturers who configure them. The term "user" is used to refer to both groups. The term *environment* is defined in this thesis as follows and differs from a container image.

Definition 1.4 (*Environment)* An environment is a container that can execute a student's code remotely and provides interactivity, e.g., it returns the program's output. An environment has installed software packages (e.g., libraries and tools) for some programming language (e.g., `C++`, `Python`).

There are two main ways to allow lecturers to set up their environment flexibly. One can provide a fixed set of packages and programming languages as large pre-built images, or one could set up the environment at runtime to the lecturer's needs. The first approach is fast once the image is built but has issues, e.g., installing two different versions `Python` (e.g., 3.7 and 3.9) in the same container image is challenging. Thus today's typical solution is to create an image per language and version, resulting in many images. Also, every newly added package might break existing code in these ever-growing images, which become hard to maintain, test and update. Furthermore, having preconfigured images means that updates to the configuration can take some time until they are deployed on production systems.

The second approach to building images at runtime based on the lecturer's needs seems more promising. Ideally, each lecturer should be able to use any language and install any package with minimal fuss. Some potential advantages of this approach are:

- The lecturer gets more flexibility and responsibility to set up environments
- The lecturer can quickly iterate over and test different configurations

- The developer would only need to maintain and update a few minimal base images
- The developer has minimal responsibility for setting up a correct and secure image.
- It is easy to combine environments and specify predefined configurations (called "presets").

The main challenge of this approach is quickly building an environment at runtime. If the building is too slow, the user experience might be impacted poorly by the long wait time. There is an essential distinction between the environment's first-time build and subsequent builds with the same configuration. The latter's performance is much more important than the former. This is because many students usually share the same configuration and start an environment frequently. Furthermore, all other students should benefit from the one who must wait for the first build.

In this work, we propose two approaches that solve the problem of building environments at runtime based on the lecturer's configuration. Both approaches use the Nix build system (see 2.6), which features, among others, a purely functional and reproducible package manager. Nix allows to build and compose seemingly unlimited environments at runtime. The first approach uses Nix to build a new container image at runtime inside a particular container (called "builder" container). A new image is built for every configuration provided by the lecturer. The second approach uses Nix to build an environment at runtime specified by the lecturer's configuration. Environments are created from a single image by modifying the persistent state associated with the container. Each approach has a corresponding prototype evaluated using four criteria: performance, user experience, developer experience, and security. The evaluation results of our approaches are compared to the current implementation of environments in CodeExpert.

The rest of this work is organized as follows: In section 2, the theoretical background of our approaches and argumentation is introduced. In section 3, the implementation of the two proposed prototypes and the methods used for data collection are described, while in section 4, the benchmark results are presented. This is followed by section 5 with the discussion of the results. The arguments of the discussion are used to conclude in section 6 with a recommendation to which approach is favorable. Finally, further work is described.

Chapter 2

Background

This chapter will introduce the theoretical background of this thesis. We start with a high-level discussion of virtualization, which forms the basis of container-based virtualization and cloud-based IDEs such as CodeExpert. We close this chapter by introducing the Nix build system.

2.1 Virtualization

Virtualization is one of the most fundamental ideas in computer science, and many components of a computing system's software, hardware, and network use it. It has become prevalent and is used by many developers and organizations for cloud computing, security, testing, and development.

Virtualization is commonly defined as a technology that introduces a software abstraction layer between the hardware and the operating system and applications running on top of it. This abstraction layer, called virtual machine manager (VMM) or hypervisor, hides the physical hardware resources from the operating system (OS). This means that an application program cannot differentiate between a virtual and a real execution environment, but the abstraction layer allows the underlying mechanisms to differ. Since the VMM, not the OS, controls the hardware resources, they can be partitioned into several logical units called virtual machines (VM) to allow running multiple OSs in parallel [41].

The two primary advantages of virtualization are generally *resource sharing* and *isolation*. The former is the ability of a virtualized environment to share resources (i.e., memory, disk, and network devices) of the underlying host. For virtualization, it is essential to provide isolation between VMs running on the same hardware.

2.1.1 The uses of VMs

This section briefly covers the essential use cases of VMs, where the use case of cloud technologies is described in more detail since it is important for this thesis.

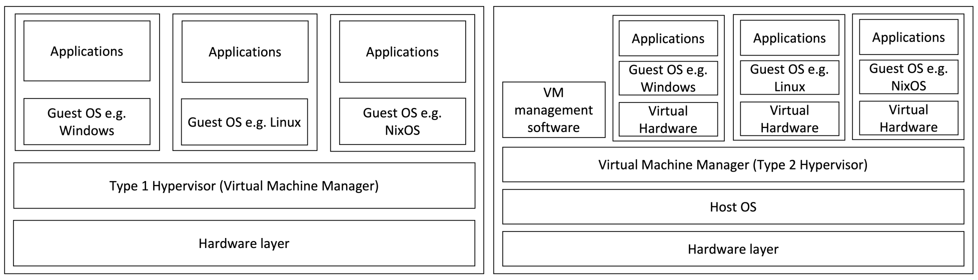
Important use cases of VMs Programmers use VMs to ease software development and testing. Furthermore, they are used in *cloud computing* that enables access to a shared pool of computing resources that can be provisioned and released with minimal management effort [23]. In cloud computing, hypervisors decouple the allocation of resources (VMs) from infrastructure provisioning (physical machines). VMs provide *resource isolation*, which is the property of an OS guaranteeing to one application that others will not impact its performance. The OS does not guarantee resource isolation, as it lacks resource containers [4]. The OS can sometimes perform resource isolation between processes, but applications comprise many processes, and multiple applications share many processes. Thus when multiple applications content for resources (CPU time, physical memory), the performance of one or more may degrade in ways outside the control of the OS.

Cloud technologies Cloud technologies achieve, among others, the following attributes: rapid elasticity and resilience [23]. Resilience refers to handling failure, whereas elasticity is the ability to request more resources on demand. Cloud technology is usually split into two components: infrastructure and services, where, e.g., Docker provides both the infrastructure and platform to host services. Software running as a service is *cloud native* if built from the very first steps to run inside the cloud. Cloud-native applications are generally run inside a container and are resilient and elastic by design [3][45].

2.1.2 Hypervisor-based virtualization

There is commonly a distinction between a VMM and a hypervisor. A VMM is a functionality required to create the illusion of real hardware for a single *guest OS*. Specifically, a VMM creates a *single* virtual machine. A guest operating system is an OS, plus associated applications, running inside a virtual machine. On the other hand, a hypervisor is software that runs on actual, physical hardware and supports *multiple* virtual machines – each with its associated VMM. A hypervisor is usually an operating system and is required to create and run VMs. A hypervisor-based virtualization system can be of two types: *hardware-layer virtualization* or *type 1* and *type 2* hypervisors (see figure 2.1).

In hardware-layer virtualization, the hypervisor runs "on the metal", i.e., it runs directly on the hardware layer as an OS kernel, controlling and synchronizing the access of the guest OS to the hardware resources. A VM



(a) Hardware-layer virtualization or type 1 hypervisor **(b)** Full virtualization or type 2 hypervisor

Figure 2.1: Different types of hypervisor-based virtualization architectures (adapted from [41]).

is represented by a guest OS, which can differ across VMs, libraries, and applications. Figure 2.1a depicts this architecture. It is commonly used due to its high isolation and performance of VMs [41]. IBM VM/CMS, VMware ESX, and Xen are examples of type 1 hypervisors.

Type 2 hypervisors run on top of or as part of the host OS, commonly as an application in *user-space*. The result is that in each VM, the application and the guest OS run on top of the virtual hardware provided by the VMM (see fig. 2.1b). As with type 1 hypervisors, each VM requires a full copy of the guest OS, applications, and libraries [41]. The user-space is a set of memory locations normal user processes can access. It is distinct from the kernel space, which the kernel can only access. VMware Workstation, KVM, and VirtualBox are all examples of type 2 hypervisors.

2.1.3 Container-based OS-layer virtualization

Containers share some characteristics with VMs but are an example of *OS-level virtualization*. OS-level virtualization, represented by the *virtualization layer* – also known as *container engine*, virtualizes a single host OS to provide the illusion of multiple instances or "containers" of that same OS (see fig. 2.2). Unlike hypervisors, each container is constrained to run the same OS as the host. This means that the host's kernel is shared, and applications can safely share system libraries and executables across containers rather than having possibly redundant copies in each VM. Multiple containers can be run in parallel, and each can be started, stopped, and removed by the container engine. The ability of containers to share resources with the host OS and share data with other containers makes them efficient [24][22].

OS-level virtualization achieves isolation by limiting the *filesystem namespace* and the *process namespace*. The process namespace binds a process to its execution environment. The filesystem namespace refers to the organization of the bindings between names and files. The filesystem namespace can be limited by changing the root for each container. The process namespace is

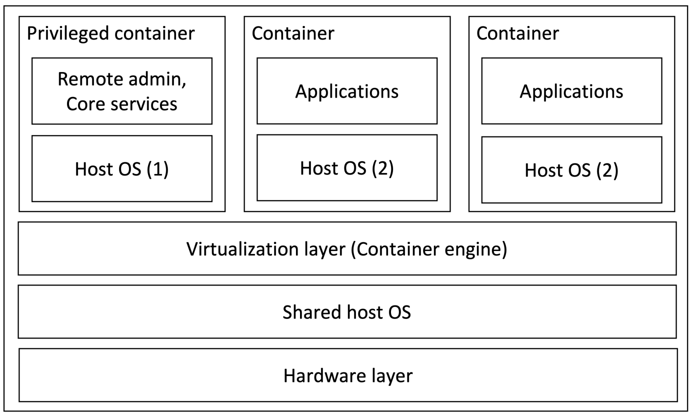


Figure 2.2: OS-Layer virtualization architecture overview as the basis of containers (adapted from [41]).

limited, so processes can only "see" other processes that share their container. The OS or virtualization layer may provide more sophisticated scheduling and memory allocation policies for running containers instead of standard processes. How this is done exactly is out of the scope of this thesis.

A privileged container (see fig 2.2) exists in this system that provides an interface to the virtualization layer. This allows a system administrator to manage other containers and assign resources to containers at the time of creation or dynamically at runtime. These resources could include disk space, CPU guarantees, and memory limits. To applications and the user of a container-based system, the container appears just like a separate host. Processes running inside the container are native processes on the host with the same system call interface as the underlying host OS [43][24].

2.1.4 Containers versus VMs

Containers and VMs can both be used to isolate applications from each other running on the same hardware. They are commonly used to achieve different goals – containers are used to make applications *self-contained* and *portable*, while the purpose of VMs is to emulate a system entirely. Self-contained means that the application does not rely on dependencies external to the container. Portability is hiding the details of an application's execution environment to transfer it to another platform effortlessly. A platform includes hardware, the OS, system users, and software interfaces. Containers are somewhat limited in functionality compared with VMs, but they enable use cases that are difficult to achieve with VMs [24].

The self-containment and portability properties of containers have advantages in cloud deployment and for developers who can download images and run containers without worrying about configuration, differences in user environments, and installing dependencies [24]. Containers can be started and stopped in a fraction of a second. Processes of containers do not incur the overheads of hypervisor execution (e.g., interpreting system calls). The sharing property of a container system makes containers more efficient ("lightweight") compared to VMs. This results in system administrators being able to run many more containers on a single host machine than VMs.

Research confirms the above claim that containers tend to be more efficient and suggests that they guarantee almost the same resource isolation as VMs (at present; some researchers dispute both claims) [43]. However, hypervisors are a trusted and battle-tested technology, whereas containers are comparatively new, resulting in companies not fully trusting the isolation of containers. This lack of trust has led to hybrid systems with containers running inside VMs to take advantage of both technologies [24].

2.2 Containers

Containers are an old idea, where a basic form of system-level virtualization was introduced to FreeBSD in 1998. Then in 2008, the Linux Containers project (LXC) started and brought together system-level virtualization and resource limiting for memory and CPU. The result was a complete containerization solution. The final piece to the containerization ecosystem was brought by Docker in 2013 and enabled widespread adoption of the technology by companies like Red Hat, Microsoft, and Amazon and the developer community. Docker extended existing Linux container technology (i.e., LXC) with a standard container format to allow portable images. It also added a user-friendly interface and tools to build, run and distribute containers [24].

The different available container solutions, each having a different container image format, and the rise of Docker led to the development of an independent formal standard for the container runtime and image format. This standard is known as *OCI Image Specification* after the governance structure Open Container Initiative (OCI) that created it. The OCI Image Specification was based on the original Docker image format and is adopted today by almost all major container engines and tools. LXC and many other container engines and container orchestration options exist, such as OpenVZ and Kubernetes, but Docker remains the industry standard containerization software [24].

This section introduces essential concepts of a container system used in this thesis. The theoretical and abstract view of such a system was introduced in 2.1.3 Since the current CodeExpert approach and our prototypes use Docker, we will use it in the following as an example for some of the more general containerization concepts.

2.2.1 Container image

Container images are already introduced in definition 1.2. An image often extends another image, called the *base image*, with some additional customization, i.e., software packages necessary for running one's application. Images are then used to store one's application and deploy it to a server. In the case of Docker, the instructions defining the steps needed to build the image and run it are written with a `Dockerfile` [16].

2.2.2 Container

A container has been defined in 1.3, and some other important properties are introduced next. Typically, configuration options can be added to a container when starting it, like limiting the memory capacity or setting environment variables. Thus a running container is defined by the container image used as a mount point to start it and the provided configuration options [22]. One can connect a container to subsystems such as network or persistent storage (introduced in 2.2.5) and control the degree of isolation of each container from other containers and the host machine. The container life-cycle ends when removing a stopped container and any state not stored in persistent storage or the image's layers at build time disappears [16].

2.2.3 Image layers

We need to know how layers are created, stored, and cached to understand the uses and performance aspects of images and containers. In this section, we will use Docker as an example to explain the relation between images and layers and the connection between containers and layers.

Images and layers Recall from definition 1.2 that container images are a collection of layers. Layers in an image relate to each other through a parent-child relationship. The union of all layers comprises the content of an image. Each image layer represents the changes between itself and the parent layer. Each parent layer contains the set of differences (`diff`) to the filesystem of the child layer. For example, in the case of Docker, if a `Dockerfile` instruction copies a file into the image's filesystem, this file is located in the `diff` of a layer [18][7]. Each layer is identified by a *content addressable ID* that is the hash value of its content [7]. A content addressable store is analogous to the concept of *content addressable memory* that 'refers to a principle of organization and/or management of memories [...], or searching of data on the basis of their contents rather than by their location [21, p. 1].' The modifications of the `Dockerfile` that result in changes of the `diff` of a layer are cached as a separate layer cache. A container image is built by progressively merging the files inside the layers on top of the base image. The cached layer is used whenever possible during the merging. The merging of layers is possible due to the *union filesystem* that Docker uses that allows stacking multiple filesystems or directories to appear as a single filesystem [18][50].

The two primary advantages of constructing images and layers in this format are efficient storage usage and faster rebuilds. The separation of the image configuration file from the files that store the contents of the layers allows sharing layers among multiple image configurations by referencing the same layer. Thus, for example, if multiple images use the same base image, then the layers of the base image have to be stored only once on disk resulting in efficient disk usage [18]. The cached layers avoid recomputations in an image rebuild after a configuration change. Specifically, almost only those changed layers are rebuilt, resulting in faster image builds [16].

