FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Improving the Developer Experience of Dockerfiles

João Pereira da Silva Matos

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Filipe Correia

July 27, 2023

Improving the Developer Experience of Dockerfiles

João Pereira da Silva Matos

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Jácome Cunha
Referee: Prof. Filipe Correia
Referee: Prof. Florian Rademacher

July 27, 2023

Resumo

A contentorização é uma técnica usada num número bastante elevado de sistemas para facilitar a instalação dos mesmos. A ferramenta de contentorização mais popular é o Docker. Para utilizar Docker, um desenvolvedor cria um Dockerfile, um ficheiro de configuração usado para criar contentores. Criar estes ficheiros nem sempre é uma tarefa simples, e até ficheiros funcionais podem ter problemas. Para além disso, os desenvolvedores também consideram que o desenvolvimento de Dockerfiles é uma tarefa que consome bastante tempo.

Uma revisão de literatura foi desempenhada para determinar os desafios com que os desenvolvedores de Dockerfiles se deparam e como têm lidado com eles até agora. Descobrimos que longos tempos de construção e cheiros são alguns dos problemas que afetam os desenvolvedores, e várias abordagens têm sido propostas para lidar com estes problemas.

Nós acreditamos que a situação pode ser melhorada através da oferta de um ambiente com um elevado nível de *liveness* aos desenvolvedores. Um ambiente deste tipo deve ser capaz de melhorar a experiência de desenvolvimento e a qualidade dos Dockerfiles que os utilizadores criam. Esta é a nossa hipótese.

Com isto em mente, criámos uma ferramenta (baseada no Dockerlive) que visa melhorar a experiência de desenvolvimento e a qualidade dos Dockerfiles através da oferta de um elevado nível de *liveness* com a implementação de funcionalidades como geração e modificação automática de ficheiros. A geração permite a um desenvolvedor gerar um Dockerfile com base na informação de um projeto, enquanto que a reparação encontra problemas num Dockerfile e sugere ao utilizador modificações que resolvem estes problemas.

Para avaliar o impacto desta ferramenta, realizámos um estudo empírico, mais especificamente, uma experiência, com participantes da indústria. Os dados que foram recolhidos mostram que a nossa abordagem consegue reduzir a quantidade de tempo e de alterações de contexto que os utilizadores precisam de realizar quando trabalham com Dockerfiles. Consequentemente, isto reduz a carga cognitiva, promove a utilização de *flow state* e aperta os ciclos de *feedback*, levando a uma melhor experiência de desenvolvimento.

Palavras-chave: Dockerfile, Docker, Geração de ficheiros, Reparação de ficheiros, Experiência de desenvolvimento, Infra-estrutura como Código, Ambientes de Desenvolvimento

Classificação ACM: CCS → Software e a sua engenharia → Notação de software e ferramentas → Frameworks e ambientes de desenvolvimento → Ambientes de desenvolvimento integrados e visuais

Abstract

Containerization is a technique used in a very large number of systems to ease deployment. The most popular containerization tool is Docker. To use it, a developer writes a Dockerfile, a configuration file used to create the containers. Creating these files is not always straightforward, and even functional files can have problems. Furthermore, developers also consider Dockerfile development to be a time-consuming task.

A literature review was performed to determine the challenges that Dockerfile developers face and how they have been addressed so far. We discovered that long build times and smells are some of the problems affecting developers, and many approaches have been proposed to address these problems.

We believe the situation can be improved by offering developers an environment with a high level of liveness. Such an environment should be capable of enhancing the development experience and the quality of the Dockerfiles that users create. This is our hypothesis.

With this in mind, we have built a tool (based on Dockerlive) that aims to improve the development experience and the quality of the Dockerfiles by offering high liveness through the implementation of features like automatic generation and modification of files. The former allows a developer to generate a Dockerfile based on project information, while the latter finds problems in a Dockerfile and suggests modifications that fix these problems to the user.

To evaluate the impact of this tool, we conducted an empirical study, more specifically, an experiment, with participants from the industry. The data that was gathered shows that our approach can lower the amount of time and context switching that users need to perform when working with Dockerfiles. Consequently, this reduces cognitive load, promotes flow state, and tightens feedback loops, leading to a better development experience.

Keywords: Dockerfile, Docker, File generation, File repair, Development experience, Infrastructure as Code, Development Environments

ACM Classification: CCS → Software and its engineering → Software notations and tools → Development frameworks and environments → Integrated and visual development environments

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Filipe Correia, for guiding me through this journey and teaching me everything I needed to complete it.

Secondly, I would like to thank everyone I worked with and learned from during the years I have spent at *Faculdade de Engenharia da Universidade do Porto*. You have all taught me something important, and I have grown as a person because of it.

Finally, I would like to thank my mother for always supporting me and being the best mother on the planet.

João Pereira da Silva Matos

"Most good programmers do programming
not because they expect to get paid or get adulation by the public,
but because it is fun to program."

Linus Torvalds

Contents

List of Figures

List of Tables

Listings

Abbreviations and Symbols

AST	Abstract Syntax Tree
DevOps	Developer Operations
HPC	High Performance Computing
IaC	Infrastructure as Code
IT	Information Technology
OS	Operating System
DSL	Domain-specific Language
IDE	Integrated Development Environment
SATD	Self-admitted Technical Debt
APT	Advanced Packaging Tool
APK	Alpine Package Keeper
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
DAVS	Dockerfile analysis-based vulnerability scanning
RPM	RPM Package Manager
NPM	Node Package Manager
URL	Uniform Resource Locator
CLI	Command-line Interface
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
PDF	Portable Document Format
OBS	Open Broadcaster Software
CSV	Comma-separated values
ID	Identification
UI	User Interface

Chapter 1

Introduction

In this chapter, we introduce the topic of this document, explain what we aim to achieve, and the contributions we plan to make.

1.1 Context

In recent years, we have seen an increase in the adoption of DevOps (Developer Operations) practices. These practices allow companies to automate tasks related to development and IT (Information Technology) operations, with the intention of improving and speeding up the software development life cycle [24].

An important part of DevOps practices is the use of IaC (Infrastructure as Code), a set of practices where machine-readable configuration files enable the automation of IT infrastructure maintenance, allowing organizations to operate at a larger scale than would otherwise be possible if everything was done manually by humans [21].

The surging popularity of these practices translates into a growing need to operate at a large scale. However, this need to work at a large scale does not just apply to the infrastructure or the methodologies used by the developers. It also applies to the systems that companies are building. They are becoming more complex and, consequently, harder to maintain.

In response to this, architectures composed of microservices have been gaining popularity. This architecture uses several small services that each have a specific purpose and communicate with each other to create more complex systems. These systems typically use containers to create each service, and Docker is the most popular container platform available [23, 55].

To use Docker, one must first create a configuration file called a Dockerfile containing the information required to build the container. However, creating high-quality Dockerfiles is not always trivial. In fact, a large portion of Dockerfiles used in projects contains smells [59]. Many Dockerfiles can not even be used to build an image [60]. Furthermore, developers also report spending more time than they would like creating containers [49].

1.2 Objectives

We think the problems affecting Dockerfile developers need to be addressed. The existence of these problems suggests developers' perception of their work (in other words, the developer experience [40]) is negative and their development environment lacks the ability to quickly provide feedback after performing modifications (also known as liveness [50]). It also shows that the quality of the Dockerfiles can be improved.

Therefore, our objectives are to improve the Dockerfile developer experience and the quality of the Dockerfiles. With this in mind, this is our hypothesis:

The Dockerfile developer experience and the quality of the resulting images can improve if the environment provides a high level of liveness regarding the most challenging aspects of Dockerfile development

We believe that by providing developers with a development environment that offers a high level of liveness, the development experience and the quality of the Dockerfiles can be improved. Our hypothesis is aligned with the proposal made by Aguiar et al. and the concept of Live Software Development where higher levels of liveness are used to tighten feedback loops by giving more information to the developer [2]. Furthermore, data provided by Reis also suggests that higher levels of liveness can improve the development experience [50].

We plan to accomplish this by providing a tool (based on Dockerlive[1]) that offers automatic modification and generation of files. We believe this set of features can also improve the quality of Dockerfiles.

1.3 Methodology

Now that we know what our objectives are, we need to know how we are going to achieve them. We will first create a tool, Dockerlive V2, that offers these features to developers, and then, we will conduct a study where developers use this tool to write Dockerfiles.

1.3.1 Dockerlive V2

We will develop a tool that extends Dockerlive [50, 48] (covered in section 2.12), a Visual Studio Code[2] extension. By itself, Dockerlive already provides high liveness. However, it does not have the ability to automatically modify or repair Dockerfiles, so we will add these capabilities to increase the level of liveness that we can offer to a user.

[1] Dockerlive - Visual Studio Marketplace, https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive

[2] Visual Studio Code, https://code.visualstudio.com/

Before going any further, we will explain what the levels of liveness are. Tanimoto defined 6 levels of liveness that an environment could provide, with higher levels providing more liveness than lower levels. In level 1, the environment merely provides some information to the programmer. As the levels increase, the environment must provide more relevant information at a faster rate while reducing the amount of input that is required from the user. Eventually, the environment starts to predict the user's intentions and perform large modifications based on these predictions [57]. This corresponds to the 6^{th} level of liveness.

The functionality provided by Dockerlive and implemented by Reis [50] corresponds to the 4^{th} level of liveness that is described by Tanimoto [57]. The features we propose go beyond the 4^{th} level, but we can not say they correspond to the 5^{th} level since the implementation proposed by Tanimoto for this level would require more processing power than many development machines can offer. The tool we propose could be considered a more realistic implementation of the 5^{th} level of liveness.

The generation functionality will be offered by Hermit [34] (covered in section 2.5). This is something that Dockerlive is completely incapable of doing as it does not possess any components that can generate files of any kind.

Although in its current state, Dockerlive is already capable of detecting some problems in the Dockerfiles it analyzes, there are still limits to what it can detect. For example, it can not detect sub-optimal use of package managers. Furthermore, it can only tell the user where they might want to perform modifications, it can not perform them by itself. For these reasons, we will develop our own components that are used to perform modifications to existing Dockerfiles.

The final result will be a Visual Studio Code extension that can generate Dockerfiles and perform some automatic modifications while maintaining all of Dockerlive's original functionality, creating an environment that offers a high level of liveness and improves the development experience. The architecture of this tool is described in more detail in Section 3.3.

1.3.2 Study

To answer our research questions, we will conduct a study (an experiment[3]) with the intent of evaluating the extent to which liveness is actually capable of improving the development experience and the quality of the Dockerfiles created by professional software developers. This study will be an experiment with two groups of professionals where each group has to perform the same set of tasks, but while one of them has access to the tool, the other one does not have access to any special tools. A short questionnaire will be run through the participants before the experiment to make sure the level of proficiency and experience is similar across both groups. Another questionnaire will be used during the experiment to evaluate the participants' perceptions. During the experiment, we will also gather data by measuring the amount of time spent on each task and the amount of context switching that is performed. Finally, data will also be gathered from the participants' Dockerfiles.

[3]Empirical Standards - Experiments, https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=Experiments

1.4 Contributions

With the work outlined in the previous section, we expect to bring more attention to this topic and further develop what is currently known about Dockerfile development. With this in mind, these are the contributions that we made:

- An analysis of the current challenges affecting Dockerfile development and the approaches that aim to address them
- A tool that extends Dockerlive by adding automatic generation and repair functionality to a high-liveness environment to help developers write Dockerfiles
- A replicable empirical study (more specifically, an experiment) with participants from the industry from which we will gather data regarding the effect of the new features on the development experience

Firstly, we decided to conduct a literature review to analyze the problems affecting Dockerfile developers. We also decided to create a tool with the features that have been mentioned, as we believed they would allow us to create an environment with a higher level of liveness than would otherwise be possible. We also thought a study was necessary to verify if the tool had the desired impact on the Dockerfile development experience.

1.5 Document Structure

This chapter provides a brief introduction regarding the topic of this document, along with our objectives and contributions.

After this chapter (Chapter 1) establishes our motivations, Chapter 2 goes over the state of the art and provides the background that is required to understand the rest of the document. This allows us to learn more about the problem we are trying to address.

After this analysis, Chapter 3 describes the tool we implemented.

The tool is then evaluated with an empirical study, which is described in Chapter 4.

Finally, Chapter 5 summarizes all the information the other chapters present and concludes our work.

Chapter 2

Dockerfile Development Challenges

Now that we've established why it's important to research Dockerfile development, we need to look into the existing literature and understand the problems surrounding this task. At the same time, we will also provide any background that is required to understand the topics discussed in this section.

2.1 Background

This section covers concepts that are required to understand the rest of the document, including Docker (cf. Section 2.1.1), Developer Experience (cf. Section 2.1.2) and Liveness (cf. Section 2.1.3).

2.1.1 Docker

A problem commonly faced during development tasks is the variety of different systems and configurations used for not only development but also deployment. Modern applications require many dependencies, and ensuring every single machine contains the correct set of dependencies is not always efficient or straightforward [8].

A common solution for this problem is the use of containers [56]. Containers are isolated environments containing all the dependencies that are required for an application to be executed. Containers can't be executed directly by the hardware, requiring a host OS (operating system) in order to be used. In this regard, containers might seem similar to virtual machines. However, because containers don't require a hypervisor, they have a much lower overhead, making them preferable in many scenarios [22]. Figure 2.1 illustrates these differences.

Docker[1] is the most popular container platform available and offers a suite of tools that allow developers to create their own containers. To accomplish this, a developer starts by creating a Dockerfile[2], a file written in a DSL (domain-specific language) with instructions that tell the

[1] Docker, https://www.docker.com/

[2] Dockerfile Reference, https://docs.docker.com/engine/reference/builder/

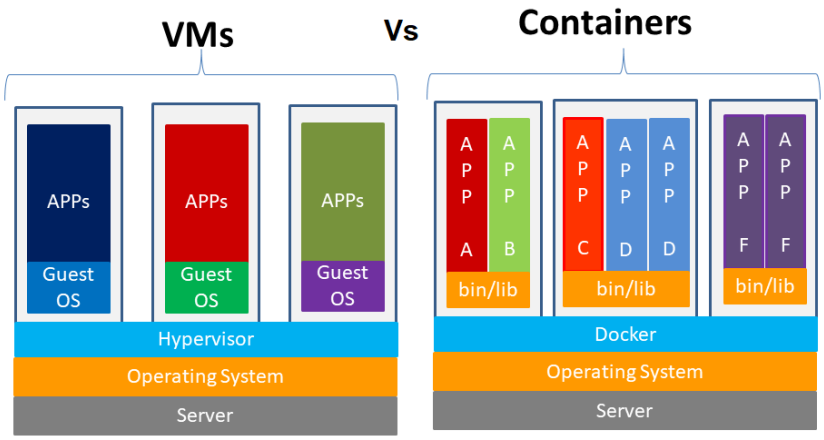

Figure 2.1: Differences between containers and virtual machines
Source: [10]

Docker Engine[3] what characteristics the environment should have. Each instruction leads to the creation of a layer, with each one having a hash associated with it to verify integrity. Afterward, the information from all the layers in this Dockerfile is processed by the Docker Engine and used to build an image. Usually, developers don't develop their own images from scratch and rely on preexisting images hosted on a repository like Docker Hub[4]. Because of this, during the build phase, the engine usually has to fetch several files from the network. These files contain information about the image that is used as a base for the new image. After the build procedures are complete, an image can be executed, resulting in a container that follows the previously defined configuration.

Something else that we need to explain is how code smells and good practices present themselves in the context of Dockerfiles. Code smells are elements that may indicate the existence of a deeper problem in the project being developed [14]. However, smells can also show up in other development artifacts like configuration files. This also applies to Dockerfiles.

For example, in some situations, it may be necessary for a container to use software that is distributed as a tarball[5]. In order to use this kind of software, a developer must first extract the contents of the tarball. However, after the extraction, the tarball itself is no longer needed and should be removed to avoid wasting space. Failing to do so creates an instance of the temporary file smell [33].

A concept that is related to smells is the concept of good practices, a set of procedures that should be followed by a developer in order to maximize efficiency and avoid problems. Typically, following these practices also reduces the number of smells that are created. There are also good practices that should be followed when writing Dockerfiles. Some are provided directly by

[3]Docker Engine, https://docs.docker.com/engine/
[4]Docker Hub, https://hub.docker.com/
[5]tar(computing), https://en.wikipedia.org/wiki/Tar_(computing)

Docker's developers[6] while others can be found in the literature [41].

2.1.2 Developer Experience

As the title of this document suggests, we are trying to improve the experience of those that need to edit Dockerfiles. The previous section explained what a Dockerfile is. Now, we move on to the concept of Developer Experience. Before proceeding, it should be noted that for our purposes, "developer" and "programmer" are almost synonymous terms. According to Morales et al., the programmer experience is "the result of the intrinsic motivations and perceptions of programmers regarding the use of development artifacts" [38]. In other words, developer experience encapsulates the way a developer feels about the artifacts that are used to aid development tasks. In the context of our work, the artifact we are concerned with is the configuration file used by Docker, the Dockerfile.

An alternative definition is proposed by Noda et al., this definition divides developer experience into "three core dimensions: feedback loops, cognitive load, and flow state" [40]. They also suggest that developer experience is affected by other elements besides those directly related to development tasks, broadening the scope of developer experience to include elements like company protocols and the frequency of meetings. This is the definition that the rest of our work is based on.

2.1.3 Liveness

The last concept we need to define is liveness. This concept is important because it is through liveness that we aim to improve the developer experience, which was the focus of the previous section. Our definition is, again, based on other authors'. Maloney and Smith define a live interface as one that is always being updated and providing the user with information, whether it's in response to a user's actions or not [35]. Furthermore, Aguiar et al. provide a very similar definition and emphasize the advantages of applying liveness to the different activities of the software development life cycle as a way to provide a developer with more immediate feedback regarding what is currently being constructed [2].

2.2 Goals and methodology

Our end goal is to help developers write Dockerfiles. To do this, we first need to know what challenges they face when writing Dockerfiles and how the quality of the Dockerfiles is affected by these challenges. Furthermore, we need to analyze current solutions that address these issues in order to build upon them. Finally, we should try to understand if the features we're proposing can actually help address these challenges. With this in mind, we came up with the following research questions to guide our review of the state of the art:

[6] Best practices for writing Dockerfiles, https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

- **RQ1**: What are the challenges that developers face in the development of Dockerfiles?
- **RQ2**: How have these challenges been addressed so far?
- **RQ3**: To what extent can these challenges be addressed by automatic repair or generation approaches?

To answer these questions, we tried to find as much information as possible about Docker and Dockerfiles. Furthermore, we also tried to find works that analyzed a developer's subjective experience when performing development tasks. We ended up using the following queries to look for information in Google Scholar[7]:

- dockerfile challenges
- dockerfile generation
- distroless
- docker build
- dockerfile
- dockerfile creation
- dockerfile generator
- dockerfile repair
- docker repair
- dockerfile readability
- dockerfile evaluation
- dockerfile analysis
- docker bloat
- docker build time
- dockerfile practices
- programming experience
- developer experience

These queries gave us plenty of results, which had to be filtered to reach an amount that could be reasonably analyzed in the amount of time that we were given.

First, we decided to focus on recent works. With Docker being a relatively young project, it made no sense to include any works produced before 2013. At the same time, Docker has changed considerably over time, making some of the older works outdated. We ended up selecting works

[7]Google Scholar, https://scholar.google.com/

from the last six years (it should be noted that our analysis was performed in 2022, although we managed to add some works released in 2023 while working on subsequent parts of this dissertation).

Secondly, we had to evaluate if Docker was actually the focus (or at least an important part) of the works we selected or if it was just used as a tool due to its convenience. At this point, we analyzed the abstracts in order to eliminate a few more works from the set.

Finally, we ended up having to remove a few more works from the set due to them being inaccessible, even after authenticating with our institutional account.

After all this filtering, we ended up with around 50 works to analyze. Most of these were analyzed, although some had to be excluded due to time constraints.

2.3 Challenges in the Development of Dockerfiles

After looking at the literature, we identified several aspects of the Dockerfile development experience that could be considered challenging.

Many projects struggle to follow good practices [17]. This can affect other aspects of the container, like, for example, security [45]. Furthermore, a large portion of Dockerfiles contains smells [59]. Another problem is the amount of bloat (this can include packages or tools that are either unnecessarily installed in a new image or are already part of the base image) present in images [1]. Additionally, developers report that several parts of the Dockerfile development experience are time-consuming. For example, debugging often relies on a slow trial-and-error approach [49], a situation that is further exacerbated by the amount of time required to build an image [20].

This information is summarized in the following list, containing several challenging aspects of Dockerfile development:

- The amount of time required to build an image from a Dockerfile (Reis et al. [49] conducted a survey where participants reported spending a considerable amount of time rebuilding images and re-running containers)
- The number of vulnerabilities present in Dockerfiles and the corresponding images (Zarei [63] found that almost half of developers do not know if the images they are using contain vulnerabilities, furthermore they found that multiple scanners have detected a large number of vulnerabilities in popular images)
- The amount of bloat present in Docker images (Agadakos et al. [1] found that almost a quarter of the code present in popular images is bloat)
- The reliance on trial-and-error to test Dockerfiles (Reis et al. [49] found that trial-and-error is a common approach when writing Docker artifacts)
- The number of smells contained in Dockerfiles (Wu et al. [59] found that more than four-fifths of the Dockerfiles in their dataset had a smell)

- The amount of Dockerfiles that do not follow best practices (Henkel et al. [17] found that Dockerfiles on GitHub often violate practices mined from Dockerfiles written by experts)

With these issues in mind, this chapter contains sections that address each of these problems. Additionally, there are also sections dedicated to repairing and generating Dockerfiles since those are features that our tool will provide and could improve the Dockerfile development experience.

2.4 Speeding up Docker builds

Building Docker images can take a considerable amount of time, especially when a large number of files have to be fetched from the internet [15]. Therefore, we looked for approaches to reduce the amount of time consumed by this activity. Our findings are summarized in Tables 2.1 and 2.2.

Name	**Speedup**	**Limitations**	**Language used**
Code Injection [58]	Up to 100000x faster	Limited to interpreted languages and modifications in the source code	Unknown
FastBuild [20]	Up to 10x faster	Limited to network activity	Go
Slacker [15]	Up to 20x faster	Limited to network activity	Unknown
Docker Buildx[8]	Unknown	Unknown	Go

Table 2.1: Approaches that aim to speedup Docker builds along with their speedups, limitations and languages

Name	[58]	[20]	[15]	Buildx
Overrides Docker?	✓	✓	✓	X
Integrated in development environment?	X	X	X	✓
Available source code?	X	X	X	✓

Table 2.2: Approaches that aim to speedup Docker builds along with their need to override Docker's behavior, IDE integration and source code availability

Wang and Bao [58] propose a technique that bypasses typical building procedures by injecting the code modifications directly into an image. The results are very promising. However, due to the nature of the approach, it can only be used with interpreted languages and will not accelerate builds related to modifications in development artifacts that are not source code. Furthermore, in order to inject the modifications in an image without having to rebuild it, the authors need to

[8]Docker Buildx, https://docs.docker.com/engine/reference/commandline/buildx/

update the associated hashes and prevent Docker from performing integrity checks. Lastly, this work is a few years old, which could suggest some level of maturity, but the proposed tool has not been made available to the public, making it difficult to evaluate.

Huang et al. [20] and Harter et al. [15] address the network bottleneck in different ways. The former caches files locally and intercepts Docker's network requests in order to serve files that have been stored locally. The latter proposes a new storage driver that lazily fetches files from the network. Both show promising results and do not address other inefficiencies in the Docker building process. These tools are also a few years old, but, as we mentioned for the code injection approach, it's hard to make assumptions about their maturity or usability since the authors did not make them available to the public. Lastly, FastBuild was implemented using around 2600 lines of Go code.

The works described so far override the normal Docker build procedures (this means they interfere with the build process in ways that are not officially supported by the Docker development team) and do not have public source code for their tools. This makes them harder to implement or include as part of a development environment. Another solution is offered by the Docker development team, Buildx. Buildx is a different way to build images that makes use of a newer backend, BuildKit[9], which brings many features that can potentially accelerate docker builds. Furthermore, this new backend has been in development for a few years, has been shipped with Docker Engine for a similar amount of time, and is even used by other projects[10], cementing it as a mature project. However, to our knowledge, an apples-to-apples time comparison has not been made. Implementing this in Dockerlive wouldn't be very hard, but because the output of the *buildx* command is different, some modifications would still be required. From a normal user's perspective, it is also not hard to use, merely requiring a change in Docker's configuration in order to be used as part of the typical Docker development environment.

2.5 Dockerfile Generation

Many developers report some tasks involved in Dockerfile development as being time-consuming, especially for those with less experience [49]. Therefore, having a way to automatically generate a functional Dockerfile for a given project could be very useful as it would reduce the amount of time spent on these tasks, leading to higher productivity. In this section, we looked for works that showcased ways to accomplish this. Our findings are summarized in Tables 2.3 and 2.4.

Zhong et al. [64], Ye et al. [62], and Horton and Parnin [19] present solutions that require pre-existing knowledge bases in order to generate the files. Similarly, Rosa et al. [51] propose a solution that makes use of Deep Learning and requires large amounts of data to train the model. This makes them harder to implement in a project due to the amount of data required to build these knowledge bases. This is especially problematic for DockerGen and Burner because their source code is not publicly available. DockerizeMe is also limited to Python environments, while Burner

[9] Docker BuildKit, https://docs.docker.com/build/buildkit/
[10] BuildKit - Used by, https://github.com/moby/buildkit#used-by

Name	**Successful generation rate**	**Limitations**
Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations [54]	Unknown	An Open API Spec is required
Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph [64]	Up to 80%	A vast knowledge graph (2832 nodes and 62614 edges) is required, focused on Singularity
Container-Based Module Isolation for Cloud Services [26]	Unknown	Requires the use of templates to generate the files
DockerGen: A Knowledge Graph based Approach for Software Containerization [62]	Up to 73%	A vast knowledge graph (900000 nodes and 2900000 edges) is required
DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets [19]	Up to 30%	Limited to Python snippets, requires a knowledge base
ExploitWP2Docker: a Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges [9]	Up to 39%	Limited to security testing scenarios, requires an exploit description
MAKING CONTAINERS EASIER WITH HPC CONTAINER MAKER [37]	Unknown	Requires Python code to generate the files
Automatic service containerization with Docker [34]	Up to 69.5%	Limited to Python and Javascript projects
Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises [51]	34%	Large amounts of data are required for training

Table 2.3: Works about generating Dockerfiles along with their successful generation rates and limitation

Name	[54]	[64]	[26]	[62]	[19]	[9]	[37]	[34]	[51]
Available source code?	✓	X	✓	X	✓	✓	✓	✓	✓
Integrated in IDE?	X	X	X	X	X	X	X	X	X
Uses source code as input?	X	X	X	X	✓	X	X	✓	X

Table 2.4: Works about generating Dockerfiles along with their source code availability, IDE integration and ability to use source code as input

is more focused on Singularity[11], a containerization tool similar to Docker but focused on HPC (High-Performance Computing).

Caturano et al. [9] propose a tool that uses Docker to generate security testing environments from exploit descriptions. Sorgalla et al. [54]'s work can generate Dockerfiles from models, which are generated from Open API Specifications (like Swagger[12]). Kehrer et al. [26] use Apache FreeMarker[13] to generate Dockerfiles from templates. McMillan [37] offers a tool that allows developers to use Python code to define the information required to generate Dockerfiles. Lastly, Maduro [34] proposes Hermit, a tool that can generate Dockerfiles for Javascript and Python projects by performing dynamic and static analysis using the project's source code.

With the exception of Burner and DockerGen, most of the tools' source code is publicly available. However, none of them have been integrated into an IDE. Additionally, only two of the tools can generate Dockerfiles by analyzing a project's source code, with the others requiring additional input to perform this generation.

As we can see by looking at Table 2.3, these works show varying degrees of success. Some have a successful generation rate as high as 80%, while others don't even mention the success rate. In general, a successful generation means that an image can be built from the generation's output but the numbers provided have slightly different meanings depending on the work they belong to. Furthermore, given the different capabilities of each tool and the different ways in which these figures were calculated, it is hard to draw comparisons based on these numbers alone.

For our purposes, since we are trying to improve a developer's experience, we would like to minimize the amount of input that is required from the user making Hermit and DockerizeMe more appealing since they can analyze a project's code to perform the generation. Since DockerizeMe is limited to Python snippets, Hermit becomes the most desirable option.

Unfortunately, only Maduro provided some data regarding the characteristics of the generated files and reported that, for some metrics, the generated images were worse than those created by human developers [34]. This suggests that automatic generation may not be as helpful as we thought. However, we believe that, when combined with automatic modifications, the resulting tool could still be useful to developers.

2.6 Dockerfile Smells

Smells are commonly found in Dockerfiles [59], making it important to create ways of detecting and, if possible, removing them. This section covers works related to this. Table 2.5 contains a list of smells found in the literature.

Lu et al. [33] and Xu et al. [61] have focused on the temporary file smell and propose ways to detect this smell. A repair to deal with this smell could be implemented using the information provided by these works.

[11] Singularity, https://apptainer.org/
[12] Swagger, https://swagger.io/
[13] Apache FreeMarker, https://freemarker.apache.org/

Azuma et al. [5] focus on a variation of smells they call SATD (self-admitted technical debt) which can be detected in comments written in the Dockerfiles. Due to the nature of these SATDs, implementing repairs to eliminate them could be very complicated.

Wu et al. [59] analyzed a large number of open-source projects and found that Dockerfile smells are very common, and their frequency changes according to several factors like the programming language used by the project or the project's age. The authors used Hadolint[14] for the detection, allowing them to cover a large number of smells[15]. Due to the number of smells covered by this tool, implementing repairs to deal with all of them would be difficult. For the same reasons, only some of the analyzed smells were listed here.

Smell	**Related works**	**Related findings**
Temporary File	[33] and [61]	The smell is quite common and can be divided into four different types; can be detected through static and dynamic analysis
SATD	[5]	This type of smell appears in Dockerfiles and can be divided into several classes and subclasses
Lack of version tagging/pinning	[59]	N/A
Use of the MAINTAINER instruction	[59]	N/A
Use of *cd* to switch directories instead of WORKDIR	[59]	N/A
The parameter *–no-install-recommends* is not used when installing packages with APT	[59]	N/A

Table 2.5: Dockerfile smells

2.7 Dockerfile Good Practices

To prevent the creation of smells like the ones mentioned in Section 2.6, a developer should follow good practices. This section goes over works that cover these practices. Table 2.6 contains a list of good practices found in the literature.

[14]Hadolint, https://github.com/hadolint/hadolint
[15]Hadolint - Rules, https://github.com/hadolint/hadolint#rules

Henkel et al. [17] mined rules from Dockerfiles created by experts, allowing them to create a set of "gold rules", a set of patterns that often appear in Dockerfiles written by these experts. Some of these "gold rules" are not listed above because it's not clear what they refer to. It should be possible to implement the listed ones as repairs.

Giorgi [45] looked for flaws in Dockerfiles that could lead to vulnerabilities in a system. As part of that work they list a set of practices developers should follow to improve a Docker image's security. Most of the practices listed could be implemented as repairs, although some of them would be too complex to implement.

Nust et al. [41] propose a list of 10 rules developers should follow when writing Dockerfiles for data science environments. Some of these rules are applicable to other scenarios and 2 of those could be implemented as repairs.

Practice	**Related works**
Format for clarity	[41]
Document within the Dockerfile	[41]
Specify software versions	[41]
Order the instructions	[41]
Run the container in rootless mode	[45]
Use tagged minimal images and multistage builds	[45]
Use COPY with specific parameters	[45]
Update and install packages in the same RUN instruction	[45]
Use COPY instead of ADD	[45]
Do not leak sensitive information to an image	[45]
Remove unnecessary dependencies	[45]
Only expose ports that are needed	[45]
Use official images when possible	[45]
Remove temporary directories	[17]
Use flag *-f* with curl	[17]
Remove tarballs after extraction	[17]
Do not use APK's cache	[17]
Do not install dependencies recommended by APT	[17]
Use HTTPS urls with curl	[17]
Use batch flag with gpg	[17]
Use HTTPS urls with wget	[17]
Use flag *-y* with *apt-get install*	[17]
Remove APT lists after package installation	[17]
Run *apt-get update* before *apt-get install*	[17]

Table 2.6: Dockerfile good practices

2.8 Dockerfile Security

Nowadays, security is a topic that is heavily discussed and deserves a great amount of attention from developers. However, security problems are still commonly found in Dockerfiles [11] and many developers do not have the knowledge required to evaluate how vulnerable their containers are [63]. For these reasons, it's important to study Docker containers from a security perspective, which is what this section focuses on. Our findings are summarized in Table 2.7.

Name	**Findings**	**Implementation notes**
DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning [11]	DAVS can detect more vulnerabilities than competing scanners	It should be possible to repair some of the mentioned vulnerabilities, although it would be easier to use existing scanners
Investigating the inner workings of container image vulnerability scanners [63]	Many scanners use the same methods to detect vulnerabilities, which have limitations	Using one of these scanners could be useful
Outdated software in container images [31]	Having outdated software in containers brings security problems and there are limitations to what current scanners can detect, new detection method is proposed	It should be possible to implement some repairs to try to address this situation
Security Analysis of Code Bloat in Machine Learning Systems [3]	Removing bloat from containers used in machine learning environments can considerably improve security	It should be possible to implement some repairs that reduce bloat
Security Misconfigurations Detection and Repair in Dockerfile [45]	Security problems are common in containers, a way to repair them is proposed	It might be possible to implement the proposed technique to repair the problems

Table 2.7: Works about Dockerfile security

Doan and Jung [11] propose DAVS (Dockerfile analysis-based vulnerability scanning), a tool that can detect potentially vulnerable files in containers. This approach allows them to detect more vulnerabilities than current scanners, which, according to Zarei [63], rely on information provided by distributions' package managers. This information can be manipulated and, in some cases, may not even be available, which prevents scanners from detecting vulnerabilities.

Ahmed and Fatih [3] used Cimplifier [46] to debloat containers used in machine learning environments and found that the number of vulnerabilities present in those containers was significantly reduced.

Linnalampi [31] found that having outdated software introduces vulnerabilities in containers and proposed a new method to detect vulnerabilities by analyzing the binaries present in containers to detect the software versions that are in use. This approach would address some of the limitations of current scanning techniques.

Giorgi [45] found that security problems are common and proposed a way to repair them by processing the Dockerfile to obtain the abstract syntax tree, find the vulnerabilities and modify the tree before reconverting into a file that is no longer vulnerable.

Implementing repairs that address most of the problems and vulnerabilities found by these works should be possible. It may even be possible to use some of the proposed approaches.

2.9 Dockerfile Repair

As the previous sections have shown, the average Dockerfile has several problems and it can be difficult for a developer to figure out how to deal with those issues in an optimal way. This makes it important to create tools that can assist developers in the repair process. This section goes over works that do that (although other sections also discuss works that perform repairs that are related to more specific scenarios). Table 2.8 contains a list of repairs found in the literature.

Repair	**Related works**
Base image update	[27]
Gold rule enforcement (rules listed in Section 2.7)	[17]
Update portions of the Dockerfile which are tied to values in source code	[16]
Combine consecutive RUN instructions into one	[6]
Fix Ruby version error by pinning Ruby base image version	[18]
Fix RPM installation error by installing a plugin first	[18]
Update to the latest base image release	[18]
Install libpng-dev instead of libpng12-dev	[18]
Fix "Unable to locate package" by downgrading Ubuntu base image version	[18]
Fix Gemfile version error by pinning Ruby base image version	[18]
Fix Ruby encoding error by setting environment variable	[18]
Install missing packages using APK or APT without using cached data	[18]
Do not install the bzr package with APK	[18]
Add the *-L* flag when using curl to install conda	[18]

Table 2.8: Dockerfile repairs

Kitajima and Sekiguchi [27] focused on updating a container's base image by analyzing the available tags, while Hassan et al. [16] focused on portions of the Dockerfile which are tied to values in the source code.

Henkel et al. [17] offer a way to detect violations of the gold rules they obtained but don't automate the repair of said violations. Henkel et al. [18] also propose a different approach for automating repairs, although most of the repairs listed here are specific to certain programming languages or package managers.