Containers and layers When a new container is started, a new writable layer is added on top of the read-only layers. This writable layer is the significant difference between a container and an image and is why it is often called the "container" layer. This layer allows some write access to containers, as all changes made to the running container, such as writing, modifying, or deleting files, are written to this thin container layer. To achieve this, Docker utilizes a *copy-on-write* mechanism that copies a file from some read-only layer to the writable layer if the container updates the file for the first time. This mechanism implies that containers that write a lot of data, which the copy-on-write mechanism writes to the writable layer, consume more space than those that do not. When a container is deleted, the writable layer disappears, and the underlying image remains unchanged [18]. The copy-on-write mechanism helps to make containers efficient. It saves space, as the read-only layers can be shared among containers. Additionally, it reduces container start-up time, as the container engine only needs to create the container layer [18].

2.2.4 Image registry

A *image registry* is a repository, typically run as a server, for storing images. If the container engine does not have a locally cached copy of the container image, it will automatically try to download ("pull") it from an image registry server. It is also possible to upload ("push") a locally built image to an image registry. One can configure the registries the container engine will use to pull images from – Docker uses, by default, the public registry called Docker Hub. It is also possible to run a private registry instance for example in a container [16][22].

2.2.5 Storage

The following section provides an overview of standard persistent storage options that can be attached as a subsystem to a container. For this section,

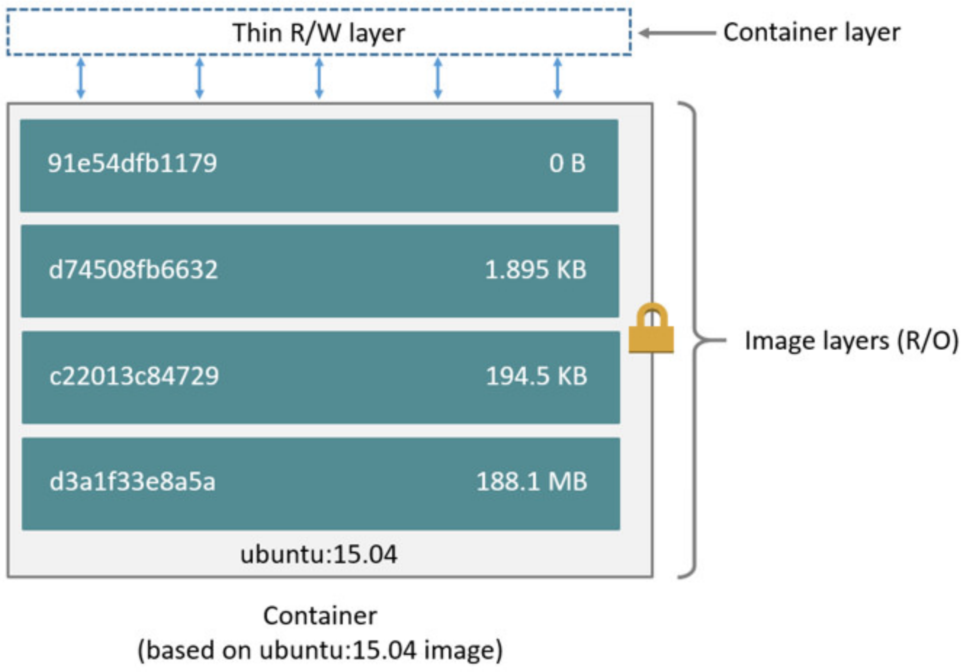


Figure 2.3: Docker container layers based on ubuntu:15.04 image. R/W means that the container can read and write to this layer. Each read-only (R/O) layer is represented by its content-addressable hash and size of the layer [18].

we use the documentation of Docker, but the different storage types apply to other containerization options. Our prototype uses some introduced storage options to persist data across the container life-cycle.

This thesis uses two storage types: volumes and bind-mounts. We can visualize their differences by looking at where the data is stored on the host machine (see figure 2.4). Another common storage type (tmpfs mounts) is out of the scope of this work.

Volumes *Volumes* facilitate persistent data storage that can be shared safely across multiple containers. New volumes can have their content prepopulated by a container. Volumes are often chosen over persisting data in a containers writable layer (see 2.2.3). The reason is that a volume does not increase the size of the containers using it and the data stored in the volume exists outside the life-cycle of a container. In the case of Docker, a volume is stored in a directory within Docker's storage directory on the host machine, and Docker manages a volume by managing that directory's contents (see fig. 2.4) [20][19].

Bind mounts When using a *bind mount*, a file or directory – specified by an absolute path – on the host machine is mounted into a container. The

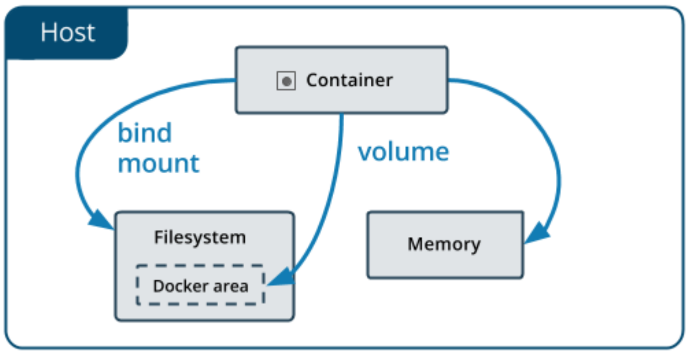


Figure 2.4: Location of where the data of volumes and bind mounts is stored on the host machine (adapted from [17]).

bind mounts may be stored anywhere in the filesystem of the host and can be modified by processes inside the container and on the host (see fig. 2.4) [17]. Bind mounts have limited functionality and are much less performant on Windows or Mac hosts compared to volumes [19][20]. Nonetheless, they help mount a small number of specific files into a container and are very performant on Linux hosts, provided that the host has the right filesystem directory structure [19].

The containerization concepts from this chapter and the virtualization chapter form the basis for the next chapter on cloud-based integrated development environments that often use containers to execute users' code remotely.

2.3 Cloud-based integrated development environments

This section introduces cloud-based IDEs and their difference from traditional IDEs. Additionally, we cover the implications of a cloud-based IDE on how a programming environment should be set up.

Traditional Integrated Development Environments (IDEs) accelerate software development by providing an effective way to browse and manipulate a system's source code as opposed to using a plain text editor and command line [9] [14]. A traditional IDE provides developers with a development environment where the IDE is commonly responsible for tasks such as dependency management, debugging, and auto-completion [3]. Popular IDEs include Microsoft Visual Studio, Eclipse, and NetBeans.

A cloud-based IDE is run in a cloud computing system, and developers can access the IDE from any web browser at any time [52]. Compared to a traditional IDE, which must be installed before using it, each developer does not have to install dependencies and compilers on the local system. Cloud IDEs set up a shared workspace, which is shared among the programmers. Furthermore, they provide a large pool of computing resources for develop-

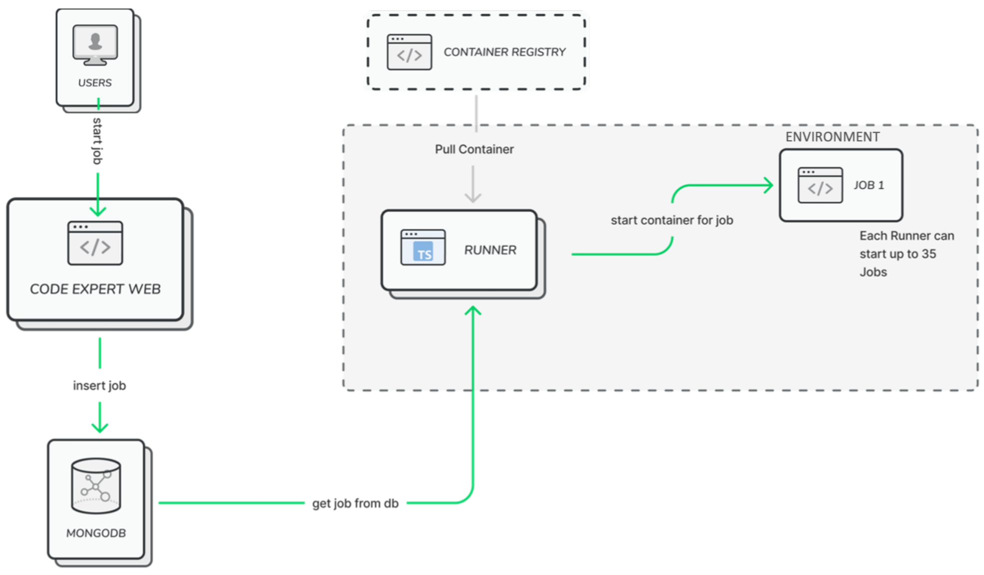


Figure 2.5: CodeExpert architecture and job execution flow (adapted from [42])

ment to support developer collaboration. Popular cloud-based IDEs include CodeSpaces, Cloud9, Replit, and Eclipse Che.

A cloud IDE should offer programming languages as runtime execution environments as developers may want to use different programming languages. These environments are often executed inside containers, making them cloud-native applications. Furthermore, a standard runtime environment configuration can be shared among developers [3][14].

2.4 CodeExpert

Since this thesis compares its results to the current implementation of *CodeExpert* and discusses its tradeoffs, we need to understand its architecture and general workflow.

2.4.1 Architecture

CodeExpert is a cloud-based IDE developed at ETH Zürich that is used to enhance computer science studies and programming classes. The architecture consists of four primary services, split into the front-end (*cxweb*) and the back-end. The back-end includes the database, back-end runner (*cxrun*), and image registry (see fig. 2.5). The CodeExpert IDE is accessed via web browser and executed inside containers. The front-end is separated from the back-end through interfaces. Each service is running on multiple different virtual machines. This decoupling of the front-end from the back-end allows the system to scale – as the workload increases – the back-end and front-end services independently.

2.4.2 Execution flow

For each interaction (e.g., executing or compiling code) of the student, cxrun starts a new cloud-native environment (*cxEnvironment*) in which the code is executed remotely. After the execution has terminated, the container is removed, and the execution output is redirected by cxrun to cxweb. This allows the student to inspect the output [11].

2.4.3 Configuration

The lecturer explicitly specifies the cxEnvironment for each project, and the lecturer can use predefined cxEnvironments for programming languages such as `C/C++`, `Python`, and `Java`. Lecturers can also define a custom cxEnvironment using a `Dockerfile`, which is then built to an image and pushed to the image registry (denoted as "container registry" in fig. 2.5). The image registry runs in a container inside the local network. The building and pushing are done at build time before the custom image can be used as runtime environment [11].

2.5 Build system

After our discussion on cloud-based IDEs and CodeExpert, this section will introduce some properties and tools of a *build system* that we will use to build images at runtime.

2.5.1 Reproducibility

Coding environments that lectures use for their student should be *reproducible*, as they are, for example, used in exams. Reproducibility of environments means building the same configuration twice should yield the same environment. To achieve reproducibility, we must manage software versions and dependencies and manage development environments. The latter is commonly handled with containers (e.g., Docker) and VMs that provide controlled environments in which the user code can be executed [6].

We have already learned that a container image allows reproducible containers (see definition 1.3). However, the Docker image build system does not guarantee reproducible images. When building images twice from the same `Dockerfile`, one might get two images that behave differently. The different behaving images could break the user's runtime environment. This non-determinism happens, for example, when a third-party package dependency gets silently updated or when the package version is not specified ("pinned") when writing the `Dockerfile`. The silent update results from non-purely-functional package managers (a concept explained in 2.6), whereas not pinning the package version is a terrible practice. Pinning all package versions helps, but there are many details to consider, and it does not relieve the issue completely [40].

The non-reproducibility issues of building images with `Dockerfile`'s motivate the use of a purely functional package manager and a system to build the coding environment, called a *hermetic* build system. Given the same input source code and configuration, this system returns the same output using isolation and *source identity*. This is the topic of the next section.

2.5.2 Hermeticity

A *hermetic* build system isolates the build from the underlying host system. Thus a hermetic build is insensitive to libraries and software installed on the host machine and depends only on specific versions of build tools (e.g., compilers) and dependencies (e.g., libraries). The result is self-contained builds [5].

A hermetic build tries to ensure the consistency and sameness of inputs, a concept called source identity. Source identity is achieved with the help of code repositories, such as Git, where the unique hash code that identifies code mutations is used to track changes to the build's input. More specifically, isolation between the host machine and the user is achieved by downloading copies of tools and managing their storage inside managed file trees [5].

The benefits of having a hermetic build system are speed, multiple parallel builds, and having multiple versions of the same package coexist. Builds can be cached unless the inputs change, resulting in fast builds. Parallel builds can be executed by computing the graph of actions to take such that the output is built from the input. As the storage of the packages is managed by the hermetic system, we can easily use different versions of tools in multiple builds without colliding installation paths. Thus, we do not need to create a different container environment for different package versions [5][39]. Furthermore, a hermetic build system implies reproducibility [5]. Since the build output does not depend on anything outside the build, we can copy it across different systems and potentially make it portable.

2.5.3 Package manager

Software platforms strive to provide modular software components, called software *packages* that can be assembled to provide the user with the desired functionalities. A package is an archive file containing a program and necessary metadata. The program can be in *source code* that must be compiled and built first. The metadata may include, among others, the package version and description and requirements for relationships (e.g., versions) to other packages and the target system. Installing more than one copy of a package on a given system is impossible. Packages cannot be composed to build a larger component, and they may conflict with each other. The latter can happen because the installation and execution of packages act on shared resources provided by the OS, like creating files or interacting through the systems input/output devices. The essential tools for managing software versions and their dependencies (installing, upgrading, and removing packages) are *package managers*. They allow to retrieve packages from central (remote) repositories and check their integrity. Furthermore, they resolve conflicts between package requirements and provide tools to manage the installation on a user's system. Resolving requirements conflicts is a functionality known as *dependency solving* and is introduced next [1][6].

2.5.4 Dependency management

Dependency management is usually specific to your OS, programming language, or application and is either done by an IDE (see 2.3) or by a separate package manager. Some examples of popular package managers include `dpkg` for Linux and `Conda` or `pip` for the `Python` language. Traditionally the IDE or package manager installs dependencies on one's local machine to set up a development environment. Thus a cloud-based IDE also needs to handle dependency management for setting up the correct runtime environment. The offloading of dependency management and maintenance to the cloud IDE relieves the developer from the burden of setup, configuring, and upgrading their environments [52].

2.6 Nix

In this thesis, we use Nix as a container image build system that includes a purely functional package manager to achieve reproducible development environments that are hermetically built. Being functional means that a build has no side effects – building a package twice with the same inputs yields the same output. This property enables Nix to compose environments at runtime easily. This is unlike many widely used package managers such as `dpkg`, `rpm`, which are not purely functional and mutate the global state (write to `/bin`, `/usr`, and `/etc`) of the system during package management operations [39][47].

Nix provides multiple tools to make builds reproducible. The first is a purely functional domain-specific language called Nix. Second, it provides a Nix store that safely stores multiple versions and variants of packages next to each other. Furthermore, it features atomic upgrades and rollbacks of package versions and builds and a garbage collector, among others [47].

2.6.1 Derivations

Nix stores all packages in isolation and as immutable file system objects (i.e., files and directories) in the Nix store. Packages or software components are build outputs referred to as *derivations* inside Nix. Derivations are built from declarative specifications, called *Nix expressions*, written in the Nix language. They describe all inputs such as sources, build script, and environment variables that go into a package build action. Each build action first constructs an isolated environment with all the inputs needed to run the build script and then executes this script to run all the build steps [47][39]. All inputs, dependencies, and steps of a build process can be represented as a dependency graph, called *closure*. This graph has build actions as its nodes [28]. Nix can construct a new reproducible environment with additional packages by adding new nodes to the closure. Nix ensures a high degree of reproducibility of each Nix expression by removing many side effects such as limiting network and filesystem access and building in an isolated ("sandboxed") container. Additionally, it clears all environment variables and patches build tools (such as `GCC`) to ensure that they do not mutate the global filesystem [47].