Benni et al. [6] describe a way to reduce the number of layers in Dockerfiles in order to take advantage of layer caching.

Implementing the repairs mentioned in this section should be possible, although these implementations would have varying degrees of complexity.

2.10 Dockerfile Bloat

Many Docker images contain bloat [46] and their removal has benefits beyond the space savings. For example, a debloated image can be much more secure [3]. Because of this, we looked for works that described ways to address this problem. Our findings are summarized in Table 2.9.

Name	Removal rate	Limitations
Cimplifier: automatically debloating containers [46]	Up to 95% of image size	Relies on good test coverage
Large-scale Debloating of Binary Shared Libraries [1]	20-25% of code in the image	Debloating happens at the binary level and does not perform any Docker-specific changes
New Directions for Container Debloating [47]	N/A	The suggested approaches are not actually implemented
Wale: A Dockerfile-Based Approach to Deduplicate Shared Libraries in Docker Containers [52]	Around 40% of disk space	Only saves space when multiple containers are used

Table 2.9: Works about Dockerfile Bloat

Cimplifier [46] is a tool that removes bloat from containers by having the developer specify how the contents of an image should be divided and creating several images accordingly. However, single container environments can also be generated if specified by the developer. This tool can drastically reduce the size of an image. However, it relies on dynamic analysis and good test coverage to perform its functions. The same authors also wrote a second article describing other approaches that could be used to perform debloating on containers [47], although these alternative approaches were not actually implemented.

Nibbler [1] is a tool that performs debloating at the binary level and can remove a decent amount of bloat from an image. However, because it does not make use of any Docker-specific mechanisms, it could be difficult to implement in our tool.

Wale [52] is a tool that can debloat containers by placing packages used by multiple containers in a core container that is used by the others. This means it can not debloat systems that use a single container. Despite this, it can save a decent amount of disk space in the scenarios where it can be used.

2.11 Dockerfile Testing

One of the activities developers consider time-consuming is testing their containers to make sure they work as intended [49]. This makes sense considering problems like build failures are quite common [60]. Therefore, having tools to help them perform this task could have a positive impact

on their development experience. We looked for tools like this in the literature, and our findings are summarized in Table 2.10.

Name	**Coverage**	**Limitations**
DockerMock: Pre-Build Detection of Dockerfile Faults through Mocking Instruction Execution [29]	68%	Can not find problems in the environment surrounding the Dockerfile
Dgoss[16]	N/A	The developer must write the tests themselves
Container Structure Tests[17]	N/A	The developer must write the tests themselves

Table 2.10: Works about Dockerfile Testing

DockerMock [29] mocks common Dockerfile instructions in order to find problems before an image is even built. The results are promising, although issues outside the Dockerfile can not be detected.

Dgoss and Container Structure Tests make it easier to create and execute tests meant for containers but do not automate the creation of the tests.

2.12 Liveness

Since the coinage of the term *Live Software Development* by Aguiar et al. [2], other works have studied how the increase of liveness can impact different activities of the software development lifecycle [7, 12, 39, 13, 4], but very few of these works have approached infrastructure-related artifacts and tools [32, 48, 50].

We believe that liveness could be useful to address the challenges we have been discussing in the previous sections and improve the Dockerfile development experience. As mentioned previously, Dockerfile development usually makes use of a trial-and-error approach [49]. In this approach, a developer makes some modifications to the Dockerfile, builds a new image, and executes it to evaluate the behavior of the container and get feedback. If the container is behaving correctly, the process ends here. Otherwise, we go back to modifying the Dockerfile, and the loop restarts. This is the feedback loop (illustrated in Figure 2.2), which, as proposed by Aguiar et al. [2], can be improved through the use of liveness by giving more information to the developer in each iteration, reducing the number of times they need to go through this loop. Reis and Correia [48, 50] proposed a tool called Dockerlive, which offers an environment with a high level of

[16] Dgoss, https://github.com/goss-org/goss/tree/master/extras/dgoss

[17] Container Structure Tests, https://github.com/GoogleContainerTools/container-structure-test

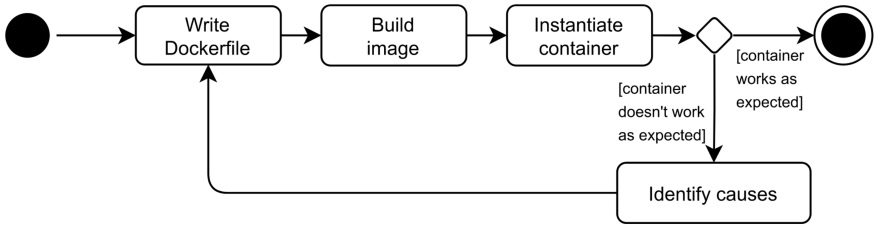

Figure 2.2: Feedback loop in Docker development
Source: [49]

liveness to those that need to edit Dockerfiles. Furthermore, Reis conducted an experiment with students to analyze the impact of this tool on Dockerfile developers. However, while our work is more focused on the developer experience, theirs was more concerned with the relationship between a developer and the IDE, as well as the effects of liveness on the actions that developers take while working on Dockerfiles. Additionally, the tool they created contains features that make it easier, for example, to detect instances of the temporary file smell, suggesting it could be useful to deal with the challenges covered in this chapter. As a result, we think that Dockerlive is a good starting point to use when creating a tool that aims to improve the Dockerfile development experience.

Unfortunately, due to time constraints, we were unable to analyze more works about liveness.

2.13 General Discussion

A few more works address additional aspects of Dockerfile development that are relevant to us but that do not fit into the categories that we covered so far.

Ksontini et al. [28] focused on refactorings and found that developers' main motivations for performing refactorings were tied to maintainability and image size, among others. Implementing some of these refactorings as repairs would be useful, although implementing all of them would be challenging.

Furthermore, Morales et al. [38] performed a systematic literature review regarding the programmer experience and provide a definition of this concept (which we covered in Section 2.1.2). Additionally, they provide several statistics about the studies that have been conducted regarding this topic. Noda et al. [40] provide a different definition of developer experience (also covered in Section 2.1.2). They also listed several examples of metrics that can be used to evaluate developer experience in an organization.

Some other works delved into making docker-related tools and artifacts easier to use [42, 43, 44], but do not focus specifically on Dockerfiles.

After all the works we analyzed, we are ready to answer the Research Questions that we listed at the start of this chapter:

- **RQ1**: What are the challenges that developers face in the development of Dockerfiles?

- **RQ2**: How have these challenges been addressed so far?
- **RQ3**: To what extent can these challenges be addressed by automatic repair or generation approaches?

Starting with RQ1, we found several challenges that are associated with Dockerfile development. Building images can take a considerable amount of time [20], which, when combined with the reliance on a trial-and-error approach, means that developing Dockerfiles can be very time-consuming [49]. Additionally, many popular images that are used as base images contain a considerable amount of bloat [1]. The bloat can also lead to security problems [3], with the number of vulnerabilities present in Dockerfiles being another challenging aspect [45]. Lastly, developers commonly violate good practices [17], leading to the prevalence of smells [59].

Moving on to RQ2, we found many approaches to deal with the challenges we mentioned. Regarding build speeds, we found approaches to accelerate them either by addressing the network bottleneck [20, 15] or skipping the need to rebuild an image [58]. Furthermore, the trial-and-error approaches can be improved through the use of liveness [50, 2] and different testing tools [29]. Additionally, it's possible to debloat containers by, for example, dividing a single container into different ones [46]. Moreover, security problems can be detected using scanners [11, 63]. Lastly, we couldn't find many approaches that address smells and good practices beyond using tools like Hadolint to detect smells [59].

With RQ3, we found some works that generate and repair Dockerfiles. However, most of the repairs we saw focused on dealing with functional problems instead of the previously mentioned challenges. As for the generation, most of the approaches had limitations that inhibited their adoption. The only data we have [34] suggests that automatic generation could actually worsen the quality of a Dockerfile. Despite this, we still believe that if we combine automatic generation with repair features targeting these challenges and offer them in an environment that provides high liveness, the development experience and the quality of the resulting Dockerfiles can improve.

Chapter 3

Designing a new version of Dockerlive

We will now describe the tool we have created in more detail. Because we used Dockerlive [50] as a starting point when building this tool, we decided to call the tool "Dockerlive V2". Like its predecessor, Dockerlive V2 is a Visual Studio Code extension. It should also be mentioned that Dockerlive's source code is available in a GitHub repository[1].

3.1 Goals

To understand the tool we have created, we must first explain the purpose behind the changes we made. We wanted to create a tool that offered the following features:

- **Repair** - the tool should be able to identify problems in a Dockerfile that the user is currently editing and suggest modifications
- **Generation** - if the user does not currently have a Dockerfile that they can work with, the tool should be able to provide a file that they can, at least, use as a starting point

We believe these features are useful to developers working with Dockerfiles and that a tool with these features can help us achieve our main goal of improving the development experience of those that need to work with Dockerfiles.

3.2 Approach

Now that we have explained what our goals are, we will describe how we will accomplish them by describing the general approach that we implemented in our tool.

Regarding the repair feature, we believe that a tool that aims to help Dockerfile developers should continuously scan the file being edited. If a problem is detected during one of these scans, then the section of the file where the problem has been detected should be highlighted using some kind of visual indicator (for example, underlining that part of the file). Furthermore, a

[1] Dockerlive, https://github.com/SoftwareForHumans/Dockerlive

user should be able to interact with the highlighted region (by using their mouse or keyboard, for example) and obtain more information about it (this information should explain what the problem is). Additionally, the tool should be able to identify a way of fixing that problem (by modifying the file) and should give the user an option to apply that fix for the user.

If a user does not have a Dockerfile, the tool should be capable of analyzing the project the user is currently working on and extracting as much information as possible from this analysis to generate a Dockerfile. Although generating an entire file requires the tool to make some choices for the user (for example, the directories where files should be placed), the user should not be bombarded with prompts asking for more information. The tool should make these decisions for the user and only ask for information when it is absolutely necessary.

3.3 Internal Design

We used Dockerlive [50] as a starting point and extended it to offer more features. The original architecture can be seen in Figure 3.1. As this figure shows, Dockerlive uses a client-server architecture[2] where the client and the server communicate using LSP (Language Server Protocol)[3]. The client interacts directly with the editor and makes requests to the server. The server can be decomposed into three main components: *dockerlive-language-server-nodejs*, *dockerlive-language-service*, and *dockerfile-utils*. This last component can be decomposed into two other sub-components: *Validator* and *DynamicAnalysis*. When the client has a file that needs to be analyzed, it sends it to the *dockerlive-language-server-nodejs* component, which then sends it to the *dockerlive-language-service* component. The file is ultimately analyzed by the sub-components of *dockerfile-utils* (*Validator* and *DynamicAnalysis*). The *DynamicAnalysis* component communicates with the Docker API[4] to be able to build images and create containers. After creating the containers and extracting data from them, it creates diagnostic information, which gets added to the *Validator* component's output. This output is sent to the client, which presents the diagnostic information to the user.

The new architecture of Dockerlive can be seen in Figure 3.2. As the figure shows, we have added new components to Dockerlive.

We have added a new *RepairScanner* component on the server side. This component scans a Dockerfile for any problems that can be repaired. If any problems are detected, they get added to the list of diagnostics produced by the *Validator*.

On the client side, a new *RepairProvider* component was added. This component analyzes each diagnostic produced by the server side and generates a corresponding repair that can be applied to the file that is currently being edited. Additionally, the client side of Dockerlive V2 can also communicate with Hermit, which can create more repair opportunities or provide the user with a Dockerfile for their project if they do not have one yet. Since the client is the component

[2]Client-server model, https://en.wikipedia.org/wiki/Client-server_model
[3]Language Server Protocol, https://en.wikipedia.org/wiki/Language_Server_Protocol
[4]Docker API, https://docs.docker.com/engine/api/

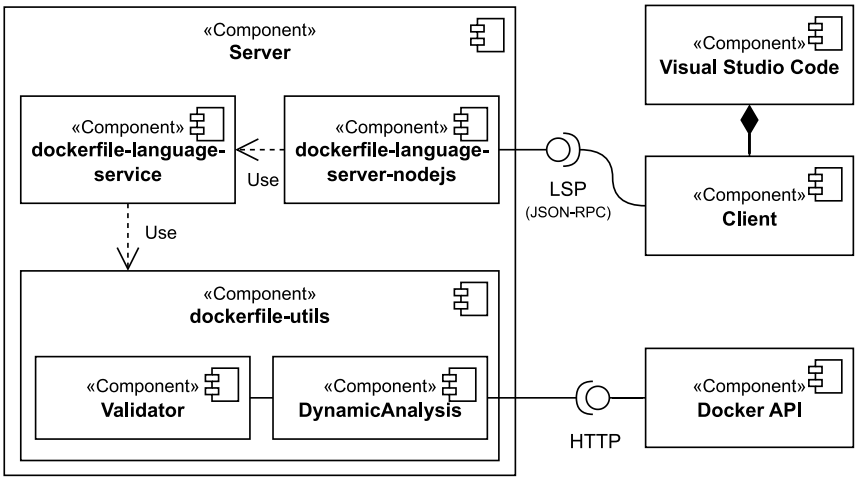

Figure 3.1: Architecture of Dockerlive V1
Source: [50]

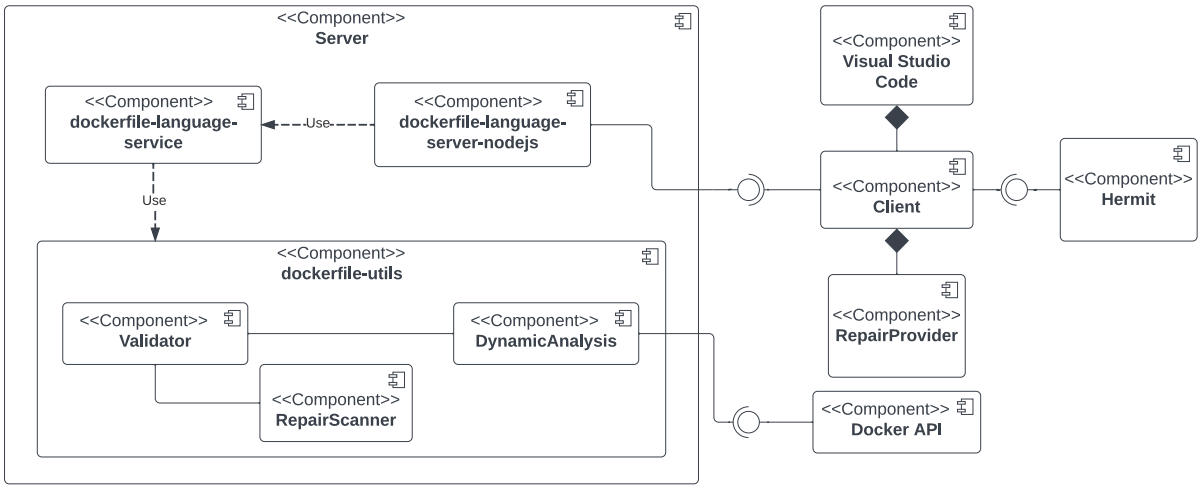

Figure 3.2: Architecture of Dockerlive V2

that a user directly interacts with and some of the functionality that Hermit provides is tied to user-facing features (which will be described in more detail in Section 3.3.2), it made sense to integrate Hermit by connecting these components instead of, for example, connecting Hermit to the server component.

More information about the repair functionality can be found in Section 3.3.1.

3.3.1 Repairs

Now that we have provided a rough overview of Dockerlive V2's architecture, we will describe each of the new components in more detail.

Starting with the *RepairScanner*, this component uses the NPM package *dockerfile-ast*[5] (which was already used by the *dockerfile-utils* component in the previous version of Dockerlive) to scan the Dockerfile and find problems that can be fixed. Depending on the problem, this component may simply check for the absence of a certain instruction, or it may need to check for the usage of specific combinations of arguments with an instruction. When a problem is detected, this component creates a diagnostic that contains a range (an object that contains data regarding the position of the file where the problem is present), a message that gives a user information about the problem that was detected, and a code that identifies the problem (for example, a diagnostic with the code R:CONFIRMINSTALL means the user forgot to add the *-y* option when installing packages with APT). In Section 3.4, there is a list of all the repairable problems and the conditions that lead to the detection of each one of them.

As for the *RepairProvider*, this component receives the list of diagnostics produced by *RepairScanner* and generates the corresponding repair for each diagnostic. The most important components of a repair are the range where the modification should be performed (a range may span one or more instructions, or even a part of an instruction, depending on the diagnostic) and the text that should replace the one currently contained in that range. For example, if the user forgot to add the *-y* option when installing packages with APT, the range will contain the *apt-get install* keywords, and the replacement text will be *apt-get install -y*. Additionally, repairs have a small message that describes what they do.

These components may also use the information provided by Hermit. Section 3.3.2 will provide more details regarding Hermit's usage.

3.3.2 Hermit

One of the improvements that Dockerlive V2 includes is integration with Hermit. Dockerlive and Hermit are still independent but, when Dockerlive needs to use Hermit, it calls it using Hermit's CLI (Command-line Interface). We found this to be the simplest and most stable way of connecting Hermit and Dockerlive.

Hermit is useful to Dockerlive in two different ways:

[5] dockerfile-ast, https://www.npmjs.com/package/dockerfile-ast

- When a project does not have a Dockerfile, Hermit can provide one
- If the project already has a Dockerfile, Hermit can still use information from this file to generate an alternative Dockerfile, which is then compared with the original one to find more repair opportunities (as described in the previous section)

Regarding the first scenario, we first need to mention that Hermit has two modes: one where the Dockerfile is generated from scratch and another where it uses information from an existing file to generate an alternative one. Both modes perform dynamic analysis to gather the information required to generate a file. However, while the first mode performs dynamic analysis while running the service natively, the second one does it while running the service inside a container (this container is generated using the information from the existing file). For these reasons, we will refer to the first mode as "native mode" and the second one as "container mode".

With this in mind, when no Dockerfile is present, Dockerlive actually executes Hermit twice. The first execution generates a new file (using the native mode), and the second one generates an alternative file using the first one as a reference (this is the container mode). This is done because we found that the output of the container mode is often better and results in a Dockerfile that contains less bloat. For example, when generating a Dockerfile for a simple JavaScript project that just prints the text "Hello World!", the native mode would generate a Dockerfile that installs many dependencies which are not actually needed. This did not happen with the container mode, leading us to believe that the detection of dependencies is more accurate when using the container mode.

As for the repairs enabled by Hermit, they will be described in more detail in Section 3.4. They revolve around port exposure and the installation of dependencies (both using distributions' package managers and languages' package managers). When the ports or dependencies detected in the current file do not match the ones in the file generated by Hermit, Dockerlive will suggest repairs that update the current file to make it more similar to the one generated by Hermit.

3.4 Repair Implementation Details

This section contains a list of all the repairs that have been implemented:

- **Add image tag** - If the base image of the Dockerfile does not contain a tag, then Dockerlive will provide one (either *18-slim* for Node.js projects or *3.11-slim* for Python projects, other types of projects are not supported). Using an image tag can improve the speed, stability, and security of a container [41, 59].
- **Use the WORKDIR instruction** - If the WORKDIR instruction is not used in the file, then Dockerlive will suggest adding the line *WORKDIR /app* at the start of the file after the FROM instruction. This keeps the directory tree more organized by having a dedicated place for a project's files [53].

- **Change the exposed ports** - If the current Dockerfile does not expose any ports (or exposes the wrong ones), Dockerlive will offer a repair that changes the ports being exposed. Hermit is used to detect the ports that should be exposed. Having the correct set of ports exposed in the file improves the project's maintainability.
- **Use the *-y* option with *apt-get install*** - If the current file attempts to install packages with APT without using the *-y* option, Dockerlive will suggest adding this option. Without this option, the *apt-get install* command will fail [17].
- **Use the *apt-get update* command before *apt-get install*** - If the current file attempts to install packages with APT without running *apt-get update* before, Dockerlive will suggest adding this command before. If the update command is not executed before, the package manager's cache will be outdated, which can cause problems [17, 18].
- **Use the *–no-install-recommends* option with *apt-get install*** - If the current file attempts to install packages with APT without using the *–no-install-recommends* option, Dockerlive will suggest adding this option. Without this option, APT may install packages that are not needed and waste space [17, 59].
- **Change the packages being installed** - If the packages being installed in the current file using the system's package manager do not match the ones detected by Hermit (this also includes situations where Hermit detects some dependencies but the current file does not install any or vice versa), Dockerlive will offer a repair that changes the packages being installed. This can fix functional problems if some of the required packages are missing or reduce the amount of wasted space if unnecessary packages are being installed.
- **Add command to remove APT lists** - If the current file installs packages using APT but does not remove the list of packages afterward, Dockerlive will suggest adding a command to remove them. After installing packages, APT's cache (the list of packages and their versions) becomes useless; removing it reduces the amount of space that is wasted [17].
- **Use the *–no-cache* option with *apk add*** - If the current file attempts to install packages with APK without using the *–no-cache* option, Dockerlive will suggest adding this option. This option keeps the package manager from caching information on the disk, reducing the amount of space that is wasted [17, 18].
- **Merge consecutive RUN instructions** - If two consecutive RUN instructions are detected in the current file, Dockerlive will offer a repair that merges them into one (connecting the commands using *&&*). Having fewer instructions in a Dockerfile reduces the number of layers that Docker needs to process and can shorten build times [6, 45].
- **Replace an isolated *cd* with WORKDIR** - If a RUN instruction that only contains a *cd* command is detected, Dockerlive will suggest replacing it with a WORKDIR instruction. The working directory is not preserved between instructions, making an isolated *cd* useless [59].

- **Use the *-f* option with curl** - If a RUN instruction containing a usage of the curl command without the *-f* option is detected, Dockerlive will suggest adding this option. When curl commands fail, they generate some output that can cause problems. Adding this option keeps the output from being generated [17].
- **Update URLs to use HTTPS** - If Dockerlive detects that HTTP URLs are used with curl or wget commands, it will offer a repair that changes the URLs to use HTTPS instead. HTTP is less secure than HTTPS. Therefore, updating the URLs to use HTTPS can improve security [17]. However, Dockerlive does not check if the updated URL is valid or a response is returned from it, which can cause errors in some situations.
- **Replace ADD with COPY** - If ADD instructions are detected, Dockerlive will suggest replacing them with COPY instructions. ADD instructions can have some unwanted side effects, making COPY instructions safer to use [45].
- **Remove MAINTAINER instructions** - If MAINTAINER instructions are detected in the current file, Dockerlive will suggest removing them. MAINTAINER instructions have been deprecated, which means they should no longer be used [59].
- **Use multiple COPY instructions** - If a single COPY instruction is detected in the current file, Dockerlive will suggest adding a second one and dividing the files being copied into two groups; the first group will only contain the files required to install packages using a language's package manager (*package.json* and *package-lock.json* for Node.js projects and *requirements.txt* for Python projects, other types of projects are not supported) while the second group will contain the rest of the files that are used by the project. This promotes layer caching by avoiding the need to rebuild layers that are only used to install dependencies [53, 41, 45].
- **Use the USER instruction** - If the USER instruction is not used in the current file, then Dockerlive will suggest adding it near the end of the file, before the last instruction (for Node.js projects, the recommended user will be *node* while for Python projects it will be *python*, with an execution of the *useradd* command being added before to create this user, other types of projects are not supported). Running applications as the *root* user can be dangerous if the application contains unknown vulnerabilities; changing users addresses this problem [53, 45].
- **Install dependencies with the language's package manager** - If the current Dockerfile does not contain a RUN instruction that uses the language's package manager to install dependencies, Dockerlive will offer a repair that adds this command. For Node.js projects, the command will be *npm install*; for Python projects, it will be *pip install*; other types of projects are not supported. Hermit provides the information used for this repair. These commands are important to ensure that the required dependencies are installed.

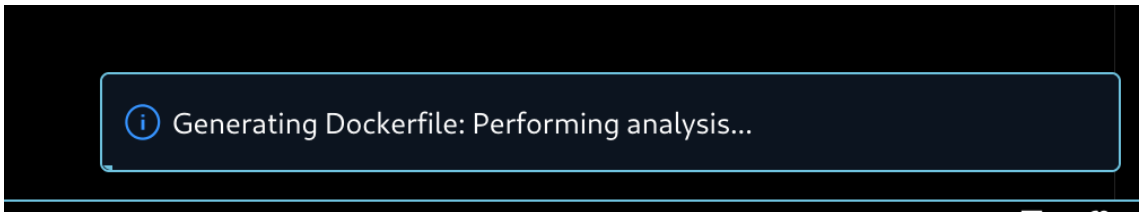

Figure 3.3: Generation progress notification

3.5 Hermit Contributions

Besides establishing a connection to Dockerlive, we also made some small improvements to Hermit. Firstly, we updated the versions of the images that were used in the generation process to more recent ones. Secondly, the generated files would often use the ADD instruction; we replaced them with COPY instructions (this matches one of the repairs mentioned in the previous section). Finally, Hermit will execute a dpkg[6] command while detecting dependencies; this command can only be used in Debian-based[7] Linux distributions. We added the ability to execute this command inside a container, increasing the number of systems where Hermit can be used. Hermit's source code is also available in a GitHub repository[8].

3.6 User Interface

Now that we have taken a look at Dockerlive V2 from an internal perspective, we will describe it from a user's perspective.

The new features that a user will see in Dockerlive V2 are the repairs and the ability to generate a Dockerfile that does not have one yet.

All the screenshots shown in this Section were taken using Visual Studio Code with a high-contrast theme to make it easier to distinguish UI (User Interface) elements.

3.6.1 Repairs

Starting with the repairs, when the user opens a new Dockerfile, because some of the repairs rely on the alternative file generated by Hermit, the user will have to wait for Hermit's generation to finish before gaining access to the diagnostic information. A notification that monitors the generation's progress will be shown in the bottom right corner of the application. Figure 3.3 shows this notification while the generation is in progress, while Figure 3.4 shows the notification after the generation is complete.

Once the generation is complete, if Dockerlive detects a problem that can be repaired, it creates a warning in the region of the file that corresponds to that problem (file regions are internally referred to as "ranges", as mentioned in previous sections). Figure 3.5 shows an example of a

[6] dpkg, https://en.wikipedia.org/wiki/Dpkg
[7] Debian, https://www.debian.org/
[8] Hermit, https://github.com/SoftwareForHumans/Dockerlive

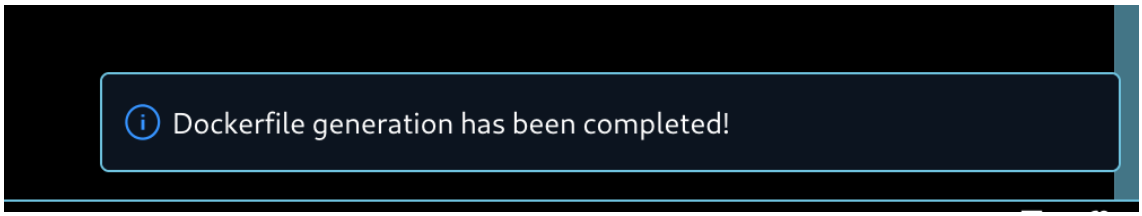

Figure 3.4: Notification showing the generation is finished

Dockerfile with repairable problems, while Figure 3.6 shows an example of a warning for one particular repairable problem.

As the image shows, a section containing a warning has a yellow underline. If the user then hovers over that part of the file, some information is shown about the problem. At the bottom of this information box, there is also a button related to quick fixes. Clicking this button shows quick fixes related to this part of the file. Figure 3.7 shows the quick fix menu.

After choosing the desired fix, a user simply has to click it to apply the fix. Figure 3.8 shows the state of the Dockerfile after applying a particular fix.

Some warnings that are associated with repairs that insert new instructions in the file may underline sections of the file that do not have any text. This is because the problem is the absence of an instruction, and the warning is placed in a common location for that instruction as a suggestion for the user. Figure 3.9 shows an example of one of these warnings.

3.6.2 Dockerfile Generation

Moving on to the Dockerfile generation, this can be done by opening Visual Studio Code's command palette and selecting the Dockerfile generation command. Figure 3.10 shows this command in the palette.

After selecting this command, the user will be prompted for the command that runs the service (this is one of Hermit's limitations, Hermit can not deduce the command that is used to start

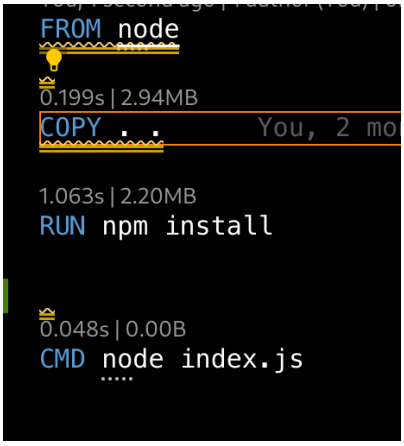

Figure 3.5: Dockerfile with repairable problems

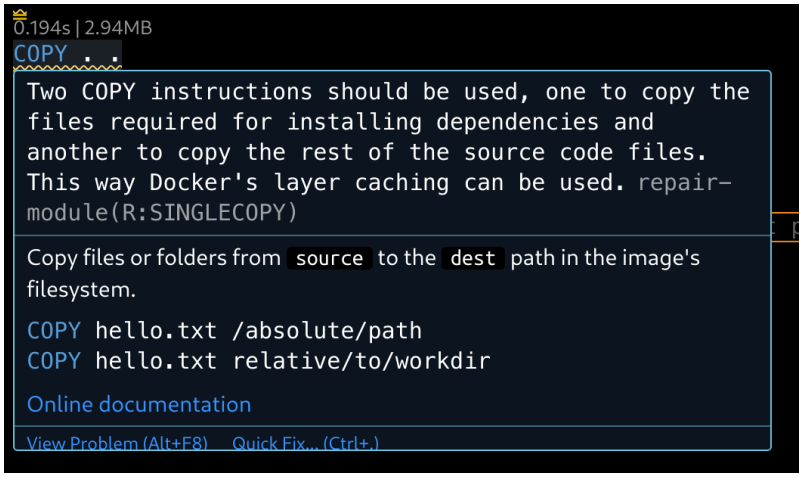

Figure 3.6: Repairable warning in Dockerlive V2

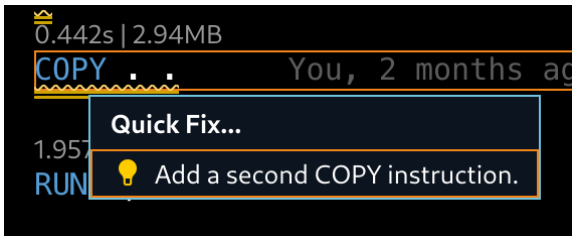

Figure 3.7: Quick fix in Dockerlive V2

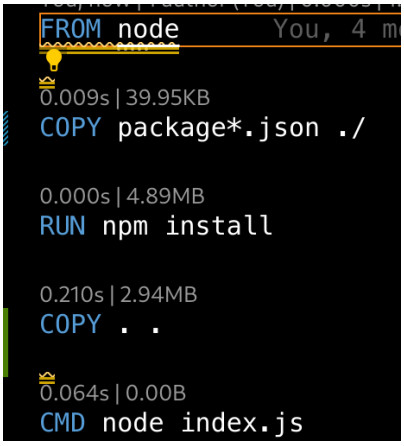

Figure 3.8: State of the example file after applying a fix in Dockerlive V2

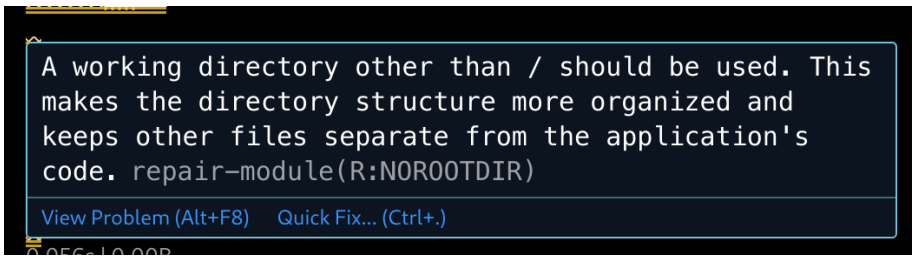

Figure 3.9: Empty line containing a warning in Dockerlive V2

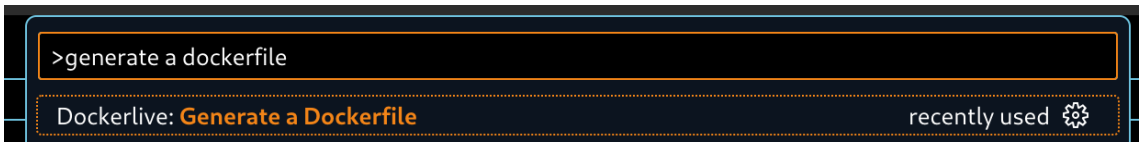

Figure 3.10: Generation command in Visual Studio Code's command palette

the service and requests it from the user). It should be noted that despite also being called a "command", this command is not the same as one of the commands present in Visual Studio Code's palette. This command is the text that would have to be inserted in a terminal to run the service (for example, *node index.js* to run a JavaScript file called *index.js*). After typing the command and pressing *Enter*, the generation will begin and a notification similar to the one in Figure 3.3 will be shown in the bottom right corner of the application. Once the generation has been completed, a notification similar to the one shown in Figure 3.4 will be displayed.

3.7 General Discussion

Overall, we would have liked to improve Dockerlive more. In its current state, there are a few aspects that could be improved:

- **Potentially undesirable repairs** - some of the repairs suggested by Dockerlive may not be desirable due to unintended effects despite being capable of fixing the issues that are detected
- **Reliance on components used by the official Docker extension**[9] - besides the fact that these components are meant to be used with a different extension, the versions of these components that Dockerlive uses are also quite old, which could cause problems in the future
- **Suboptimal integration with Hermit** - having Hermit as an external dependency not only makes the projects harder to maintain but also makes the extension's installation and initial setup more difficult for end users
- **Poor implementation** - internally, the new repair components rely heavily on basic string manipulations to perform their tasks, which affected their stability during development

[9] VS Code Docker Extension, https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker

Figure 3.11: Service command prompt

- **Poor user experience** - some of the new features could be considered unintuitive (for example, the warnings that show up in empty lines)

With this in mind, we came up with the following list of potential improvements and problems associated with them:

- **Use data from public pull requests**[10] **to suggest repairs** - To make sure the repairs the Dockerlive recommends are desirable, we could look at data from public pull requests (found on GitHub) to see how developers fix certain issues and use this data to provide recommendations in Dockerlive. This would require a substantial amount of time, but we believe it could improve the quality of Dockerlive's suggestions.
- **Reduce the reliance on the official extension's components** - This would likely require most of the server component's code to be rewritten and restructured, which would require a significant amount of time. Unfortunately, we do not believe any alternatives allow us to circumvent this problem.
- **Improve the integration with Hermit** - Initially, we tried to merge Hermit and Dockerlive into a single codebase and perform refactorings to reduce the amount of duplicated code. However, the end result was unstable and Hermit behaved abnormally. If someone were to attempt to do this again in the future, we believe they could run into similar issues. Perhaps, with more time, they could fix all the problems we found and create a more robust integration.
- **Make the repairs' implementation more robust** - As mentioned, the current implementation of the repairs uses very rudimentary string manipulation and parsing techniques. This could be alleviated by using packages that aid with string parsing, such as *mvdan-sh*[11] to parse Bash[12] code contained in the Dockerfiles. Using a package to parse the Dockerfiles on the client component would also be helpful. Unfortunately, using *dockerfile-ast* on the client side creates problems because the types used by this package are only compatible with the server side. The type declarations used by the server[13] and client components are incompatible with each other despite sometimes sharing names. After finding packages that can be used for parsing, it may still take some time to refactor the code to make use of them.
- **Improve the user experience** - Perhaps, instead of using empty lines to tell the user they should add something to the file, a notification could be used to provide this information. Furthermore, an option could be provided to allow a user to apply all the repairs at once instead of having to apply them one by one. With enough time, these improvements should be relatively easy to implement.