Every derivation has a unique subdirectory `g32imf6...-firefox-1.0.1` in the Nix store (see fig. 2.6) where part of the name `g32imf6...` is a cryptographic hash code of the packages build dependency graph (closure). The closure captures all dependencies and versions of a package. A cryptographic hash of the closure is used so that different versions of dependencies result in different hash codes. This implies that it is safe to install different versions without conflicts [47]. This mechanism is the basis of the Nix stores content addressability feature. Nix expressions describe how to build derivations from source code, which can take a long time, as potentially all dependencies in the dependency graph need to be built. To avoid this, Nix skips building from source code if the corresponding derivation is available in a binary cache. By default, Nix downloads pre-built binaries from cache.nixos.org [39].

2.6.2 Package management

Nix provides a collection of Nix expressions, called the *Nix package collection* (Nixpkgs), which is a git repository that contains packages as Nix expressions ranging from build tools like `GCC` and `Glibc` to user applications like Mozilla Firefox. It is possible to write one's own package collection or extend Nixpkgs. To easily stay up to date with new versions of Nixpkgs, we can use the *Nix channel* mechanism [26]. A nix-channel is a name for the latest "verified" commits in Nixpkgs, where "verified" is defined on a channel basis. These channels provide binary builds of packages that are cached in the Nix binary cache. One can, for example, add different channels and switch between

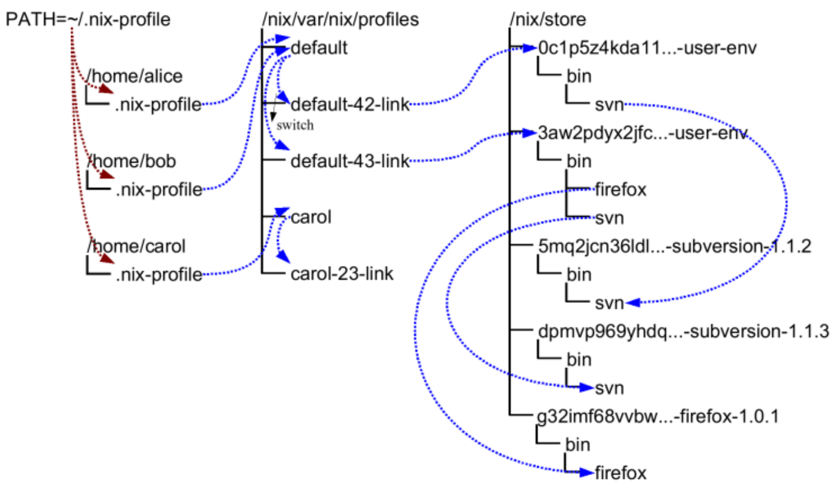


Figure 2.6: Directory tree of symlinks in the Nix store (`/nix/store`) that define the active packages (and their dependencies) of a user environment [37].

them on a user-level basis [33].

2.6.3 User environments and multi-user mode

In Nix, *user environments* allow atomic updates, atomic rollbacks, and users to have custom configurations. Different users may "see" different subsets of the set of all installed applications on the system. This subset is specified in a directory in the user's `PATH` and is called a user environment [31]. Nix uses a directory tree of symlinks inside the Nix store depicted in figure 2.6 to achieve the linking between user environments (`/nix/store/0c1p5z4...-user-env`) and the active packages (e.g. `subversion-1.1.2`) they use. The discussion of Nix profiles is out of the scope of this work [37].

Nix has a multi-user mode that allows multiple users to safely share a Nix store by ensuring that users cannot arbitrarily run builders that modify the Nix store or database or interfere with other users' builds. User-space limitations achieve this: the Nix store and database are owned by a privileged user, and builders are executed under special user accounts. If an unprivileged user runs a Nix command, then actions that operate on the Nix store are performed by the *Nix daemon* running as the owner of the Nix store [32].

2.6.4 The `nix-shell` and `nix-build` commands

The `nix-shell` command helps reproduce a development environment for a package without building the derivation [31][34]. Given a Nix expression, the command `nix-shell` constructs an isolated environment with all the inputs needed to run the build script but does not execute the build script as the `nix-build` command. To this end, `nix-shell` builds the dependencies of the expression from the source or downloads them from a binary cache if they are not already in the Nix store. It will then source a setup script that sets all necessary environment variables (e.g., the `PATH`, compiler search path, Nix profile) to their corresponding values. After that, the command starts a shell in which all the environment variables are set.

In short, the `nix-shell` command only sets up a development environment from the expression without building a new derivation. We use the `nix-build` command to build new derivations from an expression.

2.6.5 Nix functions for Docker compatible images

Nix provides a set of functions (`dockerTools`) for creating and manipulating container images. These functions do not need Docker itself for their operations. The `buildLayeredImage` function builds an image with many of the store paths on their own layer, and each is realized as a tarball. The `streamLayeredImage` function streams an uncompressed tarball of an image to `stdout`, which avoids realizing the image into the Nix store, therefore saving on I/O operations and disk/cache space [36]. Both commands' tarball(s) can be loaded into Docker or pushed to a registry.

Chapter 3

Solution approaches and methodology

This chapter describes and justifies approaches and corresponding prototypes that solve the objective of finding feasible approaches to building development environments at runtime based on the lecturer configuration. In addition, we describe how we collect the measurement data for our results.

3.1 Why Nix as image build system

The current approach of CodeExpert uses Docker as a build system to create container images for different environments and different versions of environments with all required dependencies packaged in the same image. In this section, we will discuss some of the tradeoffs with this approach to justify why we chose Nix as a build system to solve our objective.

Ideally, we want to build reproducible images that contain only the dependencies of the environment and nothing else for repeatable builds to minimize the image size for performance and the attack surface for security.

3.1.1 Issues with the current approach

Apart from the reproducibility issues that arise when building Docker images (see 2.5.1), the `Dockerfile`'s imperative language cannot always capture the exact dependencies of an image. For example, to configure a `Python` environment, one commonly manages dependencies in multiple ways, such as a base image, copying files into the image, and multiple package managers like apk and pip. Some of these tools are only needed at build-time (e.g., pip, apk) and others at runtime, but both are included in the final image [48]. Another thing to notice is that one cannot take two images and combine them. For example, suppose one wants to combine two configurations, a `C++` and a `Python` image. In this case, one has to start with either one of them as a base image and install the dependencies of the other image using `Dockerfile` instructions.

To remedy the problem of capturing the dependencies, one could use multistage Dockerfiles. However, this requires optimizing by hand and is just a workaround of proper dependency management [48].

3.1.2 Advantages of building images with Nix

The Nix build system solves the above problems and builds reproducible images that can then be deployed using the advantages of a container ecosystem such as Docker. Nix allows us to install packages in an image without using a base image. Additionally, we can declaratively separate the build from the runtime dependencies so that only the necessary dependencies end up in the final image [48]. The result is "cheap" images in terms of image size.

In addition, the properties of Nix introduced in 2.6 enable the following advantages:

- Composition of configurations (i.e., base and preset configuration) and environments (e.g., Java and Python)
- Efficient caching of dependencies, image layers, and environments for building images at runtime
- Simple customization of the configuration (e.g., specific package version) by the lecturer
- Developers and lecturers can share package configurations, e.g., in a Nix package repository

We introduce what we mean by *base* and *preset configuration* in section 3.3.

3.2 Solution approaches

In this section, we introduce the idea of our solution approaches and discuss alternative approaches. We will explain the exact implementations of our approaches in later sections of this chapter.

3.2.1 Approaches for building images at runtime

We have developed two feasible approaches that solve our first objective. Each approach's goal is to create an isolated environment at runtime where all dependencies are installed, given an environment configuration by the lecturer. Our approaches should integrate into the existing architecture of CodeExpert. Thus our approaches build environments similar in purpose and function to the cxEnvironment containers (see 2.4), which has the following implications for the design of our prototypes.

Apart from prioritizing the subsequent-startup time over the first-build time, our design considered that environments only execute a single job and are removed afterward. Ultimately, the images corresponding to the environments need to be pushed to a central registry where they are stored.

Build image at runtime (BIAR) BIAR builds a new image at runtime for every new configuration. The essential idea is to have "builder" containers with Nix installed that build a new image based on the lecturer configuration and then push the resulting image to a registry. Afterward, we can pull the image from the registry and start an environment with it. An environment does not need Nix installed, as all packages are added to the image at build time. Any number of builder containers can run in parallel and operate on a shared Nix store.

Nix-shell at runtime (NSAR) Unlike BIAR, NSAR does not build a new *image* at runtime. Instead it creates a new environment using the `nix-shell` command with all packages in the configuration installed (see 2.6.4). The `nix-shell` is run inside a container that needs to be started first and has Nix installed. NSAR creates all these containers from a single image, and their Nix installation shares the same Nix store.

3.2.2 Alternative approaches

We considered using a tool of the Nix ecosystem called Nixery as a different approach somewhat similar to BIAR for solving our problem. Nixery provides ad-hoc containers, including any packages from a package repository, such as Nixpkgs (see 2.6.2). The images are built by Nix using a particular layering algorithm, optimizing cache efficiency. Nixery has a custom caching algorithm for layers that extends the `buildLayeredImage` function of Nix [25].

This approach proved too inflexible, as one can only specify an environment by providing a list of packages by name, thereby losing the flexibility of a Nix expression [25]. Specifying packages by name would not allow lecturers to easily use custom packages with specific versions or environments with an interpreted language. Therefore we have dropped this approach from further consideration.

3.3 Configuration

Both approaches share the same configuration scheme. There are two groups of configurations: *base* and *preset*. The base group, denoted by C_{base}, contains a single configuration file that specifies the dependencies used by every environment. An example configuration file that specifies the GNU Core Utilities and Bash is shown in base.nix.

base.nix

```
{ pkgs }:
{
  inputs = with pkgs; [
    coreutils-full
    bashInteractive
  ];
}
```

The preset group, denoted by C_{preset}, contains predefined configurations (e.g., `Python`, `C++`) from which the lecturer can choose one configuration and fully customize it – potentially removing all packages. An example preset configuration for `C` is shown in preset.nix.

preset.nix

```
{ pkgs }:
{ inputs = with pkgs; [ gcc gdb ]; }
```

Sometimes we talk about the entire package set that is available in an environment, called C_{result}, that is the union of C_{base} and C_{preset}. To continue our example, the C_{result} of base.nix and preset.nix is the union of the package sets and shown in result.nix.

result.nix

```
{ pkgs }:
{
  inputs = with pkgs; [
    coreutils-full
    bashInteractive
    gcc
    gdb
  ];
}
```

All configuration files, in particular the configuration that the lecturer specifies, have a specific interface that restricts the definition of dependencies to a list of inputs – denoted by the `inputs` attribute. The interface establishes that the build process can correctly read (and eventually join) the dependencies from the configuration files. Furthermore, in the case of a file in C_{preset}, this interface has two additional purposes. First, it removes syntax – simplifying the expression the lecturer needs to write. Second, it restricts the lecturer from using arbitrary Nix expressions as a configuration. The interface may need additional attributes to allow the lecturer to specify, for example, environment variables such that the interpreter finds all libraries.

We split the configuration to show how easy it is to merge different configurations with Nix and better separate and combine different configurations for our measurement process. The two configuration groups could be unified from the beginning into a single file that the lecturer can edit. This file would make all environment dependencies directly transparent to the lecturer.

3.4 Prototype: Build Images At Runtime

This section describes the prototype of the Build Images At Runtime (BIAR) approach and its components.

3.4.1 Components

The essential component of our prototype is the *builder container* (see fig. 3.1). This container is started every time a new image needs to be built from a configuration. It has Nix installed and has a persistent and shared Nix store mounted as a volume into the container. Using shared persistent storage for the content-addressable Nix store makes it possible to have multiple builder containers perform parallel builds using the store as a shared build cache.

The container takes the configuration as input and uses `nix-build` to build a new layered image. Depending on the build function used ((1) `streamLayeredImage` or (2) `buildLayeredImage`), the output of the build is then pushed to a registry by either (1) streaming its compressed layers to the registry or (2) copying the compressed tarball from the Nix store to the registry (see 2.6.5). We build a layered image since this allows Nix to optimize the caching of the layers. Efficient caching is essential for this approach as we want to utilize the heuristic that configurations often share most packages to build new images as fast as possible at runtime.

Our prototype includes a "BIAR-data-container" container that creates a volume with the shared Nix store inside and can be used to perform management operations, e.g., running the garbage collector on the store's data.

3.4.2 Execution flow

The execution flow differs depending on the first or subsequent times the image is built from the same configuration. We assume in the following that a builder container has been started with an arbitrary but fixed C_{result} as input (see fig. 3.1).

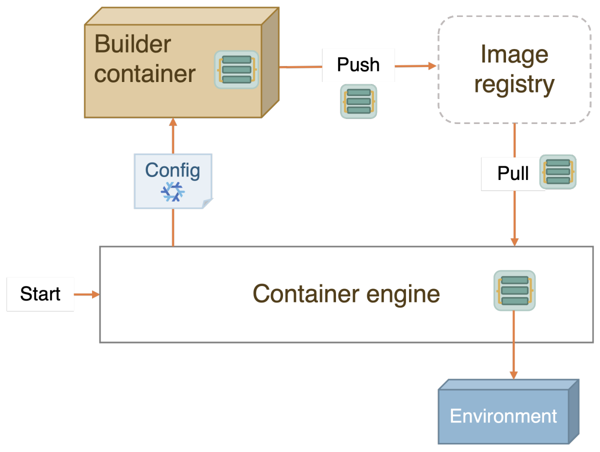


Figure 3.1: Build images at runtime approach: components and execution flow

In the case of the first-time build, the Nix store is empty, and Nix must either download the needed derivations from a binary cache or build them from source code and write them to the store. Once all derivations are available, Nix builds a layered image and pushes the layers to the image registry. The container engine on the host then pulls the image from the registry and starts a container from it, where the environment is ready. The host container engine knows which image to pull and start using the mechanism explained in (3.4.3).

In the case of subsequent-time builds, Nix can omit to rebuild the derivations and layers and use the layers cached in the Nix store. We explain how Nix caches layers efficiently in section 3.4.4. This caching allows subsequent builds to be almost instant. However, we cannot cache pushing the image to the registry with our implementation, whether it is the first or subsequent push of the same image. The key idea to make subsequent builds quick is to notice that subsequent-time builds with the same C_{result} imply that a copy of the image is already cached locally on the host container engine. Thus we can omit the build and the push step and start an environment from the cached image. This execution flow requires that the container engine must determine if an image has already been built with a given C_{result} (see 3.4.3).

Another thing to notice is that if the lecturer changes his configuration only by a "small" amount (i.e., adds/removes a few packages) in a subsequent build, then we can expect that most derivations and layers are cached in the store from a previous build. We expect this since many packages share the same base dependencies [10]. Since Nix can cache layers and derivations efficiently, the build of a small configuration change will be faster than a first-time build.

3.4.3 Identify image from configuration

Generally, a container image can be identified by its name and tag. We used the following convention in our prototype: the image name is the hash value of the base configuration's content that does not often change, whereas the image tag is the hash value of the lecturer configuration file.

Since we build a new image for every configuration change, we need to be able to identify an image from its configuration uniquely. This means we need an injective function $f\colon X \to \mathbb{H}$ with domain $X = (f_1, f_2, \dots)$ of configuration files f_i. However, we use the MD5 hashing algorithm as our function f, which is not injective and has collision vulnerabilities [44]. Nevertheless, we use this algorithm because performance is critical, random collisions are rare, and we do not need to worry about collision attacks from lecturers (explained in 5.2). Performance matters, as we must compute the function F before starting any environment.