[10] Pull requests, https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests
[11] mvdan-sh, https://www.npmjs.com/package/mvdan-sh
[12] Bash, https://www.gnu.org/software/bash/
[13] vscode-languageserver-types, https://www.npmjs.com/package/vscode-languageserver-types

Despite everything that has been said, we still believe the new features are effective and help users create better Dockerfiles. The usefulness of the new features is discussed in more detail in Chapter 4.

Chapter 4

Empirical Study

Now that we have described the tool we created, we will describe the empirical study that was performed to evaluate its impact on the Dockerfile development experience.

4.1 Goals

As we have stated multiple times, our main goal is to improve the developer experience of developers working with Dockerfiles. As we documented in Chapter 2 there are many issues that affect Dockerfile developers and that need to be addressed.

The data presented in Chapter 2 also explains our secondary goal: to improve the quality of the Dockerfiles written by developers. It is important to address the multitude of problems that can be found in Dockerfiles as the consequences of their existence can be detrimental.

4.2 Research Questions

To understand our research questions, it should again be noted that this is our hypothesis:

The Dockerfile developer experience and the quality of the resulting images can improve if the environment provides a high level of liveness regarding the most challenging aspects of Dockerfile development

We believe that an environment with high liveness can help us achieve our goals and we implemented features in our tool that promote this kind of environment. With this in mind, as well as the information presented in the previous section, these are the research questions that we will try to answer with our study:

- **RQ1**: To what extent does Dockerlive decrease cognitive load?
- **RQ2**: To what extent does Dockerlive promote a flow state?
- **RQ3**: To what extent does Dockerlive shorten feedback loops?

- **RQ4**: To what extent does Dockerlive enable developers to create higher-quality Dockerfiles?

The first three questions will help us determine if Dockerlive can actually improve the developer experience of those working with Dockerfiles. The last research question focuses on our secondary goal: improving the quality of the Dockerfiles created by developers.

4.3 Methodology

To answer the research questions outlined in the previous section, an experiment was conducted. The experiment followed a methodology that is an evolution of the one used by Reis [50] to evaluate Dockerlive when it was first created. Like Reis, our study gathers data from participants that perform tasks involving Dockerfiles. However, while Reis gathered all their data while the participants were performing the tasks, we gathered ours from recordings of the participants' performances, allowing us to be more accurate. Additionally, while Reis only collected data about context switching and time, we also collected data regarding the Dockerfiles' quality. Furthermore, Reis' study was conducted with students, while ours was conducted with professionals. We chose to use an experiment (instead of other types of studies) as it allowed us to have more control over the environment the participants were exposed to and better analyze the effects of our tool. The rest of this section describes our experiment in more detail.

The experiment has two groups: a control group and an experimental group. Both groups have to carry out the same tasks. The control group does not have access to any special tools (this includes other editor extensions or external tools like ChatGPT[1]) while performing the tasks while the experimental group has access to Dockerlive. The control group is not given access to any tools since most developers do not seem to use any tools when working with Dockerfiles [49]. The participants carry out the tasks while on a call with an observer, a person that provides instructions, clears up doubts, sets up the environment, and monitors their progress.

The data from both groups is then compared and analyzed (Section 4.7 covers this in more detail).

Several of the tools mentioned in this section can be found on a GitHub repository[2]. The PDF files found on that repository are created using Markdown[3] and an extension[4] that generates PDF files from Markdown. PDF files are also merged using pdfunite[5].

4.3.1 Data Collection

While performing the tasks, the participants are timed, not only to measure the amount of time that is spent but also to make sure the time limit is not exceeded. Furthermore, the participants' actions

[1] ChatGPT, https://openai.com/chatgpt
[2] dockerlive-study, https://github.com/matosjoaops/dockerlive-study
[3] Markdown, https://en.wikipedia.org/wiki/Markdown
[4] Markdown PDF, https://marketplace.visualstudio.com/items?itemName=yzane.markdown-pdf
[5] pdfunite, https://manpages.ubuntu.com/manpages/jammy/man1/pdfunite.1.html

are recorded. This recording is used to analyze the amount of context switching that is performed. The recordings are also used to obtain more accurate time measurements (regarding task duration as well as the amount of time spent in each context). Data is also gathered from questionnaires that aim to capture a participant's perception of their performance.

We will now explain how this data helps us answer our first three research questions. As stated in Section 4.2, our first three questions address the dimensions mentioned by Noda et al.: cognitive load, flow state, and feedback loops [40]. Since these concepts are quite complex and hard to measure directly, we are using proxy metrics. The proxy metrics we have chosen are *time* and the amount of *context switching* that is performed.

Cognitive load is "the amount of mental processing required for a developer to perform a task" [40]. The lower the *cognitive load* the better. One of the ways to reduce the *cognitive load* suggested by Noda et al. is "to reduce the amount of context or task switching required" [40]. Furthermore, Mark et al. mention that interruptions (getting interrupted involves *context switching*) force a person to put in more effort to finish a task [36] (this would increase the *cognitive load*). For these reasons, we believe that the number of *context switches* is a useful proxy metric when trying to assess the *cognitive load* of a task. Additionally, some of the questions in the questionnaires also measure how much a user struggled with a particular task. This is also useful given that more difficult tasks are associated with higher *cognitive loads* [40].

Regarding the *flow state*, this is a state where a person can be more productive. However, being interrupted multiple times while trying to perform a task makes it more difficult for an individual to enter or remain in this state [40]. When someone is interrupted they have to perform *context switching*. Therefore performing less *context switching* should help an individual who seeks to enter or remain in this state. By measuring the amount of *context switching* that is performed we can judge how easy or difficult it would be to make use of the *flow state* when carrying out a task.

In previous sections, we have already discussed how tightening *feedback loops* can improve a developer's experience. A tighter *feedback loop* means that it takes less *time* for the developer to get feedback and act based on the information they receive. As a result, we believe that measuring the *time* it takes to carry out a task allows us to verify if *feedback loops* are being tightened.

Finally, the Dockerfiles produced by the participants are also stored and analyzed to obtain some data regarding their quality. This addresses the fourth research question that was listed.

4.3.2 Tasks

Both groups have to perform three tasks (described in Section 4.4). In these tasks, the participants have to edit a Dockerfile according to the instructions that are provided. Each task has a different objective that corresponds to a different Dockerfile quality metric (for example, image size) and showcases some of Dockerlive's repairs (some of the repairs that are showcased also use the generation features). The tasks are also designed to have enough complexity to be able to showcase differences in the metrics described in the previous section (although they are still relatively simple to avoid requiring too much time from the participants).

4.3.3 Environment

The study is conducted remotely using AnyDesk[6] (with the exception of one participant where we were forced to use TeamViewer[7] instead). The remote machine is running an operating system based on Ubuntu[8].

The environment is different for both groups. While both groups have access to Visual Studio Code, only the experimental group has access to Dockerlive. In addition to this extension, the official Python extension[9] for Visual Studio Code is also preinstalled for both groups since some of the tasks include Python files (although the participants are not allowed to edit them).

4.3.4 Procedure

Each participant begins by joining a Google Meet[10] meeting held by the observer. After a short introduction, the participant connects to the remote machine using AnyDesk (or TeamViewer). They then start reading the instructions stored in a file called "instructions.pdf" (generated using a script[11]) that is stored on the remote machine's desktop. This file contains several pieces of information:

- A link to a questionnaire the participant has to fill out throughout the study
- For the experimental group, instructions regarding Dockerlive's usage[12]; the control group is given access to a cheat sheet[13] with information about Docker's CLI (Command-line Interface)
- Some general rules and instructions[14]

The general rules are listed below:

- *In each task you will be asked to edit a Dockerfile according to the instructions that are provided.*
- *Once you think you have accomplished what was requested for that task, notify the observer.*
- *You should **only edit the Dockerfile**.*
- *Feel free to consult this document at any time.*

[6] AnyDesk, https://anydesk.com
[7] TeamViewer, https://www.teamviewer.com
[8] Ubuntu, https://ubuntu.com/
[9] Python - Visual Studio Marketplace, https://marketplace.visualstudio.com/items?itemName=ms-python.python
[10] Google Meet, https://apps.google.com/meet/
[11] setup.sh, https://github.com/matosjoaops/dockerlive-study/blob/main/setup.sh
[12] Dockerlive instructions, https://github.com/matosjoaops/dockerlive-study/blob/main/dockerlive-instructions/instructions.pdf
[13] CLI Cheat Sheet, https://docs.docker.com/get-started/docker_cheatsheet.pdf
[14] General instructions, https://github.com/matosjoaops/dockerlive-study/tree/main/general-instructions

- *You have a maximum of 20 minutes to complete each task.*
- *You are allowed to inspect the source code of the scripts in each task.*
- *You are allowed to use a web browser to look for information while performing the tasks. However,* ***you are not allowed to use tools and applications besides those present on the remote machine, do not use tools present on your computer unless explicitly stated, for example, if you need to use a web browser, use the one that is installed on this machine instead of using one that is installed on yours.***
- ***You are not allowed to modify the source code of the scripts in each task.***
- ***You must use Visual Studio Code to edit the Dockerfile.***
- ***You may not install other Visual Studio Code extensions or modify the editor's configuration.***
- ***You may not install other programs on this computer.***
- *All the information you need to solve the tasks is present in this document.*
- *If there is something you do not understand while reading this document, notify the observer immediately.*

The general instructions tell users where the folders for each task can be found (the folders are placed on the desktop) and to alert the observer when starting or finishing a task. Additionally, the general instructions tell the participants to fill out the corresponding section of the questionnaire after each task and to read the instructions for each task carefully.

While the participants are performing the tasks, the observer is keeping track of the time to make sure they do not go over the limit (and also because time is one of the pieces of data that we want to collect). The observer can also answer questions to clear up any doubts the participants may have. The observer's answers are as neutral as possible to avoid influencing the participants' behavior. In between tasks, the observer also runs a script[15] to reset the development environment by clearing any Docker-related data, as well as Visual Studio Code's history. Additionally, Firefox is configured to never store any history. This is done to make sure a participant can not be influenced by another participant's performance or their own performance in a different task.

4.4 Tasks

In this experiment, users had to perform three tasks where they edited Dockerfiles according to the instructions that they were given.

4.4.1 Task 1

In this task, participants were given the following files:

[15] reset.sh, https://github.com/matosjoaops/dockerlive-study/blob/main/reset.sh

- Dockerfile (shown in Listing B.1) - The Dockerfile for this project and the file that must be edited. It copies the Python file into the container and uses *pip* to install the dependencies that are required. It also installs HTTPie[16] and uses it to download an HTML file from `http://example.com`. Finally, it runs the Python script.
- main.py - A simple script that parses the HTML file using Beautiful Soup[17] and prints some text.
- requirements.txt - A file containing the text "beautifulsoup4".

Furthermore, the task had some restrictions:

- HTTPie can not be replaced with a different utility (such as curl or wget).
- The HTML file that is downloaded can not be replaced with a different file.
- Only the Dockerfile can be edited.

The Dockerfile that is provided is already functional. The objective of this task is to make the resulting image smaller. The task ends when the participant thinks they cannot shrink the image anymore. There are multiple ways to reduce the size of a Docker image, such as using a smaller base image, making sure the system's package manager does not store a cache, and also avoiding the installation of packages that are not strictly necessary. The methods listed are part of the repairs that Dockerlive can offer but there are other ways of reducing an image size that a participant could choose to implement.

The files for this task can be found on the repository, in the directory "task-1"[18].

4.4.2 Task 2

In this task, participants were given the following files:

- Dockerfile (shown in Listing B.2) - The Dockerfile for this project and the file that must be edited. It copies the JavaScript file and the text files into the container. One of the files is copied into the current working directory while the other one is copied specifically into the root of the filesystem.
- file.js - A JavaScript file containing code that attempts to read and print the contents of the text files stored in the root of the filesystem. If it is unable to open a file, it will instead print the text "Could not get the data."
- important_info.txt - A text file containing the text "The password is: OPEN_SESAME". The file's permissions were set to 0700[19].

[16] HTTPie, `https://httpie.io/`
[17] Beautiful Soup, `https://pypi.org/project/beautifulsoup4/`
[18] task-1, `https://github.com/matosjoaops/dockerlive-study/tree/main/task-1`
[19] Filesystem permissions - Numeric notation, `https://en.wikipedia.org/wiki/File-system_permissions#Numeric_notation`

- important_info2.txt - A text file identical to the previous one but with a different name. The permissions are also the same.
- package.json - A JSON (JavaScript Object Notation) file containing data about the project.

Furthermore, the task had some restrictions:

- The COPY instructions could not be removed and their arguments could not be changed.
- The content of the text files could not be changed and they could not be replaced with other files.
- The files' permissions cannot be changed.
- Only the Dockerfile can be edited.

The Dockerfile that is provided is already functional. However, the JavaScript file is capable of accessing the data in the text files. The objective is to make it unable to access the data in those files. Using the WORKDIR instruction would change the working directory making the code unable to access one of the files. The USER instruction could also be used to change the current user to one that does not have the permissions required to open the files. Both of these practices are included in the repairs that Dockerlive can recommend.

The files for this task can be found on the repository, in the directory "task-2"[20].

4.4.3 Task 3

In this task, participants were given the following files:

- Dockerfile (shown in Listing B.3) - The Dockerfile for this project and the file that must be edited. It installs Node.js, copies the JavaScript file into the container and tries to run it.
- index.js - A JavaScript file that starts an Express[21] server on port 3000 and prints the text "Listening on port 3000...".
- package.json and package-lock.json - JSON files containing data about the project.

Furthermore, the task had some restrictions:

- Only the Dockerfile can be edited.

The Dockerfile that is provided is not functional. This is because the *npm install* command is missing and the Express NPM package[22] is not being installed. The objective of the task is to make sure the correct set of dependencies is being installed. This means the *npm install* command should be added. The participant should also remove the command that installs Node.js using the

[20] task-2, https://github.com/matosjoaops/dockerlive-study/tree/main/task-2
[21] Express, http://expressjs.com/
[22] express - npm, https://www.npmjs.com/package/express

system's package manager given its redundancy (the base image already includes Node.js), but this second change is optional. Both of the changes can be implemented using repairs suggested by Dockerlive. Some of these repairs can use information provided by Hermit.

The files for this task can be found on the repository, in the directory "task-3"[23].

4.5 Data Collection

Throughout the study, data was collected from several sources. Data was collected from the tasks the participant carried out and from the Dockerfiles they produced as a result of those tasks. Furthermore, participants had to fill out two questionnaires, one during the recruitment process and another while performing the tasks.

The data we obtained and the tools we used to obtain and process this data can be found on a GitHub repository[24]. It should be noted that some of the participants' replies to the questionnaires were written in Portuguese. We decided not to translate the answers to avoid misinterpretations and the introduction of any bias.

4.5.1 Tasks

During each task, a simple timer[25] was used to keep track of the amount of time participants spent on a task. Furthermore, the actions performed by participants on the remote machine were recorded (for most participants AnyDesk's built-in recording feature[26] was used; when TeamViewer had to be used, OBS (Open Broadcaster Software)[27] was used to record the participant's actions) and later analyzed to measure the amount of context switching that was performed. This also allowed us to calculate the amount of time spent in each context. While watching a recording, a timestamp was written down every time the participant switched from one context to another. Figure 4.1 shows the timestamps written down using a note-taking application[28].

These notes were manually converted into a CSV (Comma-separated values) file, containing the same information (Figure 4.2 shows an example of one of these files). One CSV file was used per task per participant.

After having the information in a CSV file, a script[29] was used to parse the timestamps. This script receives as a command-line argument the ID (Identification) of a participant and adds a line to a different CSV file[30]. The line added contains all the information about that participant's performance.

[23] task-3, https://github.com/matosjoaops/dockerlive-study/tree/main/task-3
[24] dockerlive-study-data, https://github.com/matosjoaops/dockerlive-study-data
[25] Timer, https://flathub.org/apps/com.github.vikdevelop.timer
[26] Session Recording, https://support.anydesk.com/knowledge/session-recording
[27] OBS, https://obsproject.com/
[28] Obsidian, https://obsidian.md/
[29] parse_timestamps.py, https://github.com/matosjoaops/dockerlive-study-data/blob/main/scripts/parse_timestamps.py
[30] participants.csv, https://github.com/matosjoaops/dockerlive-study-data/blob/main/anon-data/participants.csv

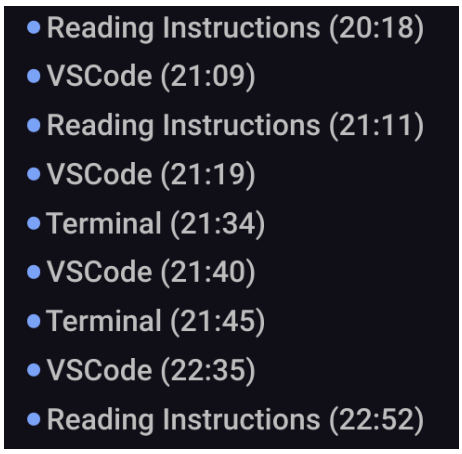

Figure 4.1: Timestamps written in a note-taking application

```
Context,Timestamp
Reading Instructions,20:18
VSCode,21:09
Reading Instructions,21:11
VSCode,21:19
Terminal,21:34
VSCode,21:40
Terminal,21:45
VSCode,22:35
Reading Instructions,22:52
```

Figure 4.2: Timestamps written in a CSV file

Several contexts were taken into consideration when analyzing the footage:

- **Reading Instructions** - includes time spent reading the tasks' (or Dockerlive's) instructions, also includes time spent reading the cheat sheet
- **Firefox** - includes time spent in the Firefox browser
- **Terminal** - includes time spent in a terminal (also includes Visual Studio Code's built-in terminal)
- **VSCode** - includes time spent in Visual Studio Code, does not include time spent in the built-in terminal
- **File Manager** - includes time spent in a file manager (this was used to, for example, navigate to a task's directory and open Visual Studio Code there)
- **File Editor** - includes time spent in a file editor (by default, if a user double-clicks a text file, the operating system will open it with gedit[31], some participants accidentally opened a file this way before switching to Visual Studio Code)
- **Excluded** - not a real context; due to some technical issues that occurred while performing the study with some of the participants, time had to be subtracted from the total; this context was used to tell the script that time had to be subtracted

All the timestamps recorded can be found on the "anon-data/individual-tasks" directory[32] on the GitHub repository related to this section.