3.4.4 Efficient caching of layers

The efficient caching of layers is essential for this approach as changes to configurations will likely result in only very few changes of dependencies that need to be built or rebuilt. If we can cache the building of as many unchanged dependencies as possible, it would result in a faster build time and less disk usage.

It is not easy to make the caching of layers efficient as most package managers mutate global directories (see 2.6), resulting in having to take a diff of the whole filesystem (see 2.2.3). This is similar when using Docker as the image build system, where Docker only sees the diff of the whole filesystem and cannot determine the exact file changes resulting from, e.g., package manager operations. Therefore it cannot know that installing package "foo" has nothing to do with the package "bar" and that the changes are separately cachable. The caching behavior can be improved by manual optimization in a `Dockerfile`, for example, adding one's files after installing packages or installing all packages in a single instruction [10].

Nix caches dependencies and improves sharing between images by having layers equal to a store path representing a dependency. Nix can automate the cache optimization because it restricts package builds to writing to specific places on disk (namely the $out directory of the Nix store). Other reasons are that Nix knows all dependencies from the dependency graph and stores build outputs in immutable files and unique paths. Since the Docker image specification limits the number of layers, Nix combines the dependencies that are less likely to be a cache hit into one layer. Dependencies that are shared the most by multiple packages are more likely to be a cache hit. Thus Nix makes separate layers for these "popular" dependencies. Using more layers increases the number of layers that could be shared between images. The shared layers are likely not affected by a configuration change. This avoids rebuilding and increases the caching efficiency [10]. Both the `buildLayeredImage` and `streamLayeredImage` (see 2.6.5) functions use the optimized layer caching algorithm.

3.5 Prototype: Nix-Shell At Runtime

This section describes the prototype of the Nix-Shell At Runtime (NSAR) approach and its components.

3.5.1 Components

Our implementation of the NSAR approach has *one image*, built with Nix. It is based on the `nixos/nix` image from the Docker Hub, which has Nix and some runtime dependencies required by Nix installed. In this base image (see fig. 3.2), we install the dependencies of C_{base} at build time, as they are the same for every container.

Our prototype has a container called "NSAR-data-container" that creates two volumes similar in purpose to the BIAR-data-container. One for the persistent shared Nix store and the other for the persistent `nix-shell` cache explained in 3.5.3. Both caches are pictorially depicted as one shared cache in fig. 3.2.

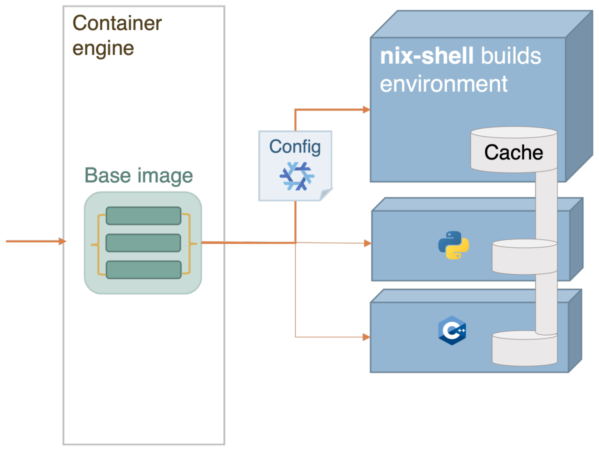


Figure 3.2: Nix-shell at runtime approach: components and execution flow

3.5.2 Execution flow

Unlike BIAR, NSAR does not build a new image at runtime; instead, this approach creates an environment at runtime inside a prebuilt container. We assume that a container was started from the base image and has an arbitrary but fixed lecturer configuration from C_{preset} as input (see fig. 3.2).

The execution flow differs depending on the first or subsequent times `nix-shell` creates the environment from the same configuration. In the case of the first-time build, the Nix store is empty. The empty store results in the `nix-shell` having to download the derivations from a binary cache or build them from source code. In the other case, we can assume that the Nix store contains all derivations from previous builds with the same configuration, and the `nix-shell` will skip the downloading or building step.

The execution flow continues with the `nix-shell` setting up a development environment, as described in greater detail in 2.6.4 from the lecturer configuration. In particular, our prototype calls the `nix-shell` command with a proxy file (`nixproxy.nix`) as input that defines a new environment with the given configuration's dependencies.

The purpose of this proxy file is twofold: to check the input interface of the configuration files and fix, among others, the version of the package collection (e.g., of Nixpkgs). Another thing to notice is that the interface to the lecturer in the configuration files of C_{preset} can be extended for this approach with a `shellHook`. The `shellHook` is run after the setup file has been sourced (see 2.6.4). This hook performs initialization specific to the `nix-shell` and allows the lecturer to define bash statements that will be executed when starting the shell [34]. This shellHook is very useful, for example, to export environment variables, and create hidden output files.

Recall that the Nix store is persistent and shared across the environments as a volume. This sharing is necessary so that only the first student will experience the first-time environment build discussed above. All other students will experience the "subsequent build time" case as the Nix store acts as a build cache, provided they use the same configuration. This mechanism ensures that subsequent builds are quick and that only the first environment build is slow, as our specification requires.

3.5.3 Caching the `nix-shell` execution

To make subsequent build times of an environment faster, we would like to skip the last remaining step of the `nix-shell` execution: the evaluation of the setup file (see 2.6.4). It is possible to omit this step by caching its result: the exported variables and reuse them on subsequent runs. This caching can be accomplished with different tools in the Nix ecosystem, such as `direnv` with `Nix-direnv` or Nix flakes.

In this prototype, we use the `cached-nix-shell` tool [51], as it traces all configuration files used during evaluation and performs cache invalidation if any used files change. This tool stores the inputs and outputs of the derivation and the traces (configurations) of the nix-shell invocation as content-addressable files inside the `~/.cache/cached-nix-shell` directory.

To persist this cache after the container lifecycle, we store the cache of this tool as a volume. The persistence is necessary since the speedups from the subsequent use of the same configuration need to remain over multiple container startups (see 3.2.1).

3.5.4 Seeded Nix store

As an additional consideration, the prototype allows for using a seeded store, which means we prepopulate the store with derivations (with the `nix-env` command; introduced in 2.6.2). A store with the most popular derivations installed can improve the first-time build performance of an environment as Nix can skip downloading derivations from the binary cache if they are already available in the Nix store.

3.6 Methods for data collection

To evaluate our prototypes, we performed measurements using a virtual machine running on Ubuntu 20.04.4 LTS. After two "warmup" iterations, we measured each command for ten repetitions. The warmup phase ensures that we measure the steady-state performance of our machine as there is an initial overhead from empty caches and TLBs, causing costly misses for memory accesses.

3.6.1 Statistical test

We computed the pairwise significance between the means of each benchmark data sample to compare our results based on a statistical test. We assume independence between two samples since they are measured independently. If the variance of both samples is the same, we used a two-sample t-test; otherwise, we used Welch's t-test, which does not assume equal variance. We followed a common rule of thumb to decide whether the variances are similar (the formula is explained in A.2). We use the p-value to assess the statistical significance with a level of 0.05.

3.6.2 Software

We ran the benchmarks with a custom bash script that uses the GNU Time command to measure the process time statistics. We used our results' *Real time* part of the time commands output. This is the wall clock time – the time from start to finish of the call and is the actual elapsed time of the process, including time spent blocked, e.g., waiting for I/O and time slices used by other processes.

3.6.3 Hardware

The remote virtual machine's hardware comprises two virtual CPUs (Intel Family 6 Model 63 Stepping 2) running at 2GHz with 2GB of memory. A virtual CPU corresponds to a single hyperthread on a processor core. The VM system includes a 60GB SSD disk, and the memory throughput could not be determined as the VM service does not support this information.

3.6.4 Test environments

We evaluate our prototypes in two different test environments to verify if our prototypes perform differently in one or the other environment. The first is a `Python` environment, an example of an interpreted language. The second is a `C/C++` environment as an example of a compiled language. The packages used for the benchmarks included in each environment are listed in A.4.

Chapter 4

Results

We evaluate our prototypes using four evaluation criteria: *performance*, *user experience*, *developer experience,* and *security*. Each section in this chapter is dedicated to one criterion.

4.1 Performance

We assess the performance of our prototypes for the two test environments, `C++` and `Python`. We first handle the results concerning the implementation details of each prototype. Afterward, we compare them with each other and to the current approach of CodeExpert on the two build cases: first-time and subsequent-time build. We refer in the following to a "full Nix store" if we benchmark a subsequent-time build where Nix can use the cached results in the Nix store of the first-time build. An "empty Nix store" contains no pre-downloaded binaries of packages.

We made the measurement process of all approaches as comparable as possible by deciding what execution flow resembles the corresponding build case in the best way for each approach. This decision is made separately due to the different implementations and is explained at the beginning of sections 4.1.3 and 4.1.4. In every benchmark that measures an environment's startup time, the time needed to stop and remove the container is also included in the measured real-time. Therefore the time until the environment is ready to execute the user's code is faster than what was measured.

4.1.1 BIAR: approach specific

For the BIAR approach, we want to determine which strategy (layered or streamed) is better to build images in the builder container and push them to the registry (see 3.4.1). We use a registry running inside another container on the same host to simulate the current image registry execution flow of

CodeExpert (see 2.4.3). We only show the figures of the `Python` environment, as the differences between the strategies are similar for the `C++` environment.

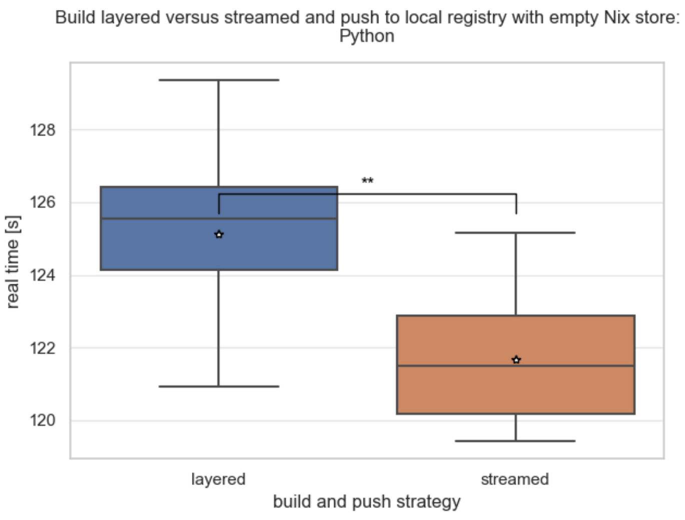


Figure 4.1: Comparing the build and push strategies streamed and layered using an *empty* Nix store. Remarks: * = a higher number of stars(*) indicates a greater statistically significant difference

When the Nix store is empty, we observe that streaming is significantly faster for both `Python` and `C++` environments than the layered strategy (see fig. 4.2. However, when we used a full cache (Nix store), there was no significant difference between the strategies, as shown in figure 4.1.

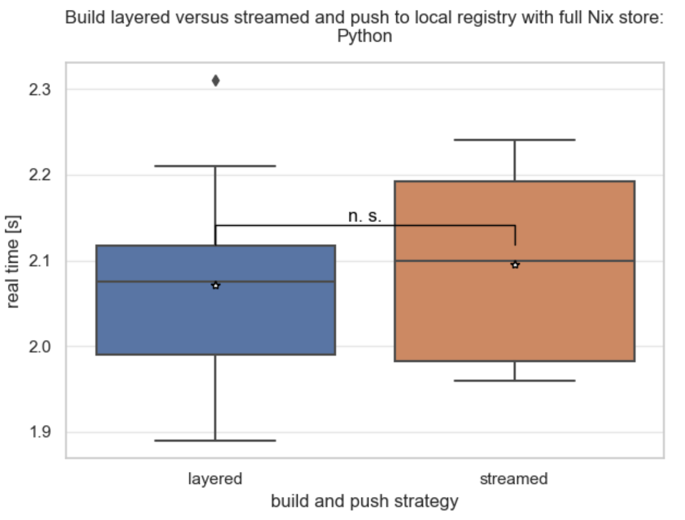


Figure 4.2: Comparing the build and push strategies streamed and layered using a *full* Nix store. Remarks: n.s. = statistically nonsignificant difference

The measured times are almost identical in the full Nix store case since both strategies use the store as a cache and skip building the image. In this case, the build time is dominated by the time (approx. 2s) it takes to push the layers to the registry. This push time means that even with caching the image built in the builder container, the current prototype's build time is still bounded by the push time. This influenced the prototype's design to skip the push step in 3.4.2. When the cache is empty, the build time is dominated by the building of the image, which takes approx. 120s, and not by the push time, which is only around 2s.

4.1.2 NSAR approach specific

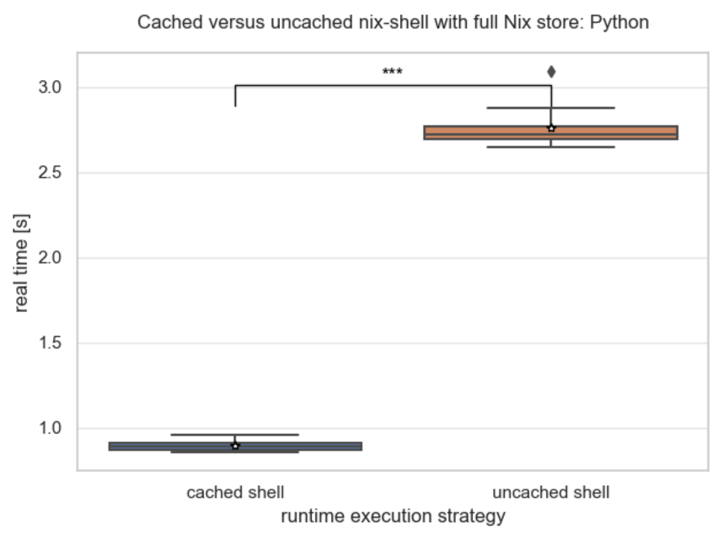
To determine the best internal implementation of the NSAR approach, we show our results of caching the nix-shell and using a seeded Nix store in the following paragraphs.

Caching nix-shell We want to test the performance of our idea of caching the nix-shell (see 3.5.3) with benchmarks. We restrict ourselves to the `C++` environment, as the results are similar to the `Python` environment.

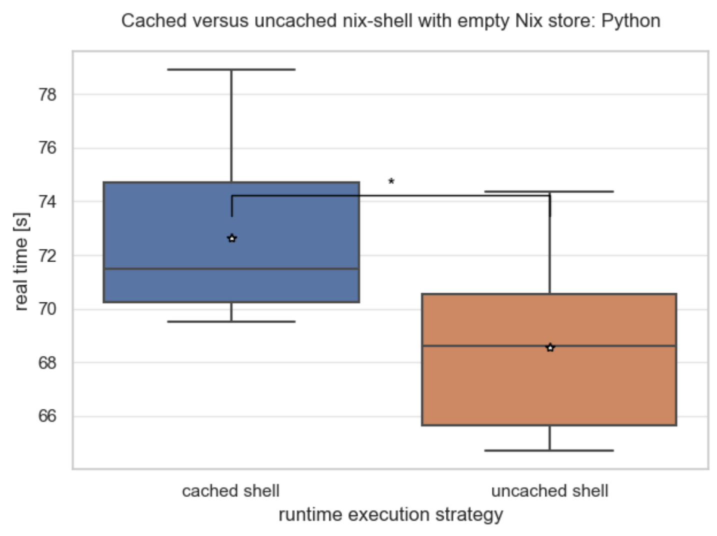
First, we assess the case where the Nix store is full (see fig. 4.3a), where the environment build-time with the cached nix-shell is almost three times faster than the uncached one. This is because the cached nix-shell can use the cached environment variables and skip the evaluation of the input files. Thus it is faster than the uncached nix-shell. When we assess first-time builds, where the Nix store is initially empty, we observe almost the opposite – the cached nix-shell is slower than the uncached nix-shell by approx. 20% (see fig. 4.3b). This is because the cache has to be initialized and updated by the cached nix-shell, which incurs some overhead.