4.5.2 Dockerfiles

After performing the tasks, a participant's Dockerfiles were copied to a separate directory using a script[33]. For each Dockerfile, we checked if an image could be built successfully. Additionally, the image was executed. However, depending on the task, the definition of a "successful" execution was different. For Task 1, this meant that the printed text was identical to the one being printed when using the original Dockerfile that was given to the participants. For Task 2, this meant that the data in the text files could not be accessed. For Task 3, this meant that the text "Listening on port 3000..." was printed.

Moreover, for Task 1, we measured the size of the image. For Tasks 2 and 3, simple notes were taken regarding the changes performed to the files (for example, in Task 3, one of the notes could say "Added npm install"). We chose this approach because it allowed us to focus on specific aspects of the Dockerfile. Other tools (for example, a linter) would have provided too much information.

[31] gedit, https://wiki.gnome.org/Apps/Gedit

[32] anon-data/individual-tasks, https://github.com/matosjoaops/dockerlive-study-data/tree/main/anon-data/individual-tasks

[33] grab_dockerfiles.sh, https://github.com/matosjoaops/dockerlive-study/blob/main/grab_dockerfiles.sh

All this data was manually added to a file[34] that can be found on the repository related to this section. The Dockerfiles can be found in the "anon-data/Dockerfiles" directory[35].

4.5.3 Recruitment Questionnaire

During recruitment, we asked people who might be interested in participating in the study to fill out a questionnaire (shown in Section A.1).

This questionnaire begins by providing a brief explanation of the study's purpose and the conditions in which it will take place. Afterward, it asks the user if they are willing to participate in the study.

If the user answers positively, they are then asked to fill out another section where they have to provide their name, email address, and current job title.

The final section of this questionnaire aims to measure the amount of experience a user has in the field of software development and, more specifically, how much experience they have developing Dockerfiles. This section also asks the user if they have ever used a Visual Studio Code extension to edit Dockerfiles. This was done to make sure none of the participants had experience with a previous version of Dockerlive.

4.5.4 Task Questionnaire

Besides the recruitment questionnaire, those who actually participated in the study had to fill out an additional questionnaire. Both groups had to fill out the questionnaire but the versions used for each one were slightly different (the control group's version is shown in Section A.2 while the experimental group's version is shown in Section A.3).

The first section simply asks for the participant's email address and name.

The second section aims to figure out how much experience the participant has with the tools and technologies they will use during the study. This is done to verify if their performance is affected by their lack of experience with the tools they had to use.

While the first two sections could be filled out before the participant had started to carry out the tasks, the next three sections asked questions about each task and, therefore, could only be filled out after completing the corresponding task. These sections asked the participant if they were able to understand the instructions and if they were able to finish the task without any problems. For the experimental group, there was also a question regarding Dockerlive's helpfulness while solving the task.

The following sections were filled out after the participant completed the tasks.

The sixth section simply asked the user how comfortable they were with the remote environment.

[34] dockerfiles.csv, https://github.com/matosjoaops/dockerlive-study-data/blob/main/anon-data/dockerfiles.csv

[35] anon-data/Dockerfiles, https://github.com/matosjoaops/dockerlive-study-data/tree/main/anon-data/Dockerfiles

The seventh section asked the user if they felt the tasks took a long time to carry out and if they felt they could have done a better job if they had access to other tools (they were also asked to list any tools that could have helped them).

The experimental version of the questionnaire had an additional section asking for feedback regarding Dockerlive.

The questions in these questionnaires used a Likert scale[36] [30, 25] with five options ranging from "Strongly Disagree" to "Strongly Agree".

4.6 Recruitment, Demographics and Group Assignment

Using our social networks, we were able to gather seven participants from the industry. Originally, we wanted to have more participants, but unfortunately, we were unable to find more people who were willing to participate. Four of the participants were in the control group and the other three were in the experimental group. Initially, the participants were randomly assigned to a group using a script[37] but since some of the participants showed up very late in the recruitment process, they were manually assigned to a group in an effort to balance the number of participants in each group. The study was performed during the months of May and June, in 2023.

4.7 Data Analysis

As previously mentioned, we conducted a study with seven participants. Three were in the experimental group, while the other four were in the control group. Data was gathered from both groups and compared. The result of this comparison is presented in this section. Mann-Whitney U tests[38] were used to analyze the differences between both groups. Although we did not expect these test results to be ideal given the small sample size, we found that they provided some interesting insights.

4.7.1 Anonymizing Data

To be able to publish the data that was collected, we started by anonymizing the data. We replaced any information that could be directly tied to the participants (such as names and email addresses) with numerical IDs. This was done using a script[39] to assign IDs to the participant and another one[40] was used to convert the data into an anonymized state.

[36] Likert scale, https://en.wikipedia.org/wiki/Likert_scale

[37] generate_groups.py, https://github.com/matosjoaops/dockerlive-study-data/blob/main/scripts/generate_groups.py

[38] Mann-Whitney U test, https://en.wikipedia.org/wiki/Mann-Whitney_U_test

[39] assign_ids.py, https://github.com/matosjoaops/dockerlive-study-data/blob/main/scripts/assign_ids.py

[40] anonymize_data.py, https://github.com/matosjoaops/dockerlive-study-data/blob/main/scripts/anonymize_data.py

4.7.2 Recruitment Questionnaire

This questionnaire had the following questions:

- **Q1**: What is your current role?
- **Q2**: Approximately how many years of professional experience do you have in software development and related tasks?
- **Q3**: Approximately how many years of professional experience do you have developing Dockerfiles?
- **Q4**: Approximately how many Dockerfiles have you written from scratch into a working first version?
- **Q5**: Approximately how many Dockerfiles have you edited?
- **Q6**: Have you ever used a Visual Studio Code extension to edit Dockerfiles?

The purpose of this questionnaire was to measure the amount of experience the participants had with Docker. The first question simply aims to determine a participant's role to see if their functions are related to Dockerfile development. With the second and third questions, we aim to measure the experience the participants have in the field, and, more specifically, with Dockerfiles. The fourth and fifth questions aim to, more accurately, evaluate each participant's experience by first ascertaining how many Dockerfiles they wrote (does not count files that the person only edited) and then, how many files the person edited (but did not create). The purpose of the last question was to figure out if any of the participants had ever used Dockerlive before, as this could have an impact on their performance.

Looking at the answers, for **Q1** (shown in Figure 4.3), we can see that both groups had people whose roles were "Operations" and "Developer". If we look at roles that were only present in one of the groups, we see that the control group had two architects, while the experimental group had a person with the role "Head of Innovation". It is difficult to draw conclusions from these roles, given that some of them may have overlapping functions despite having different names, and job titles are usually not comparable between companies. Overall, we do not believe these differences had a big impact on our results.

For **Q2**, when looking at the answers (shown in Figure 4.4) that were given, we see that both groups had participants with a medium amount of experience (between 6 and 15 years of experience). However, we also see that the control group had two participants with at most 5 years of experience, while the experimental group had a participant with more than 20 years of experience. This suggests that the experimental group's participants had more experience, which could have affected their performance in the tasks.

Moving on to **Q3** (answers shown in Figure 4.5), we see that the experimental group is unbalanced. Two participants have 2 or fewer years of experience with Dockerfiles, while the other one has 5 or more. In the control group, we see two participants with 3 years of experience, one

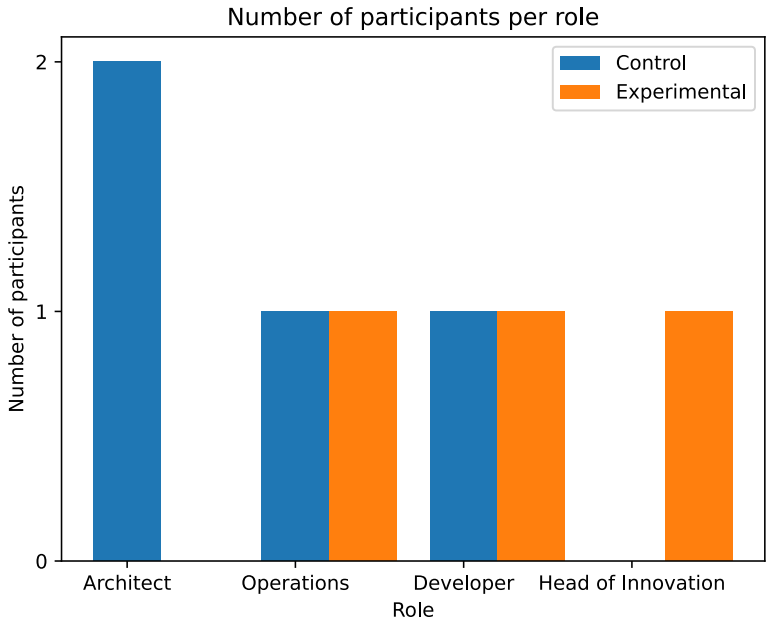

Figure 4.3: Distribution of participants' roles across groups

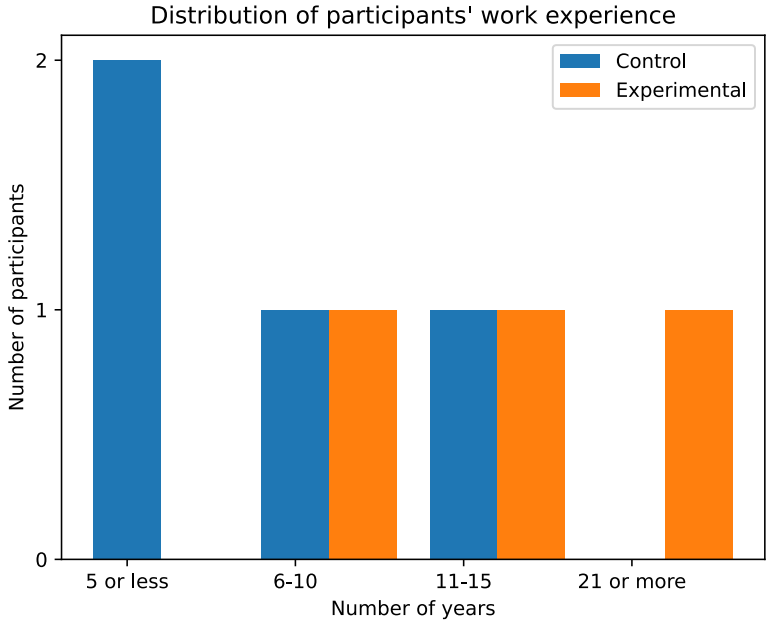

Figure 4.4: Distribution of years of experience across groups

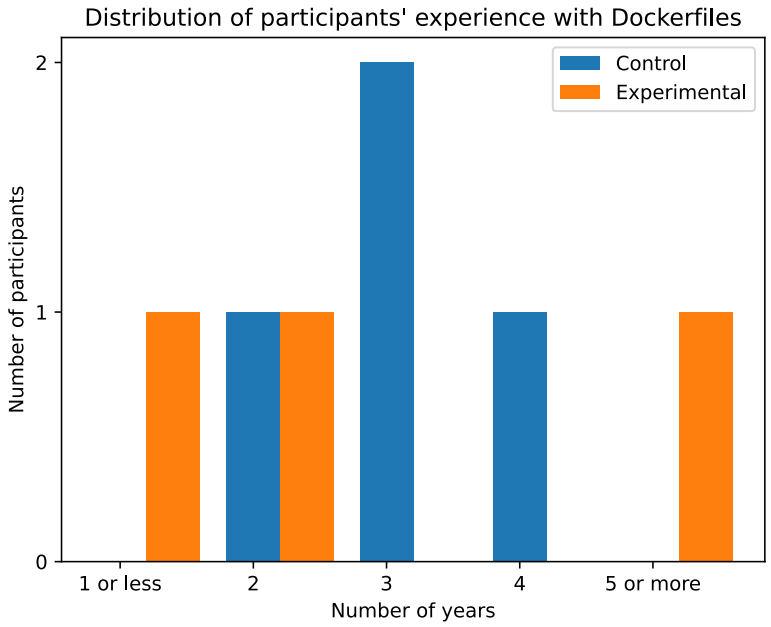

Figure 4.5: Distribution of years of experience with Dockerfiles across groups

with 2 years of experience, and the last one has 4 years of experience. Overall, we can not say that one group is clearly more experienced than the other, given the lack of balance seen in the experimental group.

When looking at **Q4** (whose answers can be seen in Figure 4.6), we see that the experimental group is less experienced than the control group, with two of the participants from the experimental group having only created up to 3 Dockerfiles. On the other hand, every participant in the control group has created at least 4 Dockerfiles.

With **Q5** (whose answers can be seen in Figure 4.7), the situation is similar to the one we saw in the previous question. Overall, the experimental group seems to be less experienced with Docker than the control group.

Finally, with **Q6**, we determined that most participants had used a Visual Studio Code extension to edit Dockerfiles in the past. However, none of them had used Dockerlive (although one of them had heard of it previously). This means their performances could not have been influenced by previous experiences with Dockerlive.

4.7.3 Task Data

Starting with the Mann-Whitney U tests, the results do not allow us to make many claims with high levels of confidence.

Due to the small number of participants, our ρ values are quite high, especially when looking at the total duration of the tasks (shown by Table 4.1). When looking at the duration of Task 3, the mean and the standard deviation are very similar, which is reflected in the ρ value of 0.86.

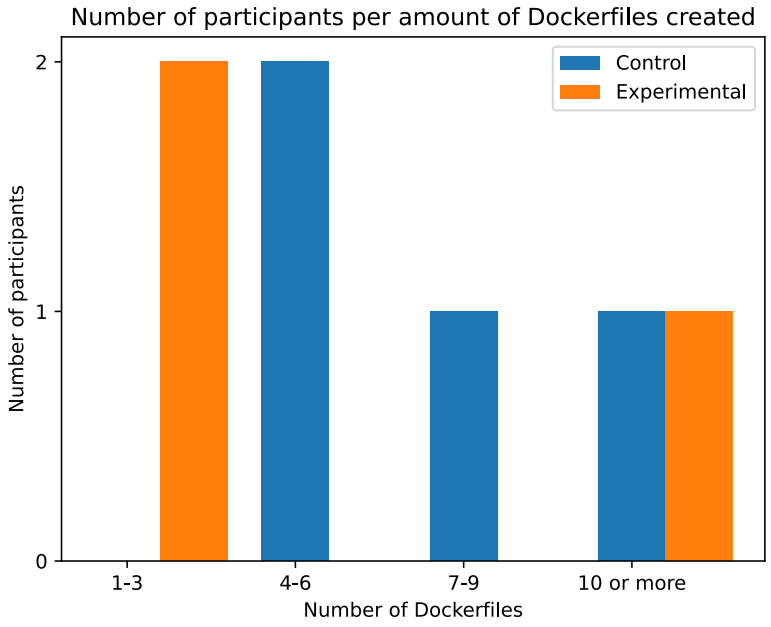

Figure 4.6: Distribution of Dockerfiles written by participants across groups

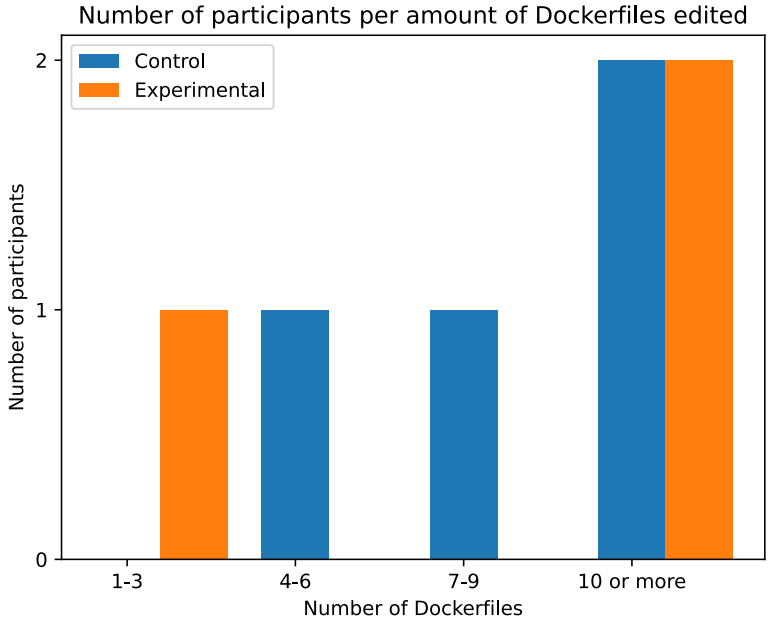

Figure 4.7: Distribution of Dockerfiles edited by participants across groups

We believe the cause may be the task's low complexity, which made it easy to complete for both groups, irrespectively. Tasks 1 and 2 show much bigger differences, despite still having relatively high ρ values.

Task	$\bar{x}$ **(Control)**	σ **(Control)**	$\bar{x}$ **(Experimental)**	σ **(Experimental)**	**U**	ρ
1	16:02	07:46	10:53	03:52	9.0	0.4
2	11:03	06:03	06:13	01:51	11.0	0.11
3	05:47	03:06	05:29	02:56	7.0	0.86

Table 4.1: Mean, Standard Deviation and two-sided Mann-Whitney U test results for total task duration (time data is presented in minutes and seconds rounded to the nearest integer) in both groups

When it comes to context switching, the test results can be seen in Table 4.2. There are significant differences between both groups across the three tasks. For Task 2, we can even say, with a confidence level of 90%, that the numbers we obtained belong to different distributions. Even for Task 3, where the task duration was similar in both groups, the number of context switches in the experimental group was much lower. This suggests the tool we created reduces the amount of context switching that is performed.

Task	$\bar{x}$ **(Control)**	σ **(Control)**	$\bar{x}$ **(Experimental)**	σ **(Experimental)**	**U**	ρ
1	44.0	25.1	9.0	6.0	11.0	0.11
2	26.0	10.98	7.33	3.21	12.0	0.06
3	18.5	10.25	4.0	2.0	11.0	0.11

Table 4.2: Mean, Standard Deviation and two-sided Mann-Whitney U test results for context switching (number of context switches) in both groups

It should be noted that the numbers presented in Tables 4.1 and 4.2 were rounded to the second decimal place (excluding time-related data).

Despite the underwhelming results of the Mann-Whitney U tests, if we look at the distribution of the total time spent on each task (shown in Figure 4.8), we see that for Tasks 1 and 2, the control group tended to need more time to carry out the tasks than the experimental group (there was a participant in the control group that finished very quickly but it should be noted that the resulting Dockerfile did not reduce the image size). The total time spent on Task 3 was similar in both groups. As a side note, the box plot shows someone going over the time limit on Task 1 because one of the participants in the control group was accidentally given extra time.

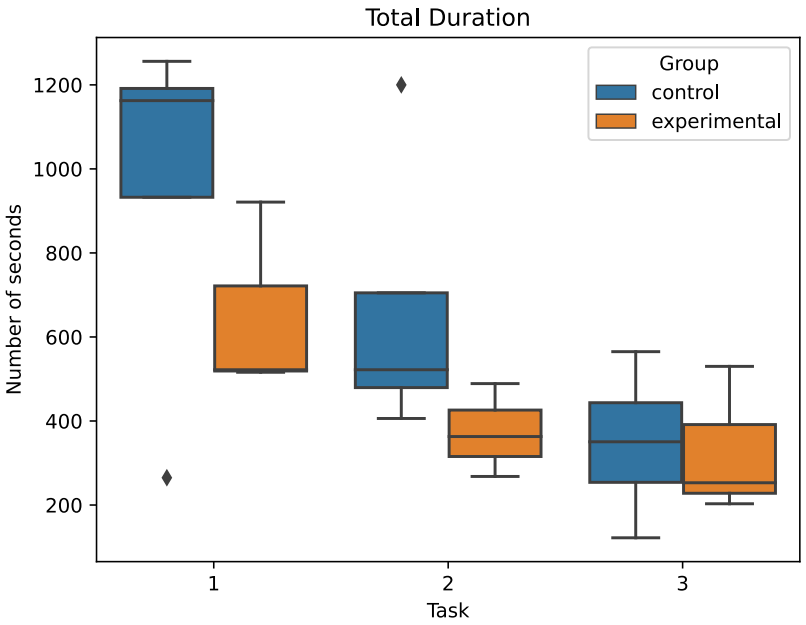

Figure 4.8: Time spent per task

When looking at the distribution of the number of context switches performed per task (shown in Figure 4.9) in both groups, we also see that the experimental group performed less context switching than the control group.

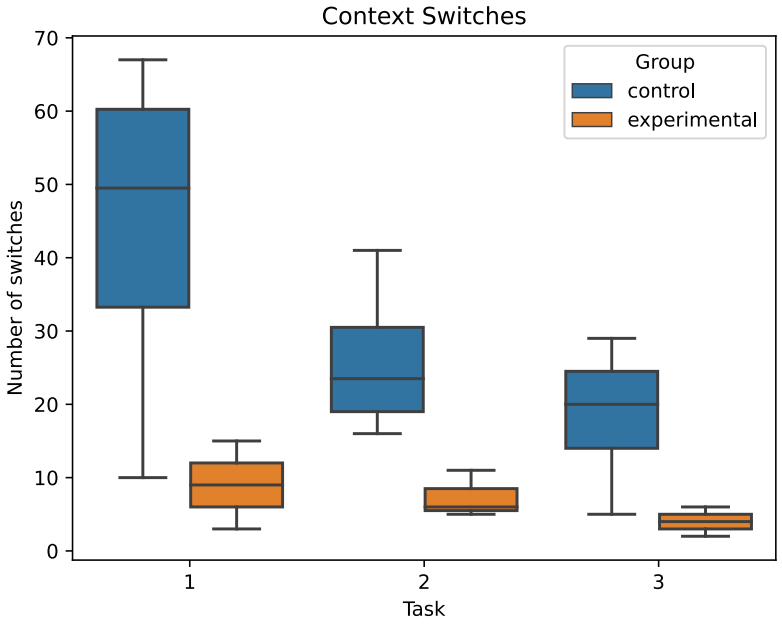

Figure 4.9: Number of context switches per task

Another data point we analyzed was the amount of time spent in each context per task. This

analysis was performed for the most relevant contexts ("VSCode", "Terminal", "Firefox", and "Reading Instructions", other contexts were not analyzed due to having very low values). Figure 4.10 shows the box plots corresponding to the amount of time spent in the contexts "VSCode", "Terminal", "Firefox" and "Reading Instructions".

Starting with Visual Studio Code, we can see that, in general, the experimental group spent more time in the editor than the control group. If we take into account the lower number of context switches performed by the control group, these two data points suggest the additional information Dockerlive provided to the experimental group reduced their need to use other apps while carrying out the tasks.

Looking at the data from the "Terminal" context, we see the inverse situation. The control group spent more time in this context than the experimental group. Again, this can be explained by the additional information provided by Dockerlive. This information made it easier for the experimental group to obtain feedback about the container being developed without having to manually build an image and execute it.

Moving on to "Firefox", the differences are even more extreme. The experimental group did not spend a single second using Firefox. This suggests the new repair features helped the participants from the experimental group carry out the tasks. On the other hand, the control group had to use Firefox to figure out ways to change the file that would allow them to accomplish the tasks' objectives.

Regarding the time spent reading instructions, it seems that both groups took a similar amount of time. Having access to Dockerlive should not make the instructions easier or harder to understand and the data seems to support that.

4.7.4 Dockerfile Data

In addition to the data gathered from the participants' performance itself, we also gathered data from the Dockerfiles they produced. As a reminder, we verified if the Dockerfiles could be used to build images and if those images could be executed "successfully". The meaning of "successful" depends on the task (this is explained in Section 4.5.2).

For Task 1, all the participants from both groups were able to produce a Dockerfile that could be used to successfully build an image that could then be executed to obtain the expected output. For this task, we also measured the size of the image (as the image size was the focus of Task 1). The distribution of the image sizes can be seen in Figure 4.11. As the image shows, the control group's amplitude is quite high, with one of the images being as small as 56 MB, while others were as large as 962 MB (the size of the image that is used in the Dockerfile's original state). For the experimental group, two of the images had a size of 178 MB, while the other one had a size of 177 MB (there was a small optimization that was performed here, making this image slightly smaller). The image recommended by Dockerlive for this task's project has a size of 178 MB, meaning the participants from the experimental group just used Dockerlive's recommendation. Overall, it is difficult to say how useful Dockerlive was at reducing the size of the image, given that some of the participants in the control group had smaller images than what Dockerlive was capable of

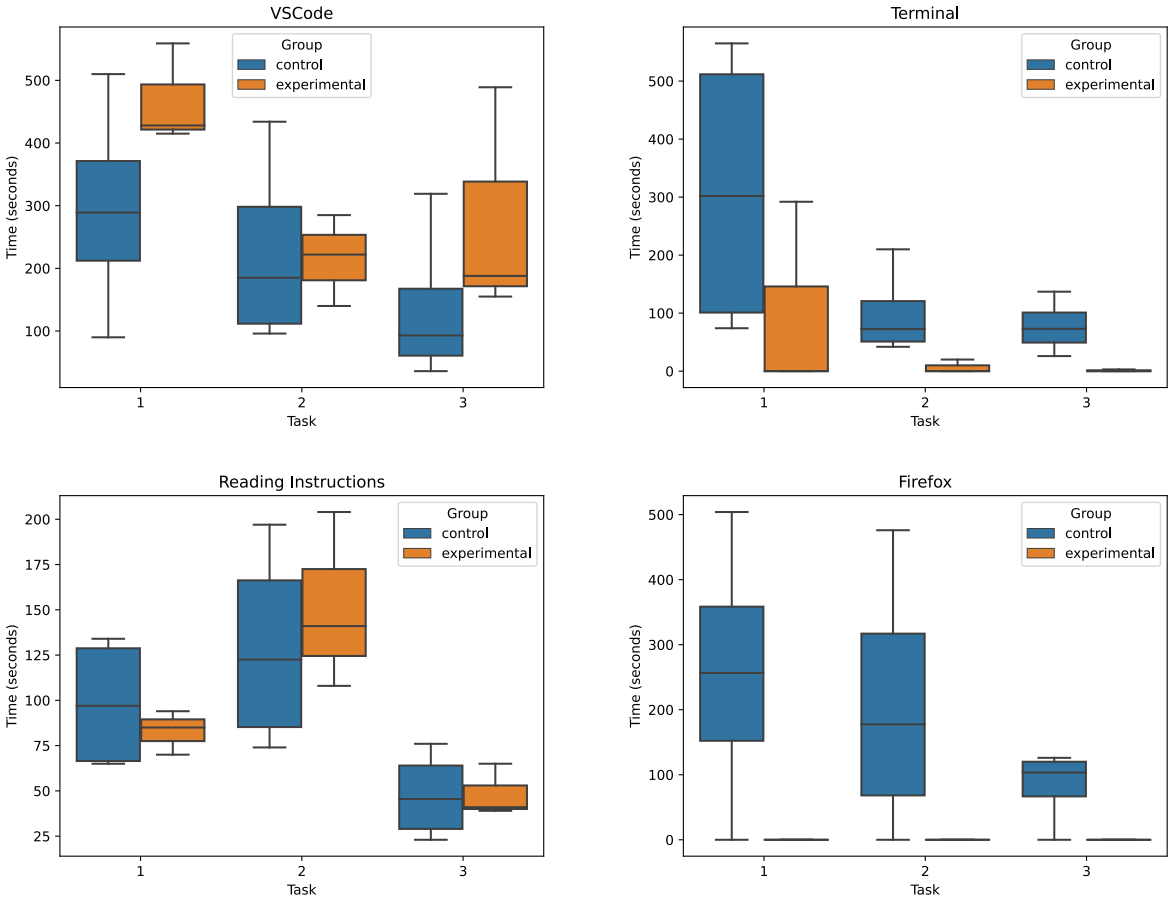

Figure 4.10: Time (in seconds) spent in each context for both groups

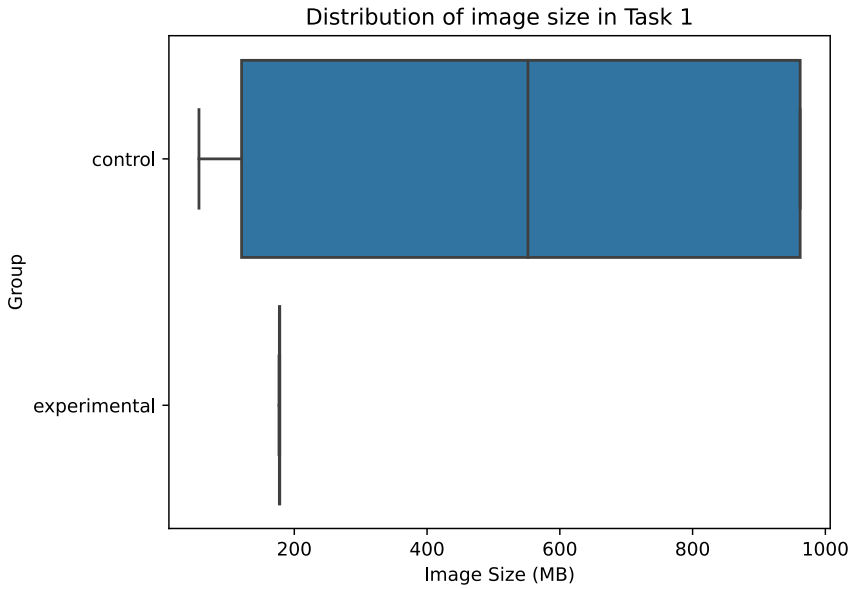

Figure 4.11: Distribution of image size for Task 1 in both groups

recommending. It is possible that the experimental group's images would have been smaller if Dockerlive's recommendations were different (for example, if Dockerlive recommended images based on Alpine Linux[41]).

For Task 2, all the participants produced a Dockerfile that could be used to build an image. However, one of the Dockerfiles from the control group could not be executed "successfully" since one of the files could still be accessed by the script. This was due to this Dockerfile not changing users. Every other Dockerfile could be executed successfully. The participants from the experimental group just followed Dockerlive's suggestions (changing users and changing the working directory). The participants from the control group employed similar techniques but one of them moved the files without changing the COPY instructions, which despite being unintended, does not violate the restrictions that were provided. With this in mind, it seems that Dockerlive was not as helpful as we would have liked in this task.

For Task 3, all the participants produced Dockerfiles that could be built and executed "successfully". This means everyone added the *npm install* command. Additionally, all the participants from the experimental group also removed the redundant Node.js installation. In the control group, half of the participants removed this installation, while the other half did not. One of the participants that removed it also used a multi-stage build. One of the participants that did not remove the installation was aware that it was redundant. This could mean the instructions for the task and the feedback provided by the observer should have been clearer. Overall, it seems this task was too simple for Dockerlive's suggestions to have a substantial impact on the quality of the Dockerfiles.

[41] Alpine Linux, https://www.alpinelinux.org/

4.7.5 Task Questionnaire

This questionnaire had two versions: one for the control group (which did not have access to Dockerlive) and another for the experimental group (which had access to Dockerlive). The differences between the two versions of the questionnaire will be pointed out throughout this portion of the document. Most of the questions used a Likert scale. The questions that did *not* use a Likert scale will be listed in the part of the document that precedes their analysis.

The first section of the questionnaire (Section **A**) had the following questions (this section was identical for both groups):

- **A1**: I am confident in my English skills.
- **A2**: I feel comfortable programming in Python.
- **A3**: I feel comfortable programming in JavaScript and Node.js.
- **A4**: I feel comfortable working with Firefox.
- **A5**: I feel comfortable working with Visual Studio Code.
- **A6**: I feel comfortable working with Linux-based desktop-focused operating systems.
- **A7**: I feel comfortable using Visual Studio Code to edit Dockerfiles.
- **A8**: I feel comfortable interacting with Docker through a terminal.

Starting with the data from Section A (Figure 4.12 shows the results for the control and experimental groups), we see that both groups are comfortable with their English. However, some participants were not comfortable with Javascript and Python (in either group). Regarding **A4** and **A5**, we see that the control group is more comfortable with Firefox and Visual Studio Code than the experimental group. For the last three questions of this section, we again see the control group showing higher levels of comfort with Linux operating systems and Docker.

The task-related sections (**B**, **C** and **D**) had the following questions (questions **B3**, **C3** and **D3** were only present on the experimental version of the questionnaire):

- **B1**: I was able to understand what to do in Task 1.
- **B2**: I was able to finish Task 1 without any problems.
- **B3**: Dockerlive helped me do Task 1.
- **C1**: I was able to understand what to do in Task 2.
- **C2**: I was able to finish Task 2 without any problems.
- **C3**: Dockerlive helped me do Task 2.
- **D1**: I was able to understand what to do in Task 3.
- **D2**: I was able to finish Task 3 without any problems.

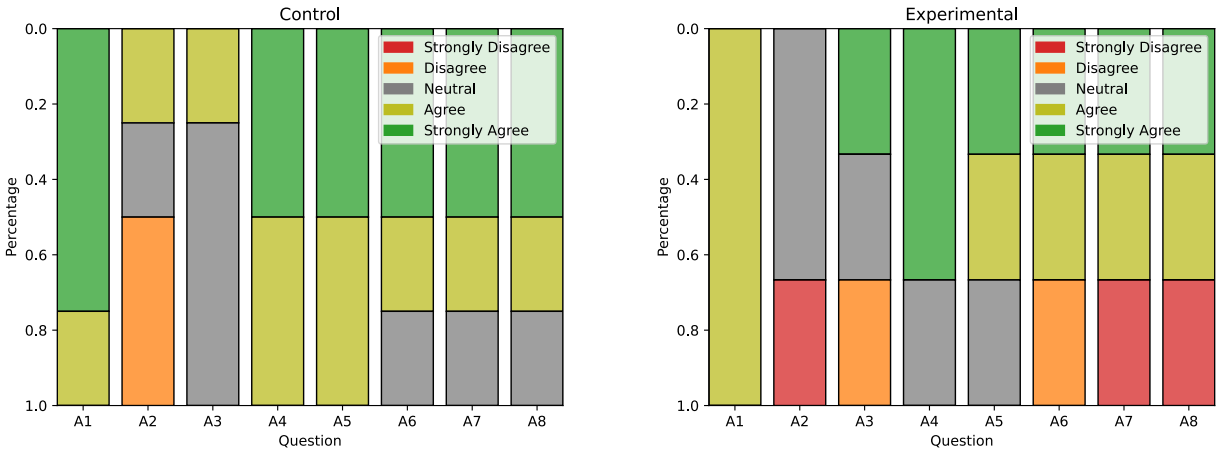

Figure 4.12: Distribution of the answers to section A in both groups

- **D3**: Dockerlive helped me do Task 3.

For Task 1, both groups understood what they needed to do. However, the control group struggled to finish the task a bit more than the experimental group did (as the answers to **B2** show). Regarding Task 2, we see that it was more difficult for the control group to understand and finish this task than it was for the experimental group. This suggests that Dockerlive helped the experimental group determine what they needed to do. In Task 3, we see that the experimental group actually struggled more than the control group. A possible explanation would be that the task was so simple that Dockerlive's numerous suggestions confused the participants from the experimental group. Finally, when looking at questions **B3**, **C3**, and **D3**, we can say that, overall, the experimental group found Dockerlive to be helpful when carrying out these tasks. Figure 4.13 shows the results for the control and experimental groups.

The final sections (**E**, **F** and **G**) had the following questions (section **G** was only present on the experimental version of the questionnaire; questions **F3**, **G5**, **G6** and **G7** did not use a Likert scale):

- **E1**: I felt comfortable with the remote environment.
- **F1**: I felt like the tasks took a long time to carry out.
- **F2**: I felt like the tasks would have been easier to carry out if I had access to other tools.
- **F3**: Please write the name of any tool(s) that you believe would have helped you carry out the tasks.
- **G1**: Dockerlive made the tasks easy to carry out.
- **G2**: Dockerlive was easy to use.
- **G3**: Dockerlive's suggestions were useful in carrying out the tasks.
- **G4**: I would like to use Dockerlive in the future.