Seeded store To determine whether the idea of using a seeded store (see 3.5.4) improves the performance, we compared it to an empty store and a full store. We prebuild the Nix store with the packages from the C_{base} file as these packages are known at build time and do not depend on the lecturer's modification. Additionally, we prebuild the Nix store with the packages specified in A.4 and use a configuration with some additional packages for benchmarking the build time. These additional packages should simulate a configuration modification, where some popular packages are in the seeded store, and the lecturer wants to install some more that are not yet available. We use a cached nix-shell for both comparisons and restrict ourselves to show the `C++` environment here.

When comparing the build time of an empty store with that of a seeded store, we can observe that the latter brings an order of magnitude (50x) improvement in performance (see fig. 4.4a).



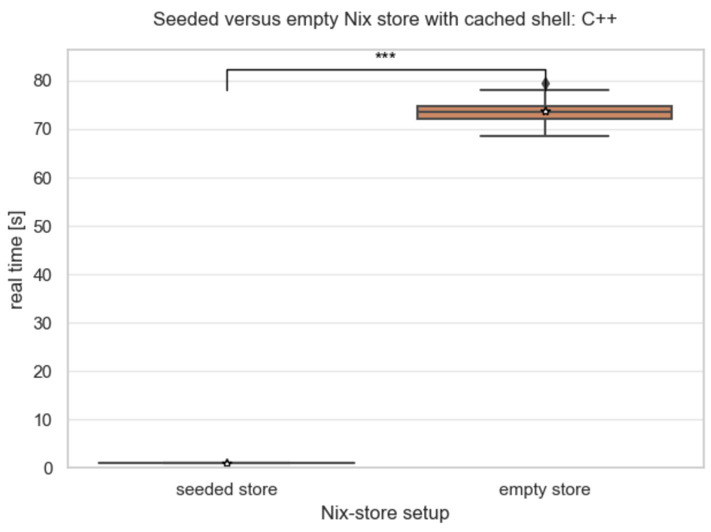
(a) Comparison using a full Nix store



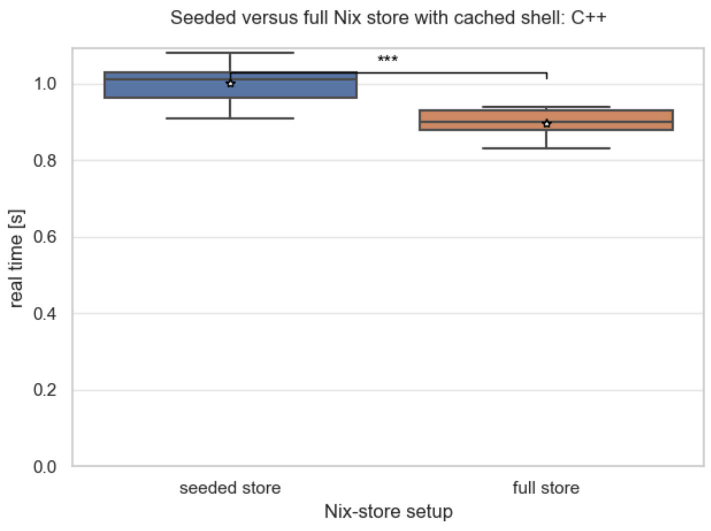
(b) Comparison using an empty Nix store

Figure 4.3: The "full store" is a measurement of a subsequent-time build that uses the cached results of a first-time build with the same configuration. The Nix store is empty for the first-time build. We show the Python environment results. Remarks: cached shell = cached nix-shell (see 3.5.3), uncached shell = standard nix-shell (see 2.6.4), * = a higher number of stars(*) indicates a greater statistically significant difference.

By inspecting figure 4.4b, we note that there is small (approx. 100ms) but signifficant difference when using a seeded store versus a full store. Interestingly when comparing both environments (fig. 4.4b and fig. A.1) we observe that the Python version has a smaller statistical difference (two stars)



(a)



(b)

Figure 4.4: The "seeded store" has a subset of the packages used to build the measured build time preinstalled. The "full store" is a measurement of a subsequent-time build that uses the cached results of a first-time build with the same configuration. We show the C++ environment results. Remarks: * = a higher number of stars(*) indicates a greater statistically significant difference.

than the `C++` one (three stars) even though we add 1 `Python` package and 5 OS-packages to the `Python` configuration compared to only 1 additional package in case of the `C++` configuration change (see A.4).

4.1.3 First-time build

The first-time build performance – starting from the new configuration until the container environment is "ready" – corresponds to the case where all build-caches are empty. This case happens, for example, when the configuration C_{result} specified by the lecturer is such that no previously cached outputs can be used. We made the measurement process of the approaches as comparable as possible by deciding what execution flow resembles the first-time build in the best way for each approach. This decision is made separately due to the different implementations. We explain how we measured the build time for each approach next.

In the BIAR prototype, we assume that the builder container image is available and the shared Nix store of the builder container is empty. We measure the following steps: start the builder container, then build and push the layered image built from C_{result} as a stream to the registry. Then pull the built image from the registry, and start an environment container from it.

We measure the build time of the NSAR by assuming that the shared Nix store of the environment container is empty and the minimal base image is cached locally on the container engine host. We measure the following step: start the environment container with the configuration and project files bind-mounted into the container and run the cached nix-shell to build the environment with the packages available.

In the current CodeExpert approach, the cxEnvrionment image extends the base image `base-rhel8`. We assume this base image is already built, as the lecturer configuration change does not influence it. We measure the following steps: build the cxEnvrionment image, then pull this image from the registry, and start a container from it. We use the `Python-3_8` and `GCC` images and compare them to our `Python` and `C++` environments, respectively. We did not measure the time it takes to push the image to the registry for this approach.

We can start an environment without network access for BIAR and the current approach, unlike the NSAR environments.

Prototypes compared with each other Depending on the environment, the NSAR prototype is two to four times faster than the BIAR approach (see figure 4.5). This considerable difference is due to the additional work BIAR does. Both prototypes need to download almost the same packages from the binary cache. However, the BIAR approach must additionally build an image, then push it to and pull it from the registry. Since the significant difference

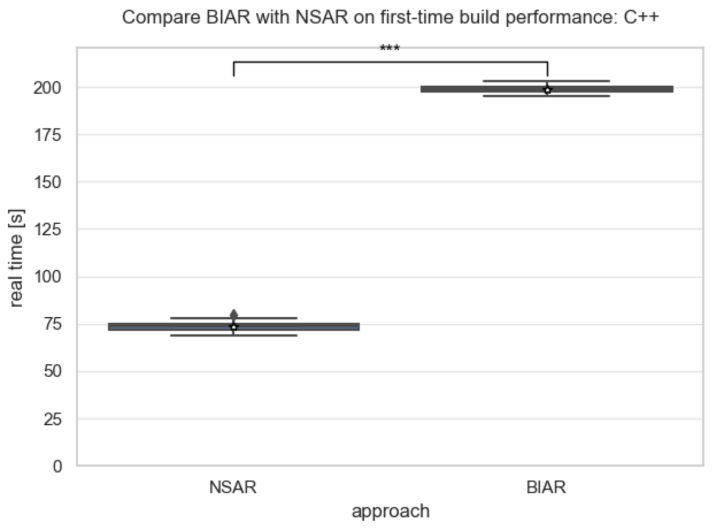


Figure 4.5: Remarks: `C++` = `C++` environment, NSAR = nix-shell at runtime, BIAR = build image at runtime, * = a higher number of stars(*) indicates a greater statistically significant difference.

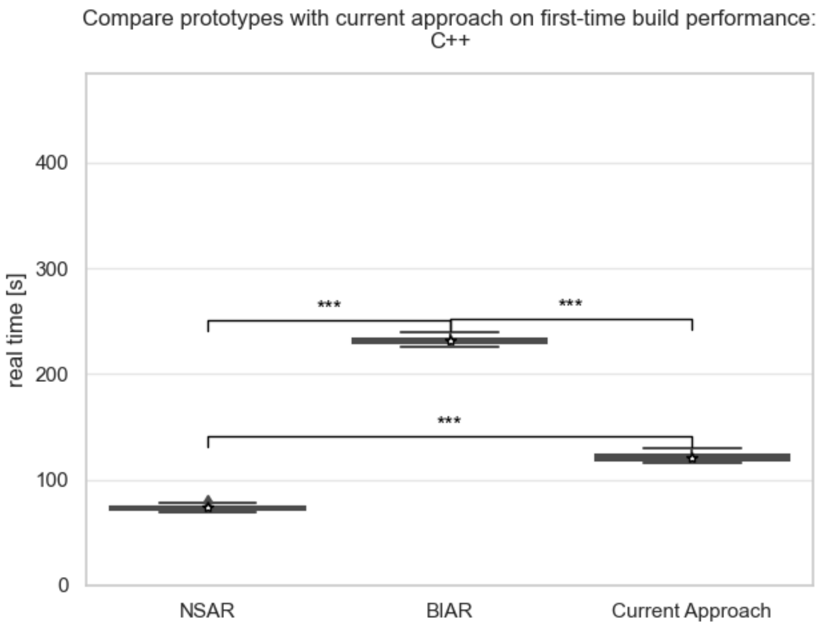
is large for both environments, we only show the `C++` environment's figure here.

Prototypes compared to current approach For the `C++` environment, the current approach is slower than NSAR but faster than BIAR (see figure 4.6a). BIAR and the current approach build the `C++` and GCC image, respectively, using only prebuilt binaries. However, the current approach performs slightly better than BIAR. This difference can have multiple reasons as they most importantly use different image build systems. Additionally, the push time to the registry is missing in the benchmark of the current approach, and the BIAR prototype includes an additional large `C++` package (see A.4) in its configuration.

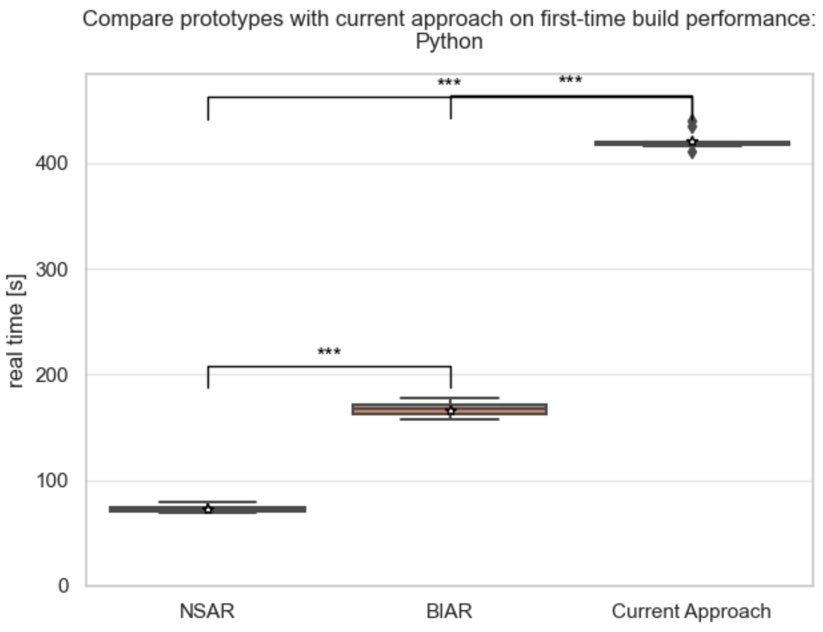
By inspecting figure 4.6b, we can note that building the `Python` environment with the current implementation is much slower than both BIAR and NSAR. This is because the current approach builds `Python` from source code, unlike our prototypes that use prebuilt binaries. Therefore we cannot compare the results for this environment.

4.1.4 Subsequent-time build

The subsequent-time build (startup) performance – start an environment container with a previously built configuration C_{result} – implies that the build-cache is full. The same argumentation for the network access holds as in the



(a) `C++` environment



(b) `Python` environment

Figure 4.6: First-time build performance of all approaches compared. Remarks: NSAR = nix-shell at runtime, BIAR = build image at runtime, * = a higher number of stars(*) indicates a greater statistically significant difference.

first-time build (see 4.1.3), and we explain how we measured the build time for each approach next.

In the BIAR approach, we assume that the environment image is available on the local container engine host since it has been previously built. The only step we need to do is start the right image inside a container. We find the correct image to start from the given C_{result} by computing the hash over the configuration files before starting the container.

The NSAR approach assumes that the store of the environment image has all derivations of the configuration installed. The execution flow follows: start a container from the environment image and run the cached nix-shell, which can instantly start the environment shell.

For the current approach, we assume that the environment image is available on the local container engine host. We start the image as a container, and the environment is ready.

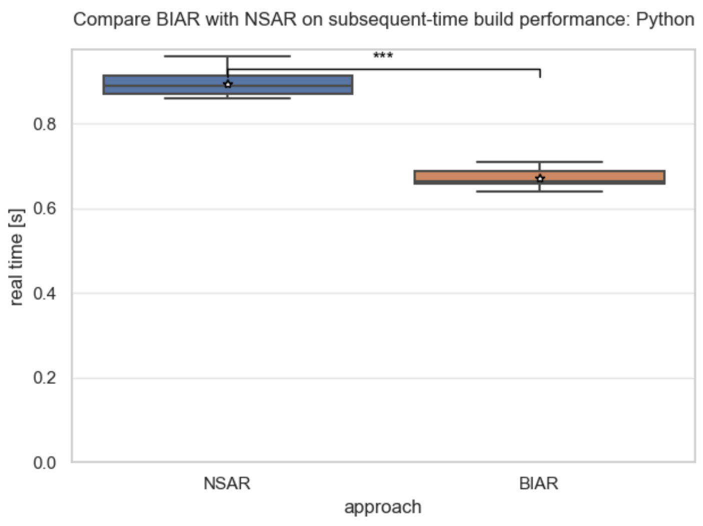


Figure 4.7: Comparing the measurements of our prototypes' subsequent-time environment startup performance using the Python environment. Remarks: NSAR = nix-shell at runtime, BIAR = build image at runtime, * = a higher number of stars(*) indicates a greater statistically significant difference.

Prototypes compared with each other We can see in figure 4.7 that BIAR is significantly faster than NSAR. BIAR takes between 0.65 and 0.7 seconds, whereas the NSAR startup time lies in the time range of 0.85 to 0.95 seconds. Since both have similar results, we can restrict ourselves to the `Python` environment. This difference is incurred mainly because the NSAR prototype must check if the cache can be reused for the nix-shell execution and bind mounts two additional directories into the container. It also needs network access, which adds overhead as it requires Docker to add network configuration at the environment startup.

Prototypes compared to current implementation The current approach has a startup time of around 0.6 seconds which is significantly faster than both prototypes (around 300ms faster than NSAR and 100ms faster than BIAR). This result can be observed in figure 4.8 and is analogous to the `Python` environment. We already provided some likely causes why NSAR is slower than BIAR. One reason for the difference between BIAR and the current approach might be the former's MD5 hash value computation over the configuration before starting the environment.

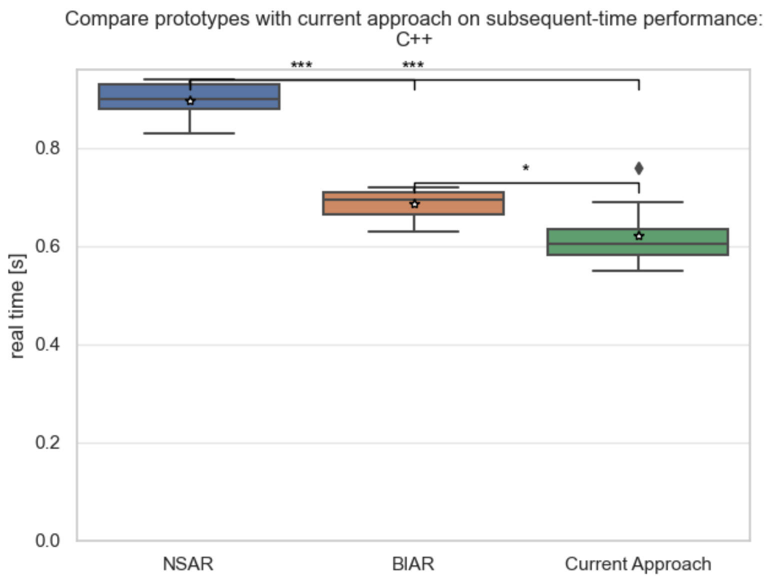


Figure 4.8: Comparing the measurements of our prototypes' subsequent-time environment startup performance to the current approach using the `C++` environment. Remarks: NSAR = nix-shell at runtime, BIAR = build image at runtime, * = a higher number of stars(*) indicates a greater statistically significant difference.

4.2 Security

The security of an environment is essential as we want to restrict users from breaking out of their isolated container to prevent attacks on the host system or other users' environments. At the same time, we want to provide a controlled and isolated environment to each user for remote code execution because we are in the context of a cloud IDE.

We assume in this section that beyond the prototype-specific implementation, an environment is implemented such that exploits are made as hard as possible (e.g., remove privileges and access to the kernel). Under this assumption, the current approach is less likely to have security vulnerabilities.

We discuss the two main attack vectors, the shared cache and network access for both prototypes, and how they could lead to security vulnerabilities. We outline ideas for preventing these vulnerabilities with potential implementations in the further work (see 6.2.5 and 6.2.4).

The BIAR prototype is generally vulnerable to attacks from the lecturers as the builder container has both attack vectors. In contrast, the NSAR prototype is vulnerable to attacks from any user since every user interacts with the environments with both attack vectors.

The environments produced by the BIAR prototype have no security vulnerabilities inherent to the implementation. Other vulnerabilities may exist that are not a direct result of our implementation – for example, insecure packages. We can remove access to the network for each environment, and there is no shared component between containers.

4.2.1 Shared cache attacks

We divide this section of cross-cache attacks into the two types of shared caches: the Nix store and the nix-shell cache. The latter is only applicable to the NSAR prototype.

Cross-cache attacks on the Nix store The Nix store cache is vulnerable to denial-of-service (DoS) attacks. For example, a malicious user could fill the cache with garbage derivations until the VM runs out of memory.

In addition, the shared Nix store is possibly vulnerable to cache poisoning. A user can put invalid entries (derivations) into the cache, which another environment container could access and deem valid. For example, a malicious user could construct a derivation that has as directory name the hash of a popular package p and put it inside the store. If a configuration specifies this package p in some following build process, then Nix will use the faulty cached derivation from the shared Nix store.

In our prototypes, the Nix database is also shared. It keeps track of the mapping between dependencies and derivations in the Nix store and can be arbitrarily modified to remap dependencies between derivations. One result is that the database becomes out of sync with the Nix store. Thus the database needs to be rebuilt by Nix if it cannot use the mappings from the database, which degrades performance and could even result in DoS.

Possibly, the lecturer could introduce unintended security vulnerabilities to an environment, for example, by misconfiguring a package or using a malicious package. It is generally the responsibility of the lecturer to configure an image correctly, but these unintended attack vectors could compromise the Nix store. Nix employs defaults that prevent packages' installation from the Nixpkgs collection based on the package metadata. Nix will not install a package if its meta value indicates that it is broken, does not run on the given system, or has known security vulnerabilities.

The multi-user feature (see 2.6.3) makes some security vulnerabilities harder to exploit (e.g., cache poisoning). This is because the root owns the Nix store and database, and builds are performed by a daemon, preventing arbitrary builds. Thus a user may need root permissions to access and change part of the Nix store and database. However, multi-user mode does not help with other attacks like DoS on the Nix store and does not improve the vulnerable nix-shell cache. The reason for this is that this feature is devised to share the Nix store across multiple user accounts to prevent cross-user attacks, but not share the Nix store across environment containers that all have the same user.

Cross-cache attacks on nix-shell cache Environments in the NSAR prototype share two caches, one for the Nix store and the other for the nix-shell. The NSAR approach needs both caches to achieve fast startup times for subsequent environment builds.

The same security vulnerabilities as for the shared Nix store potentially exist for the nix-shell cache, which currently has no barriers against cross-cache attacks. This is because the files inside the nix-shell cache are writable by the user, unlike the files in the Nix store, which are read-only. Thus a user could quickly delete the cached files, which delays other environments' startup time, or change the files in arbitrary ways. An example of the latter is changing the symlink of the `nix-shell` 's derivation so that it points to another derivation in the nix store.

4.2.2 Network access attacks

Each builder container of BIAR or environment of NSAR has access to the public network, mainly to provide the nix-shell access to the binary cache (cache.nixos.org) to download prebuilt packages not already in the Nix store. Another reason to provide network access would be to allow the use of "fetcher", which Nix provides to fetch source code from repositories like Github, Gitlab, or PyPI [29]. These fetchers are needed if users want to, for example, build specific versions of packages from source code that are not available in the Nixpkgs collection.

Unrestricted network access is problematic as it would allow a user to potentially fetch arbitrary sources such as code repositories or web pages. This broad access would make it significantly easier to execute malicious code remotely to attack the cloud system or, for example, use the computing resources for adverse purposes. Another thing to remark is that in the particular use case of CodeExpert, students are not allowed to access the network, for example, during programming exams. This case is not a security vulnerability per se but a requirement that must be satisfiable for our prototype to be used in this context.

4.3 User experience

To evaluate if our prototypes boost the flexibility in coding exercises, we evaluate the user experience compared to the current solution. The lecturer must set up the coding environments first such that students can run coding exercises in the environments. Therefore this section covers the process, flexibility, and wait time for setting up a configuration. Additionally, it looks at students' interaction response time for environment startups.

We define a "good" user experience in our context with the following properties that are influenced by the "user experience" (UX) concept described in [12].

1. Ease of writing configurations
2. Flexibility of setting up environments with any language and packages
3. Efficient change of configurations, which includes the build time of the configuration
4. Quick environment startup times for subsequent environment startups

4.3.1 Configuration

We first cover the identical arguments for our prototypes because both use Nix as a build system. With the BIAR and NSAR prototypes, the lecturers can directly specify all environment packages themselves (without interacting with the developer) by changing one of the configurations of C_{preset}. Such a configuration has a fixed interface (see 3.3) for increasing the ease of use when writing configurations. Changing the configuration for lecturers that only specify packages from the Nixpkgs collection is trivial, as the lecturers do not need to learn the Nix language. To showcase the ease of specifying and changing the configuration, one can look at the preset-python.nix configuration and compare it to the changed-python.nix configuration. The preset-python.nix is a reduced example from C_{preset}. One Python package and two other packages have been added as highlighted in changed-python.nix.

preset-python.nix

```
{ pkgs }: with pkgs;
let
  pythonEnv = python39.withPackages (p: with p; [
    numpy
  ]);
in
{ inputs = [ pythonEnv ]; }
```

changed-python.nix

```
{ pkgs }: with pkgs;
let
  pythonEnv = python39.withPackages (p: with p; [
    numpy
    networkx
  ]);
in
{ inputs = [ pythonEnv openssl sqlite ]; }
```

If the lecturer wants to specify a custom package or package version, then he needs to learn some details of the Nix language for declaring the package as a derivation. Specifying a custom derivation can be challenging, but there are many examples in the Nix documentation (e.g., [38]) which should provide ample resources for this task. With some packages and languages, the lecturer may need to figure out which environment variables need to be set at runtime, e.g., such that the compiler finds the shared libraries for linking. This can be challenging, but there are many examples of typical use cases in the Nix documentation, e.g., in [38]. The NSAR prototype makes the specification and restriction of global environment variables inside the shell easy and has a `shellHook` attribute in the configuration interface for this purpose. This specification and restriction of environment variables may be more challenging with the BIAR prototype as environments do not have the `nix-shell` functionality. Special attention is needed when configuring interpreted language using our prototypes. Nix wraps one interpreter and other executables together such that they can find each other and all of the modules. Thus Nix only allows one of these wrapped environments to be installed globally since they otherwise conflict with the interpreter to load out of the PATH environment variable [38].

In the current solution, the lecturer cannot deploy the configuration changes directly and must contact the developer to deploy every (minor) change in the configuration. This process can sometimes be advantageous for the lecturer since the developer is responsible for making the images run correctly in production. On the flip side, it may also happen that the developer cannot make a (small) configuration change requested by the lecturer as it may be, e.g., too tricky. If the lecturer does not want to use a predefined configuration, he can write his custom configuration or extend a predefined configuration. This requires knowledge of writing a `Dockerfile` [11]. Unlike our prototypes which allow the composition of configurations, in most cases, the lecturer cannot easily compose two different configurations written as `Dockerfile`'s into one image in the current solution as argued in 3.1.1.

4.3.2 Flexibility

The current solution restricts the lecturers to packages and specific versions that are available in the used Linux distribution, such that it may happen that a package requested by the lecturer is not available for installation. With our prototypes, lecturers can choose from over 60'000 predefined packages from the Nixpkgs collection or declare any custom package or version they need. This is because Nix can build a package from source code by fetching this code from remote code repositories. The result is almost unlimited flexibility for configuring the environment.

4.3.3 Wait time for user interactions

This section discusses the time the user must wait until different interactions (e.g., changing the configuration) are complete with our environments.

The faster the build time, the quicker the lecturer can iterate over configurations during development and when changing the configuration, which improves the UX. Therefore we use the results from 4.1.3 to evaluate the efficiency of configuration changes. The build time for BIAR is approx. between 2 and 4 min, which is slower than NSAR, which takes around 1min for a complete build with empty caches. Note that after some packages have been built, the build caches are unlikely to be empty, and the build times tend to be faster. Furthermore, both of our approaches allow us to significantly optimize the first-time build performance with a seeded store in the case of NSAR (see 4.1.2) and in the case of BIAR see 6.2.3.

The current solution has a bit faster build time than BIAR but is slower than NSAR. However, with the current solution, the lecturer only reaches these build times when testing the environment on a local machine. In contrast, our approaches allow direct testing on production systems after the above build time. To test a configuration change with the current approach in the cloud system, the lecturers must contact the developer. Then he must wait potentially many days until the developer has deployed the corresponding image to use it.

The other primary user interaction with our environment that incurs wait time is when a student wants to run coding exercises. In this case, the first student who starts an environment from a new configuration must wait longer, whereas all following students experience the subsequent startup time. Since students invoke an environment startup many times for each job execution, the faster the subsequent startup time, the better the UX. Note that the current solution is faster than BIAR, which is faster than NSAR (see 4.1.4).

To summarize the UX of the lecturer, our approaches have advantages in configuration, flexibility, and build time over the current solution. On the other hand, the current solution has the edge in the subsequent startup performance for students.

4.4 Developer experience

In this section, we evaluate the developer's experience in maintaining and developing the implementation of our prototypes compared to the current approach.

We define a "good" developer experience for our problem with the following properties influenced by CodeExpert developer's feedback and the developer experience defined in [12].

1. Minimal workload from configuration changes by the lecturer
2. Less responsibility for usage defects
3. Ease of development and testing
4. Minimal maintenance of implementation (e.g., images and dependencies)
5. Increase robustness of the cloud-based system.

4.4.1 Developer involvement with configuration changes

Ideally, the developer does not need to be involved whenever a lecturer submits a new or changed configuration for deployment on the production systems. This goal minimizes the developer's workload and is fulfilled with our prototypes. This stands in contrast to the current approach, where the developer needs to be in direct contact with the lecturer and perform the following manual steps

- check if the requested packages are available for installation,
- figure out how these packages need to be installed,
- if they can be installed without much difficulty, then build and then test the image,
- and if all the above steps are successful, deploy it to the image registry.

Another critical point is that our approaches shift the responsibility for setting up a correct and secure image (e.g., not using broken or vulnerable packages) from the developer to the lecturer. This argument follows from the above argumentation, where we outline the impact of a configuration change on a developer.

4.4.2 Development and testing

With our prototypes, a developer can easily add, e.g., a new language as a configuration to the C_{preset} group by writing a new template as a Nix expression. This template can then be used or customized by a lecturer. At the moment, adding a new predefined configuration means writing a custom `Dockerfile` and then manually building it and deploying it to the production system, which can be cumbersome.

The developer needs to learn how Nix works and its domain-specific programming language to implement our prototypes, develop them further and test them. Learning Nix can be challenging since it is a purely functional programming language with limited documentation and usage examples. Nix also differs from other functional programming languages such as Haskell. It is often unclear what the “standard” Nix approach for a common problem is. However, it is generally advisable to use and compose the many predefined helper functions from the standard library. To aid with development, Nix provides a repl tool that is useful for trying out small Nix code examples and the nix-shell that allows for incremental builds and debugging inside the build-environment (see 2.6.4).

Nix provides tools that make debugging more manageable: the nix-shell sets up the build environment that can be debugged and inspected and from which the developer can build the package or image manually. Testing container images is best done in a continuous integration (CI) pipeline by, for example, building the reproducible images and then running tests specified, e.g., with a bash script. The primary tool to aid with CI and testing in the Nix ecosystem is Hydra (see [30]), which can cache CI builds and make subsequent pipeline runs faster. Furthermore, testing one or two base images of our approaches that rarely change is likely less of a burden to the developer than testing more than a dozen, each with different languages, in the current approach.

4.4.3 Maintenance

For the NSAR prototype, the developer only needs to maintain a single image, i.e., built with Nix. However, the Nix store needs maintenance as the developer needs to run the garbage collection or store path optimization regularly to avoid running out of disk space. When the developer runs the garbage collection tool of the Nix store, it is essential not to remove potential configuration files (e.g., shell scripts) stored in the Nix store. These files are needed for the container at runtime. Additionally, the nix-shell cache could need maintenance for the NSAR prototype.

The NSAR prototype has the `cached-nix-shell` package as a dependency, a Nix package developed by the community that needs to be updated by the developer and might break after an update. Adding features to this package or making changes might make maintaining this dependency a burden. However, our prototypes have much fewer dependencies to maintain than the current approach, which has many packages in multiple image configurations with different package managers such as pip and DNF. The fewer dependencies our prototypes have on external packages, the better the developer experience.

The BIAR prototype has two images that the developer needs to maintain and a shared Nix store, where the same arguments apply as described for the NSAR prototype. Because the BIAR prototype builds an image for every unique configuration, the image registry likely becomes overloaded with many images. Therefore it is necessary to remove images from the registry regularly. It would be best to remove the least used images to prevent unnecessary rebuilds, but this would require collecting image usage statistics.

Our prototypes use a specific version of Nixpkgs and Nix, which needs to be updated in the future and would require a rebuild of the images. For the current prototypes, updating the Nix version would imply updating the Nix base image from Docker Hub. Updating the Nixpkgs version is handled using the Nix package `niv`.

It is crucial for an approach's development and robustness that the same build inputs lead to the same output in future builds. The builds of both prototypes produce the same output because they are reproducible, except if the developer changes the configuration or script files that are copied into the image or bind-mounted into the environment. On the other hand, the current approach is not reproducible since the image builds can fail (see 3.1.1).

Chapter 5

Discussion

After presenting the results in the last chapter, this chapter will interpret the results to propose the best approach for solving our problem.

5.1 Performance results

This section discusses the performance criteria results presented in the last chapter. The measurement process we used for the performance results produces relevant results for our problem. The reason is that we benchmark with a "warmup"-phase on a virtual machine architecture (see 3.6) to simulate the conditions of a running cloud-based IDE. To compare the performance of different approaches, we tried to measure the execution beginning from identical start-points and ending at the same result (outlined in 4.1) and with about the same packages installed (see A.4) to get comparable results.

We can summarize from the BIAR approach-specific benchmarks results that saving I/O operations and disk/cache space tends to pay off in the first-time build performance for the streamed build strategy. According to the Nix documentation, we assumed that the streamed strategy saves I/O operations and disk/cache space. We could make a stronger argument if we verified this assumption specifically with benchmarks. As expected, the subsequent-time build performance did not result in statistically significant differences between both strategies since Nix uses the cached layers and does not rebuild for both strategies. Therefore we should opt for the streamed strategy implementation.

For the NSAR approach, we can conclude that the results matched our expectations. This implies that we should use a cached nix-shell, and we could use a seeded Nix store from a performance standpoint to improve the first-time build. The cached nix-shell is slower for the first-time build performance than the uncached nix-shell, but this is not problematic since we can be slower in this case. The cached nix-shell allows us to achieve an up to three times faster subsequent-time startup compared to not using caching. This result is somewhat similar to the very limited data (of an unknown measurement process) of [51] where a slighter difference was measured between the cached and uncached nix-shell.

The results of the seeded store evaluation also indicate that adding a few packages, which are available in the binary cache, to a previously built configuration results in a small overhead compared to the first-time build. This result infers that Nix can efficiently cache previous builds in the Nix store and also transfers to the BIAR prototype since Nix efficiently caches both the nix-shell execution and the resulting layers of an image build as derivations in the Nix store (see 2.5.2, 3.4.4).

Recall that in 4.1.2, we measured a more negligible overhead when building six additional packages in the Python environment that were not in the seeded store than building one package in the `C++` configuration change. This result is unexpected since we predicted the opposite. The reason for this result could be that the dependencies of the additional six packages have a large intersection with the dependencies of packages already in the seeded store. This would confirm our expectation outlined in 3.4.2 and aligns with the argumentation of [10]. However, we can not make a decisive argument since the packages added to each environment are different and, obviously, the environments themselves (see 3.6.4).

To summarize, even though the seeded Nix store improves the first-time build, it does not provide a strong enough argument to justify its implementation for our problem. This is because, after a first-time build, the resulting Nix store is comparable to a seeded store from the viewpoint of subsequent builds with small configuration changes. Our argument makes sense for our problem since suppose a running cloud-based IDE in which we provide configuration templates that most lecturers will not change or only slightly. Then after the first build, which produces a "seeded store", all other lecturers' configuration changes will only experience a small overhead.

This discussion on the NSAR implementation has implications: caching the nix-shell is required to achieve fast environment startups for subsequent-time invocations, and implementing a seeded Nix store is unnecessary. The former implies that we must implement the unsafe shared nix-shell evaluation cache (see 4.2.1).

The first-time build performance of the NSAR prototype is the fastest among all three approaches (see 4.1.3) and even can be optimized with a seeded store, as argued before. The prototype of the BIAR approach has a comparable first-time build performance to the current approach, as argued in the `C++` environment results of 4.1.3. Unlike the current approach, the BIAR and the NSAR prototypes automatically optimize the build time for small configuration changes, but in the case of BIAR, this optimization is currently limited by the push-time as discussed in 4.1.1.

The results of the subsequent-time build performance were as expected. The overhead measured for checking the nix-shell cache (300ms) in the NSAR prototype align with the, again, very restricted result (290ms) from [51]. This overhead is roughly three times the time it takes to compute the hash value over the configuration in the BIAR prototype. This hash value computation might also be the difference in the subsequent startup performance between the BIAR prototype and the current approach since they both start a container without a network and without checking cached results like the NSAR prototype. This means that it is essential for our problem to optimize the cache verification in the NSAR prototype and the hash value computation in the BIAR implementation. The cache verification is challenging to optimize as it is already optimized and an external package outside our prototype implementation.

In our benchmarks, we used two test environments: a `C++` – to represent compiled languages – and a Python one – to represent interpreted languages. We wanted to evaluate whether one approach is more suited for one environment than the other or if they perform equally well regardless of the environment. Our results from 4.1.3 and 4.1.4 imply that no approach is more suited to one type of environment over the other.

5.2 Security results

The shared cache attack vector results showed that the NSAR prototype has more security vulnerabilities than the BIAR prototype for two main reasons. The first, apart from the shared Nix store, is the additional cache for part of the nix-shell execution. This cache has no barriers against cross-cache attacks, resulting in security vulnerabilities or performance degradation for other users. The second is due to a reasonable assumption in our context that the lecturers have no intention to maliciously attack the system since they configure the environments for teaching purposes. This assumption holds for the network attack vector as well.

Another point is that the shared cache security vulnerability could easily be removed from the BIAR approach by just removing the shared Nix store from the builder container without impacting the subsequent-time build performance. The implication would be that the build performance for configuration changes would equal the first-time build with empty build caches. This quick-fix would tradeoff performance for security. This stands in contrast to the NSAR prototype, where the shared Nix store and shared nix-shell cache is required to achieve the essential quick subsequent time environment startup. Therefore the BIAR prototype has even more advantages from a security standpoint than the NSAR prototype.

As argued in 4.2.1, installing Nix in multi-user mode would improve the security of the shared Nix store to a small extent. This would, however, come at the cost of needing to run the Nix daemon inside each environment and configuring custom `systemd` initialization mechanisms [32]. The former could result in performance degradation and the latter in poor developer experience. Therefore we conclude that the implementation tradeoffs prevail over the security gains for implementing the multi-user mode.

To summarize, the security vulnerabilities of both prototypes have their cause in which part of the prototype's architecture the Nix build system is installed. Since the resulting environments of the BIAR prototype and obviously of the current approach do not have Nix installed, the environments do not have a shared cache nor network access and therefore have none of the two attack vectors. This contrasts with the NSAR approach, where the environment needs to have Nix installed.

5.3 User and developer experience results

The results are based on our analysis of the prototypes and current approaches implementations based on the enumerated properties of what we define as a "good" user or developer experience. For the developer and user experience results, we do not have results from previous studies for comparison. This is because of our particular use cases in the context of a cloud-based IDE where we have developers on one side and lecturers and students on the other. Each side has individual requirements on the system and challenges with the current approach.

5.3.1 User experience

From the results of 4.3.1, it seems that our prototypes have a slight advantage. However, we cannot decisively conclude if they provide an improved UX over the current approach regarding the ease of writing configurations. Indeed they have the edge for composing configurations and for simple configuration changes, considering the challenge of writing a custom `Dockerfile` for the current approach. However, whether the advantages outweigh the fact that the lecturer needs to take on more responsibility with our prototypes is unclear. Furthermore, some details – e.g., figuring out the correct environment variables – of configuring custom configurations with Nix could be more or less challenging. To make a more decisive argument, testing the configuration process of our prototypes with lecturer feedback would be beneficial.

Regarding boosting the flexibility of configuration, our prototypes clearly have the edge over the current approach. Users can efficiently change the configuration with our prototypes and test the effects on the production systems within a few minutes instead of possibly multiple days for the current approach. Since the subsequent startup time of the current approach is slightly faster than BIAR and much faster than NSAR, the UX for executing code in environments tends to be better for the current approach. To make a more decisive argument in this case, we would need to test our prototypes with student feedback to find out whether, e.g., the small startup overhead of the BIAR prototype compared to the current approach impacts the user experience poorly.

5.3.2 Developer experience

The developer's workload from configuration changes by the lecturer is undoubtedly minimized for our prototypes as opposed to the current approach, where configuration changes are likely to be very demanding. Our prototypes also have the advantage over the current approach as the developer has less responsibility for usage defects if the lecturer does not set up the environment correctly. Thus our approaches have a much better developer experience for both points.

The further development and testing of our prototypes can be challenging initially. However, it is hard to make a powerful argument on how much the tools provided by Nix for development and testing alleviate the ease of implementing and developing our prototypes compared to the current approach in the long run. Feedback from developers who worked with the prototypes would help make a more solid argument for this aspect.

Our prototypes have one or two base images and few dependencies to maintain, resulting in a better developer experience than the current approach. The advantage of our prototypes might shrink from maintaining the Nixpkgs and Nix versions, the shared Nix store, the nix-shell cache (for NSAR), and the image registry (for BIAR). Maintaining the nix-shell cache and additional dependency (`cached-nix-shell`) of the NSAR prototype likely overweighs the burden of the BIAR prototype's additional image and image registry. Our prototypes offer more robustness and are future-proof as they are reproducible.

Chapter 6

Conclusion

In this thesis, we found two feasible approaches to building images at runtime based on the lecturer configuration, described the implementation of the prototypes for each solution approach, and evaluated our prototypes and the current approach to arrive at a comparison among the evaluation results.

6.1 Proposition

The final aim of this thesis is to propose the best approach based on the results and discussion. To this end, we summarized the quantization results of each evaluation criteria in table 6.1. Each cell in the table is a real number from the interval [0, 1], where higher values represent a more relevant result for a criterion. We describe how we reach these points in A.1.

Evaluation criteria or approach	**Performance**	**Security**	**UX**	**DevX**	**Total**
NSAR approach	0.45	0	0.75	0.9	2.1
BIAR approach	0.6	0.8	0.75	0.9	3.05
Current approach	0.75	1	0.35	0.1	2.2

Table 6.1: Approaches are compared with each other based on the results of A.1. Note that the BIAR approach has the highest point total. Remarks: UX = user experience, DevX = developer experience

To conclude, we propose the BIAR approach, as it has the highest point total and is more suited for our problem than the NSAR approach, which, although radical and lightweight, is unacceptable from a security standpoint. The BIAR approach has the potential to alleviate many tradeoffs of the current approach, e.g., by improving the flexibility of the users and the developer experience.

The BIAR approach could be used as a basis to implement the next version of the cxEnvironments for CodeExpert. Due to the design or our approaches that respect the current architecture and execution flow, the proposed approach can potentially be made available to lecturers in parallel to the current approach, such that existing cxEnvironments will continue to run, and newly created environments will use the latest advantages of the proposed approach.

6.2 Further work

After having answered our problem statement by proposing a prototype that solves our objective, we turn to describe future work in this section.

6.2.1 Testing prototypes with user feedback

As we have argued in some parts of 5.3, testing our prototypes with user feedback is important. These tests could, for example, include:

- How difficult it is for lecturers (knowledgable or not) to configure environments to test the configuration experience.
- What configuration parameters are needed such that the lecturer can configure environments (to determine the configuration interface of our prototypes).
- What is the impact on the UX for students caused by the potential additional startup time of our prototypes compared to the current approach.

6.2.2 Custom package collection

It would make sense to add a custom Nixpkgs extension (see 2.6.2) for CodeExpert, such that, for example, the developer and lecturers can share custom tools that every lecturer can use in configurations. This would allow the lecturer to improve coding exercises' debugging or testing functionalities for students. Another use case is if developers and lecturers need to override the default configuration for some package, e.g., to fix a package bug or add environment variables needed for our context.

This private package collection can be achieved using *overlays* i.e., a method to change and extend Nixpkgs [35]. The overlay could be built regularly in a CI pipeline using the Nix CI tools [30]. Then the builder container or environments for the BIAR and NSAR prototypes, respectively, could use this extended package collection by just adding it as a channel.

6.2.3 Optimize build and push time for BIAR approach

The optimization of the subsequent-time build performance of the current BIAR prototype is limited by the push time as seen in 4.1.1. To overcome this limitation, we could try to avoid pushing layers that are already available on the image registry. This idea makes sense since images can share common layers according to the OCI image specification. Furthermore, the shared dependencies of packages likely result in their own layer when building images with Nix (see 3.4.4). Nix can efficiently cache these layers (see 5.1). A Nix library exists as a package (see [2]), which is a promising solution to our problem. The limited benchmarks outlined in [2] provide some intuition on the order of magnitude the build and push time could improve.

6.2.4 Improve security of shared cache

To improve the security of the shared Nix store in the BIAR and, more importantly, the NSAR prototype, we present one possible implementation idea.

We can use an overlay filesystem (e.g., `OverlayFS` [8]), also called union filesystem, similar to the way Docker shares layers between images (see 2.2.3). A read-only snapshot of the shared Nix store cache is mounted into the container (i.e., at `/.nix-store-cache`). We then overlay-mount this cache (using the overlay filesystem) into `/nix/store` where Nix expects the cache to be with a read-only snapshot of `/.nix-store-cache` as the "lower" directory and a user-writable directory (e.g., `/nix/store/upper`) as "upper" directory. Now we have a similar result to 2.2.3 and the user cannot modify the shared Nix store to attack the host system or other environments because each container has a copy-on-write view of the cache. This means that if the user modifies the cache by, e.g., performing builds, the accessed read-only files get copied to the upper directory, which is deleted when the container gets removed.

Before we take a snapshot of the shared cache, it is essential to preinstall the most popular packages analogous to the seeded Nix store. Then we can update the cache frequently with the most popular packages to keep the cache up to date. Determining the most popular packages would require gathering usage statistics, e.g., tracking which packages have been downloaded from the binary cache.

6.2.5 Restrict network access and improve security

We present two ideas that can be used to circumvent the need to have a container with open access to the network, which results in security vulnerabilities outlined in 4.2.2.

Private binary cache One can set up a private binary cache inside a local container and access it over the local network [27]. Apart from removing the need to access the public network, it would speed up the installation of packages by reducing the network roundtrip time. This private binary cache would require preinstalling most packages of the public one and frequent updates to keep the local cache up to date, which needs to be scheduled by a developer. A better solution that automatically takes care of the last two requirements would be to set up a binary cache inside the local network as a proxy of the public cache. This proxy fetches first-time accessed packages from the public cache and saves them for subsequent accesses on the local disk [15].

Network traffic filter Another idea would be to restrict each container's access to specific public IP address ranges, which are used, for example, by the public Nix binary cache CDN. To this end, we can filter the egress network traffic initiated from inside the container using a proxy HTTP-server connected to each container and restricting its outgoing requests. This proxy server would allow fine-grained control over which repositories and binary caches can be accessed, but the developer must configure them manually.

Remove network access from environments Suppose no network connection is allowed for an environment. In that case, we can use the first idea of a local binary cache proxy to solve the problem of needing access to the public binary cache. If we want to allow fetchers for building custom packages, we could either make sure that these packages are cached in the Nix store from previous builds, or we could set up a custom package collection (see 6.2.2). If the lecturer wants to use fetchers, he must submit this package (i.e., as a Nix expression) to this package collection. The packages inside the collection must then be prebuilt before an environment can use them.

Appendix A

Appendix

A.1 Conclusion points assignment

We outline in this section how we quantify the comparison of our approaches by assigning points to each approach for the qualitative results of the evaluation criteria. The points are real numbers of the interval [0, 1], where the highest possible point for each criterion is 1, and the lowest is 0.

A.1.1 Peformance

To quantize the performance discussion, we use weighted points listed in table A.1 for first-time and subsequent-time performance categories. Since the maximum is 1, the weights are chosen such that this maximum is attainable but cannot be exceeded. The best approach in a category gets the points in the first row, while the second and third-row looks at the relative difference to the best approach. What a "small" or "large" difference means is not rigorous and different for each category, i.e., a small difference for the subsequent-time build can be an order of magnitude lower than a slight difference for the first-time build. This is because the subsequent time build needs to be fast, and we penalize a difference to the best approach more.

The approaches have the following point sums (in order of the categories in table A.1):

- NSAR: $0.35 + 0.1 = 0.45$
- BIAR: $0.25 + 0.35 = 0.6$
- Current approach: $0.1 + 0.65 = 0.75$

The assignment of each point is justified in the results and discussion sections (5.1 and 4.1).

Category or relative comparison	First-time build	Subsequent-time build
Best approach	0.35	0.65
Small difference to best approach	0.25	0.35
Large difference to best approach	0.1	0.1

Table A.1: For each category, we assign points (real numbers in [0, 1]) to an approach based on the relative comparison to the other approaches. There is only one "best" approach, and the others either have a "small" or "large" difference from this best approach.

A.1.2 Security

We quantize the security criteria with a penalty that gets subtracted from 1 if the approach is vulnerable to a given attack target from a user group. We penalize attacks from students more than from lecturers (see table A.2), for reasons discussed in 5.2. The weights are chosen so that each approach's worst or best assignment corresponds to the total maximum and minimum attainable points. From the discussion of 5.2, we conclude that the NSAR prototype is vulnerable to all attack targets and users and therefore gets 0 points. The BIAR prototype is only vulnerable to attacks from lecturers and therefore gets 0.8 points, whereas the current approach is safe and gets 1 point.

Attack target or user group	Host system	Other environments
Students	0.4	0.4
Lecturers	0.1	0.1

Table A.2: If an approach is vulnerable to a given attack target from a user group, then the penalty is subtracted from 1. The result is the point assigned to the approach.

A.1.3 Developer experience

For the developer experience, we assign weighted points to each property listed in 4.3 according to how "important" it is for the developer experience.

The weights are chosen so that each approach's worst or best assignment corresponds to the total maximum and minimum attainable points. Table A.3 shows the points an approach receives if it fulfills the given property. In this case, it is not clear whether the approach fulfills the given property or not he receives half of the listed point.

As mentioned in the discussion (see 5.3.2), the developer experience of the NSAR and BIAR prototypes is very similar for each property. Thus the point total is for both 0.9 since they meet all properties except the third one (development and testing) since it is unclear if it is fulfilled. In contrast, the current approach has a total of 0.1 as it does not satisfy any property except 3, where it is not fully clear whether this property is satisfied.

Developer experience property number	1	2	3	4	5
Points	0.3	0.1	0.2	0.3	0.1

Table A.3: For each property, we assign points (real numbers in [0, 1]) to an approach if it fulfills the property. The properties are listed at the beginning of 4.4 and have the same enumeration order.

A.1.4 User experience

For the UX we assign weighted points to each property analogeous to the developer experience case mentioned above. Table A.4 shows the points that an approach receives which is analogeous to the developer experience. The approaches have the following point sums (in order of the properties):

- NSAR: 0.15 (unclear) $+ 0.3 + 0.2 + 0.1 = 0.75$
- BIAR: 0.15 (unclear) $+ 0.3 + 0.2 + 0.1 = 0.75$
- Current approach: 0.15 (unclear) $+ 0 + 0 + 0.2 = 0.35$

The assignment of each point is justified in 4.3 and 5.3. As argued in the discussion, it is unclear if each approach fulfills the first property (ease of configuration), and therefore, only half of the point is assigned in this case.

A.2 Statistical test resources

We used the following formula to decide if the variances between two datasets X_1, X_2 with correspoding variances $s^2_{X_1}$ and $s^2_{X_2}$ are similar or not. If $max(s^2_{X_1}, s^2_{X_2})/min(s^2_{X_1}, s^2_{X_2}) < 2$ then the variances are similar otherwise they

UX property number	1	2	3	4
Points	0.3	0.3	0.2	0.2

Table A.4: For each property, we assign points (real numbers in $[0, 1]$) to an approach if it fulfills the property. The properties are listed at the beginning of 4.3 and have the same enumeration order.

are not. This "rule of thumb" is adapted from [13], where we chose 2 instead of 3 as upper bound for the ratio.

A.3 Additional benchmark graphs: seeded Nix store results

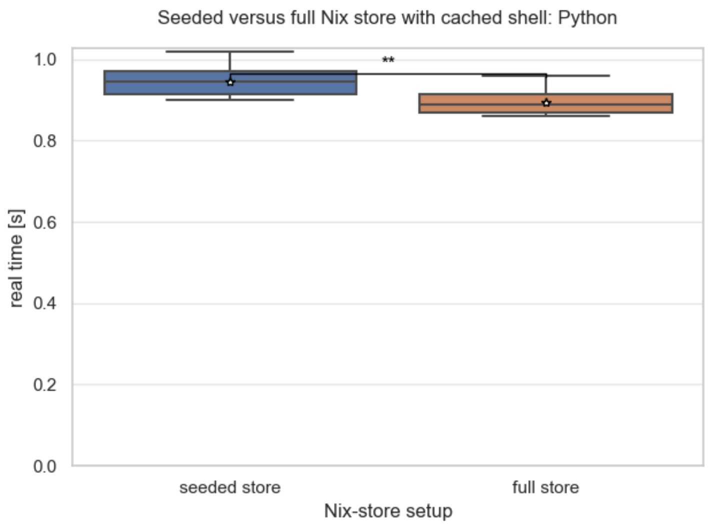


Figure A.1: The "seeded store" has a subset of the packages preinstalled to build the measured build time. The "full store" measures a subsequent build using the cached results of a first-time build with the same configuration. This graph shows the Python environment. Remarks: NSAR = nix-shell at runtime, BIAR = build image at runtime, n.s. = statistically nonsignificant difference., * = a higher number of stars(*) indicates a greater statistically significant difference.

A.4 Packages included in environments for benchmarks

The following configuration belong to the group C_{base} and the specified packages are included in the benchmarks of both prototypes.

A.4.1 Base packages

base-pkgs.nix

```
{ pkgs }:
{
  inputs = with pkgs; [
    coreutils-full
    bashInteractive
    curl
  ];
}
```

A.4.2 C++ packages

The Nix configuration that is used to prebuild the seeded Nix store is cpp.nix.

cpp.nix

```
{ pkgs }:
{
  inputs = with pkgs; [
    autoconf
    automake
    bison
    flex
    binutils
    gdb
    libtool
    cmake
    strace
  ];
}
```

The Nix configuration used for benchmarking has the additional numerical library package `eigen` in its configuration.

A.4.3 Python packages

The Nix configuration that is used to prebuild the seeded Nix store is python.nix.

python.nix

```
{ pkgs }: with pkgs;
let
  pythonEnv = python39.withPackages (p: with p; [
    scipy
    numpy
    pandas
    termcolor
    networkx
    gnureadline
    jinja2
    matplotlib
  ]);
in
{
  inputs = [ pythonEnv ];
}
```

The Nix configuration that is used for benchmarking (see bench-python.nix) has `scikit-learn` as an additional Python package as well as six additional OS packages compared to python.nix.

bench-python.nix

```
{ pkgs }: with pkgs;
let
  pythonEnv = python39.withPackages (p: with p; [
    scipy
    numpy
    pandas
    termcolor
    networkx
    gnureadline
    jinja2
    matplotlib

    # added Python package
    scikit-learn
  ]);
in
{
  inputs = [
    pythonEnv

    # added OS packages
    ncurses
    bzip2
    openssl
    sqlite
```

```
      xz
   ];
}
```

A.4.4 Current approach packages

The `Python-3_8` cxEnvironment, which we used for benchmarking, has the same OS packages and Python packages as bench-python.nix. The `GCC` configuration we used for benchmarking and comparing to our `C++` environment has, apart from an additional package (`eigen`), the same packages installed as cpp.nix. Note that the `nixos/nix` base image already has the `gcc` and `gcc-c++` compiler installed and are therefore not in the Nix configuration.

Bibliography

[1] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459–474, 2013. Special Section: Component-Based Software Engineering (CBSE), 2011.

[2] Lewo Abesis. nix2container. https://github.com/nlewo/nix2container, 2022. Online; Retrieved 22 June 2022.

[3] Leonhard Applis. Theoretical evaluation of the potential advantages of cloud ides for research and didactics. In Michael Becker, editor, *SKILL 2019 - Studierendenkonferenz Informatik*, pages 47–58, Bonn, 2019. Gesellschaft für Informatik e.V.

[4] Gaurav Banga and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, February 1999. USENIX Association.

[5] Bazel. Hermeticity. https://bazel.build/concepts/hermeticity, 2022. Online; Retrieved 16 May 2022.

[6] Justin Bedő, Leon Di Stefano, and Anthony T Papenfuss. Unifying package managers, workflow engines, and containers: Computational reproducibility with BioNix. *GigaScience*, 9(11), 11 2020. giaa121.

[7] Dominik Braun. Docker images and their layers explained. https://dominikbraun.io/blog/docker/docker-images-and-their-layers-explained/ , 2022. Online; Retrieved 9 May 2022.

[8] Neil Brown. Overlay filesystem — the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html, 2022. Online; Retrieved 26 June 2022.

[9] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. Ide 2.0: Collective intelligence in software development. pages 53–58, 01 2010.

[10] Graham Christensen. Optimising docker layers for better caching with nix. https://grahamc.com/blog/nix-and-layered-docker-images, 2018. Online; Retrieved 23 May 2022.

[11] CodeExpert. Documentation for lecturers — codeexpert guide. https://docs.expert.ethz.ch/lecturers/, 2022. Online; Retrieved 13 May 2022.

[12] Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *2012 International Conference on Software and System Process (ICSSP)*, pages 73–77, 2012.

[13] Clay Ford. A rule of thumb for unequal variances. https://data.library.virginia.edu/a-rule-of-thumb-for-unequal-variances/, 2013. Online; Retrieved 27 June 2022.

[14] G. Fylaktopoulos, M. Skolarikis, I. Papadopoulos, G. Goumas, A. Sotiropoulos, and I. Maglogiannis. A distributed modular platform for the development of cloud based applications. *Future Generation Computer Systems*, 78:127–141, 2018.

[15] Niklas Hambüchen. nix-binary-cache-proxy. https://github.com/nh2/nix-binary-cache-proxy, 2017.

[16] Docker Inc. Docker documentation: Docker overview. https://docs.docker.com/get-started/overview/, 2022. Online; Retrieved 9 May 2022.

[17] Docker Inc. Docker documentation: Manage data in docker. https://docs.docker.com/storage/, 2022. Online; Retrieved 22 June 2022.

[18] Docker Inc. Docker documentation: Storage drivers. https://docs.docker.com/storage/storagedriver/, 2022. Online; Retrieved 9 May 2022.

[19] Docker Inc. Docker documentation: Use bind mounts. https://docs.docker.com/storage/bind-mounts/, 2022. Online; Retrieved 9 May 2022.

[20] Docker Inc. Docker documentation: Use volumes. https://docs.docker.com/storage/volumes/, 2022. Online; Retrieved 9 May 2022.

[21] Teuvo Kohonen. *Associative Memory, Content Addressing, and Associative Recall*, pages 1–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.

[22] Scott McCarty. A practical introduction to container terminology. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction, 2018. Online; Retrieved 24 June 2022.

[23] Peter Mell and Timothy Grance. The nist definition of cloud computing, 2011-09-28 2011.

[24] Adrian Mouat. *Docker*. dpunkt, 1st edition edition, 2016.

[25] Nixery. Nixery docs. https://nixery.dev/nixery.html, 2021. Online; Retrieved 23 May 2022.

[26] NixOS. Basic package management — nixos docs. https://nixos.org/manual/nix/stable/package-management/basic-package-mgmt.html, 2022. Online; Retrieved 17 May 2022.

[27] NixOS. Binary cache — nixos wiki. https://nixos.wiki/wiki/Binary_Cache, 2022. Online; Retrieved 08 June 2022.

[28] NixOS. Enter the environment — nix pills. https://nixos.org/guides/nix-pills/enter-environment.html, 2022. Online; Retrieved 16 May 2022.

[29] NixOS. Fetchers — nixos docs. https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers, 2022. Online; Retrieved 08 June 2022.

[30] NixOS. Hydra — nixos wiki. https://nixos.wiki/wiki/Hydra, 2022. Online; Retrieved 26 June 2022.

[31] NixOS. Introduction — nixos docs. https://nixos.org/manual/nix/stable/, 2022. Online; Retrieved 17 May 2022.

[32] NixOS. Multi-user mode — nixos docs. https://nixos.org/manual/nix/stable/installation/multi-user.html, 2022. Online; Retrieved 17 May 2022.

[33] NixOS. Nix channels — nixos wiki. https://nixos.wiki/wiki/Nix_channels, 2022. Online; Retrieved 25 June 2022.

[34] NixOS. Nix-shell command — nixos docs. https://nixos.org/manual/nix/stable/command-ref/nix-shell.html, 2022. Online; Retrieved 17 May 2022.

[35] NixOS. Overlays — nixos wiki. https://nixos.wiki/wiki/Overlays, 2022. Online; Retrieved 26 June 2022.

[36] NixOS. pkgs.dockertools — nixos docs. https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-dockerTools, 2022. Online; Retrieved 23 May 2022.

[37] NixOS. Profiles — nixos docs. https://nixos.org/manual/nix/stable/package-management/profiles.html, 2022. Online; Retrieved 16 May 2022.

[38] NixOS. Python user guide — nixos docs. https://nixos.org/manual/nixpkgs/stable/#python, 2022. Online; Retrieved 23 May 2022.

[39] NixOS. Why you should give it a try — nix pills. https://nixos.org/guides/nix-pills/why-you-should-give-it-a-try.html, 2022. Online; Retrieved 16 May 2022.

[40] Tim O'Hearn. The case for pinning versions of docker dependencies. https://www.tjohearn.com/2018/03/01/the-case-for-pinning-versions-of-docker-dependencies/, 2018. Online; Retrieved 10 May 2022.

[41] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 Second International Conference on Computer and Network Technology*, pages 222–226, 2010.

[42] David Sichau. Technical overview code expert. Power-Point Presentation; Slide 4, 1 2022.

[43] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery.

[44] Marc Stevens, Arjen Lenstra, and Benne Weger. Chosen-prefix collisions for md5 and applications. *International Journal of Applied Cryptography*, 2, 07 2012.

[45] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 72:165–179, 2017.

[46] Divya Vaishnavi and Eric Rosado. Github's engineering team has moved to codespaces — the github blog. https://github.blog/2021-08-11-githubs-engineering-team-moved-codespaces/, 2021. Online; Retrieved 3 June 2022.

[47] Sander van der Burg. A generic approach for deploying and upgrading mutable software components. In *2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pages 26–30, 2012.

[48] David Wagner. Building container images with nix. https://thewagner.net/blog/2021/02/25/building-container-images-with-nix/, 2021. Online; Retrieved 23 May 2022.

[49] Cory Wilkerson. Prepare for next semester with github global campus and codespaces — the github blog. https://github.blog/2022-05-09-prepare-for-next-semester-with-github-global-campus-and-codespaces/, 2022. Online; Retrieved 3 June 2022.

[50] Charles P. Wright and Erez Zadok. Kernel korner: Unionfs: Bringing filesystems together. *Linux J.*, 2004(128):8, dec 2004.

[51] xzfc. Cached-nix-shell docs. https://github.com/xzfc/cached-nix-shell, 2022. Online; Retrieved 28 May 2022.

[52] Hideaki Yanagisawa. Evaluation of a web-based programming environment. In *2012 15th International Conference on Network-Based Information Systems*, pages 633–638, 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Boost flexibility in coding exercises by building containers at runtime

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Stoll	Romeo Maurus

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Schaffhausen, 11.07.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.