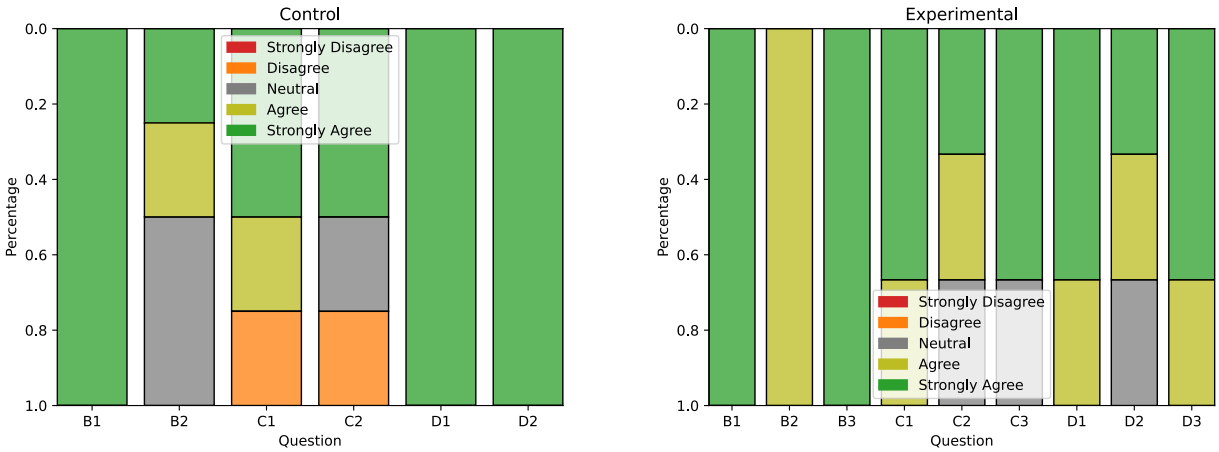

Figure 4.13: Distribution of the answers to sections B, C and D in both groups

- **G5**: Which of Dockerlive's features helped you the most? And why?
- **G6**: What problems did you run into while using Dockerlive to do the tasks?
- **G7**: If you could, how would you improve Dockerlive?

When looking at section **E**, we see that neither group felt uncomfortable with the remote environment. When looking at section **F**, the experimental group did not feel that tasks took a long time to complete, and they also did not feel they needed other tools to help them out. In the control group, while the overall sentiment is still of disagreement some participants felt more inclined to agree with this section's statements. This suggests that Dockerlive was helpful to the experimental group. This viewpoint is reinforced by the experimental group's answers to section **G**, with everyone agreeing that Dockerlive was helpful, easy to use and that they would like to use it in the future. Figure 4.14 shows the results for the control and experimental groups.

Finally, we are looking at the answers to the questions that did not use a Likert scale. Regarding **F3**, the control group said they would have liked to use the official Docker extension[42] for Visual Studio Code or IntelliJ IDEA[43] (which, apparently has some features that help developers edit Dockerfiles). The experimental group did not list any tools but one participant said that it felt like Dockerlive was only used with "happy paths"[44] while carrying out the tasks. Moving on to **G5**, users from the experimental group considered the detection of dependencies and the "user related operations" to be useful. In **G6**, users reported that the user experience related to hovering over warnings could be improved and that the problems tab in the editor's bottom panel should be highlighted. Lastly, in **G7**, users suggested making it clearer that a file's processing takes place right after the file is opened. Furthermore, a user proposed adding more context information for each suggestion, relying less on quick fixes, and providing templates with scaffolding options.

[42]Docker, https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker

[43]IntelliJ IDEA, https://www.jetbrains.com/idea/

[44]Happy path, https://en.wikipedia.org/wiki/Happy_path

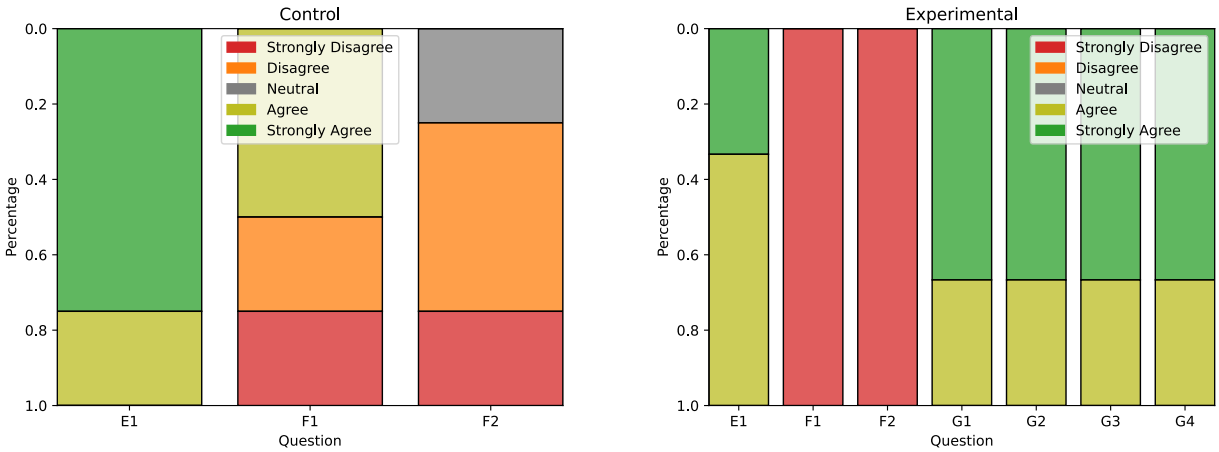

Figure 4.14: Distribution of the answers to sections E, F and G in both groups

4.8 Threats to Validity

Now that we have presented all the data we have, we will now analyze elements that can threaten the validity of this study. We have identified the following problems.

4.8.1 Internal Validity

This section addresses threats to internal validity.

Usage of one proxy metric to represent two dimensions. To determine if our tool was capable of improving the developer experience, we tried to measure the dimensions listed by Noda et al. [40] using proxy metrics. However, we used one proxy metric to represent two of the three dimensions. A possible solution would have been to use NASA's task load index[45] to measure the cognitive load. Other solutions could require a collaboration with researchers from the medical field to analyze data points like a person's heartbeat to be able to more accurately measure the cognitive load. Regardless, we still believe context switching is an appropriate metric to address the flow state dimension.

Unbalanced groups. When looking at the answers to the recruitment questionnaire (presented in Section 4.7.2), we saw that, overall, the experimental group had less experience with Docker than the control group. On the other hand, the experimental group had more general experience in the field of software development. Given the small sample size, we should have manually assigned the participants to the groups, instead of using random assignment. Having more participants could also lead to more balanced groups when using random assignment. These two elements should be taken into account if similar studies are conducted in the future.

Poor task instructions. Some of the data presented in Section 4.7.5 could suggest that the tasks' instructions were not as clear as they could have been. Perhaps asking an impartial third party to read the instructions beforehand could have helped mitigate this. However, since both

[45] Task Load Index, https://humansystems.arc.nasa.gov/groups/TLX/

groups were given the same instructions, we do not believe this had a major impact on the results that have been presented.

Task complexity. One could argue the tasks the participants had to perform were very simple. This was done on purpose, given that more complex tasks would have required us to give the participants more time, which, in turn, would have made the study harder to schedule and would have reduced the number of participants even further. Additionally, if the tasks were more complex, it would have been harder to measure differences between groups. For example, if the participants were given a task where Dockerlive could only help the participants accomplish 5% of the objectives, the differences between the groups would have been negligible regardless of Dockerlive's helpfulness. As future work, we believe that a study with more complex tasks could provide a better representation of real-world scenarios (this is discussed in more detail in Section 4.11).

4.8.2 External Validity

This section addresses threats to external validity.

Small amount of data. It is undeniable that the number of participants is very small and that we can not make any claims with high levels of confidence. However, even with such a limited amount of data, we can already identify some data points that suggest Dockerlive has a positive impact on the Dockerlive development experience. We believe these data points would still be present if the study had been conducted with a larger group of participants, but we would need to carry out empirical studies with more participants before we can confidently assert the generalizability of our results.

Technical problems. Some technical problems occurred while performing the study with some of the participants, which could have affected the results that we presented. However, because the data was extracted from recordings we were able to exclude the time spent dealing with the technical problems (the scripts we used even take this into account, as mentioned in Section 4.5.1). Therefore, we do not believe the problems had a big impact on the results.

4.9 Findings

With all the data presented and any possible concerns addressed, we can now answer the questions listed in Section 4.2.

Starting with **RQ1**, we can say that Dockerlive can decrease the cognitive load associated with Dockerfile development tasks as the results from Section 4.7.3 show the participants from the experimental group performed less context switching than those in the control group. This is also reinforced by the shorter periods of time that the experimental group spent in contexts that were not Visual Studio Code. Looking at the results of the Mann-Whitney U tests, we saw lower ρ values when comparing context switches between both groups, although they are still a bit higher than we would have liked. Furthermore, the answers to the questionnaire filled out while the tasks were carried out (presented in Section 4.7.5) show the participants from the experimental group struggled less with the tasks than those in the control group.

Regarding **RQ2**, we can also say that Dockerlive promotes a flow state. As our answer to **RQ1** pointed out, Dockerlive reduces the amount of context switching a user needs to perform.

RQ3 can also be answered positively. The data presented in Section 4.7.3 shows the participants from the experimental group took less time to complete Tasks 1 and 2, although we can not make any claims with high levels of confidence given the results of the Mann-Whitney U tests.

However, with **RQ4**, we can not say that Dockerlive improved the quality of the Dockerfiles. When looking at the results presented in Section 4.7.4, we see that, in Task 1, some participants from the control group were able to create Dockerfiles with smaller image sizes than those seen in the experimental group. In Tasks 2 and 3, we can not say that the experimental group's Dockerfiles were clearly better than the control group's. Actually, in some cases, the reverse was true, with a participant from the control group implementing a multi-stage build in Task 3.

To conclude, we can answer **RQ1**, **RQ2** and **RQ3** positively. The data shows that Dockerlive can reduce, not only, the amount of time spent, but also, the number of context switches performed while developing Dockerfiles. Since we used these metrics as proxies to measure the developer experience (as mentioned in Section 4.3.1), we can say that we have achieved our primary goal of improving the developer experience of Dockerfile developers. However, we can not say the same thing for our secondary goal of improving the quality of Dockerfiles, as the data did not show higher-quality Dockerfiles being produced by the participants who had access to Dockerlive.

4.10 General Discussion

The results we presented show that the features we chose to implement as a way of achieving a higher level of liveness are useful and can improve the developer experience of developers working with Dockerfiles. This corroborates Reis' findings. They were also able to improve the developer experience through the use of liveness, although the version of Dockerlive that was used at that time offered a lower level of liveness than the one we used [50].

As we have been mentioning throughout this chapter, all the resources used to perform the study and analyze the data that was gathered can be found, respectively, in the *dockerlive-study*[46] and *dockerlive-study-data*[47] repositories. This combination of repositories can be considered to be this study's replication package.

4.11 Future Work

Now that the study has been presented, we will list some elements that should be taken into consideration if someone wishes to conduct a similar study in the future.

- **Larger scale** - As previously mentioned, this study did not have many participants. A future version of this study should be conducted with more participants to be able to obtain data

[46] dockerlive-study, https://github.com/matosjoaops/dockerlive-study
[47] dockerlive-study-data, https://github.com/matosjoaops/dockerlive-study-data

that is a better representation of reality. Furthermore, we believe the tasks' complexity could also be increased to more closely resemble the kind of tasks that Dockerfile developers deal with in the real world. Finally, with more participants, it should also be possible to randomly assign them to groups, without making the groups unbalanced. Overall, we believe that increasing the scale of the study could provide a better representation of reality, along with more balanced groups. Although, this would require a substantial time investment.

- **Case study** - A different approach would be to conduct a case study[48] instead of an experiment. The participants would be given access to the tool and they would use it while performing real-world tasks in an uncontrolled environment for an extended period of time. Because the tool is being used in the real world instead of a controlled environment, the data that would result from such a study would provide a very accurate representation of the tool's performance. However, this would also require a substantial time investment and the tool would also have to be very robust.

[48] Case Study, https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=CaseStudy

Chapter 5

Conclusion

Now that everything has been presented, this chapter summarizes the contents of this dissertation and concludes it.

5.1 Overview

Over the past few years, we have seen developers build highly complex systems. As the complexity of those systems has increased, it has become more difficult to maintain them. In response to this, developers have started to adopt the microservices architecture, which allows them to build larger systems using small components. One of the technologies that enable this architecture is Docker [23].

When using Docker, developers create a Dockerfile, a configuration file that is used to create a container. Creating a Dockerfile can be a difficult task, and the developers creating them face a few challenges (as described by Chapter 2). Overall, the Dockerfile development experience can be improved.

This brings us to our hypothesis (mentioned in Section 1.2). We believe that a development environment with a high level of liveness can address the problems that developers face when writing Dockerfiles. We also believe that the level of liveness can be increased by offering features like automatic modification and generation of Dockerfiles.

With this in mind, we created a tool that offers these features. The automatic generation feature allows a user to generate a Dockerfile using information gathered from the project the user is working on. The automatic modification feature scans the file a user is currently editing and, if problems are found, the tool suggests modifications to the user. The tool is called Dockerlive V2, and its design has been described in Chapter 3.

To verify Dockerlive V2's helpfulness and ability to improve the development experience, we conducted an experiment with two groups of professionals (a control group and an experimental group). These groups were asked to complete a set of tasks where they had to edit Dockerfiles. One of the groups had access to Dockerlive V2, while the other did not. Data was gathered from both

groups and then compared. The results were analyzed in Chapter 4 and suggest that Dockerlive V2 may be capable of improving the Dockerfile development experience.

This shows that the higher level of liveness that was achieved with the implementation of the automatic generation and modification features can have a positive impact on the Dockerfile developer experience, validating our hypothesis.

In the future, we believe Dockerlive could be improved by making the new features more intuitive and using a different method to connect with Hermit. Furthermore, we believe a study with more participants and more complex tasks would provide a better representation of the real world. Sections 3.7 and 4.11 describe future work in more detail.

5.2 Contributions

As part of the work required to validate our hypothesis, we made a few contributions that we believe can be useful to the scientific community. These contributions are described in this section.

- **Literature review of works covering Dockerfile development challenges** - We performed a literature review involving several works to learn more about the challenges affecting Dockerfile developers. These works were split into different categories depending on the challenge they covered and compared with others from the same category. Many works also contained approaches designed to deal with the challenge they covered. These approaches were also compared. Overall, we gathered a substantial amount of data regarding the issues surrounding Dockerfile development. This analysis is covered in more detail in Chapter 2.
- **Implementation of an approach that improves the Dockerfile developer experience** - We designed an approach that would provide additional features to developers working with Dockerfiles. One of the features would require a tool to continuously scan a file for problems that could be automatically repaired and suggest the corresponding modifications to a user. The other feature would give users the option to generate a Dockerfile by analyzing the project the user is currently working on. We have created an implementation of this approach and called it Dockerlive V2. This is covered in more detail in Chapter 3.
- **Empirical study with industry participants** - To evaluate the impact of the created tool on the Dockerfile development experience, we conducted a study where we analyzed the performance of users (recruited from the industry) who were asked to perform three tasks that required them to edit Dockerfiles. The study had two groups: an experimental group and a control group. Data was gathered from both groups and then compared. The results have shown that the created tool can improve the developer experience of developers working with Dockerfiles by reducing the amount of time required to complete tasks, as well as the amount of context switching that users need to perform. Although the study did not have many participants, we believe that all the information we provided (the methodology we followed along with a replication package) allows someone to easily replicate the study with more participants. The study is covered in more detail in Chapter 4.

References

[1] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Large-scale Debloating of Binary Shared Libraries. *Digital Threats: Research and Practice*, 1(4):19:1–19:28, 2020.

[2] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, Programming '19, pages 1–6. Association for Computing Machinery, 2019.

[3] Fahmi Abdulqadir Ahmed and Dyako Fatih. Security Analysis of Code Bloat in Machine Learning Systems. Master's thesis, University of Gothenburg, 2022.

[4] Diogo Amaral, Gil Domingues, João Pedro Dias, Hugo Sereno Ferreira, Ademar Aguiar, Rui Nóbrega, and Filipe Figueiredo Correia. Live software development environment using virtual reality: A prototype and experiment. In *Evaluation of Novel Approaches to Software Engineering: 14th International Conference, ENASE 2019, Heraklion, Crete, Greece, May 4–5, 2019, Revised Selected Papers 14*, pages 83–107. Springer, 2020.

[5] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering*, 27(2):49, 2022.

[6] Benjamin Benni, Sébastien Mosser, Philippe Collet, and Michel Riveill. Supporting microservices deployment in a safer way: A static analysis and automated rewriting approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1706–1715. Association for Computing Machinery, 2018.

[7] Diogo Campos. Tests as specifications towards better code completion. Master's thesis, Universidade do Porto (Portugal), 2019.

[8] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, 2009.

[9] Francesco Caturano, Nicola d' Ambrosio, Gaetano Perrone, Luigi Previdente, and Simon Pietro Romano. ExploitWP2Docker: A Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges. In *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–7, 2022.

[10] Admin's Choice. Containers Vs VMs : Top 5 Differences you must know. https://www.adminschoice.com/containers-vs-vms-top-5-differences-you-must-know, 2021. Accessed: 2023-02-02.

[11] Thien-Phuc Doan and Souhwan Jung. DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning. *Computers, Materials & Continua*, 72(1):1699–1711, 2022.

[12] José Pedro da Silva e Sousa et al. Live acceptance testing using behavior driven development. Master's thesis, Universidade do Porto (Portugal), 2020.

[13] Sara Fernandes. A live environment for inspection and refactoring of software systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1655–1659, 2021.

[14] Martin Fowler. CodeSmell. https://martinfowler.com/bliki/CodeSmell.html. Accessed: 2023-01-28.

[15] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, 2016.

[16] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. RUDSEA: Recommending updates of Dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 796–801. Association for Computing Machinery, 2018.

[17] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 38–49, 2020.

[18] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d' Amorim, and Thomas Reps. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1148–1160, 2021.

[19] Eric Horton and Chris Parnin. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 328–338, 2019.

[20] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 28–37, 2019.

[21] IBM. Infrastructure as Code | IBM. https://www.ibm.com/topics/infrastructure-as-code. Accessed: 2023-02-02.

[22] IBM. What are containers? https://www.ibm.com/topics/containers. Accessed: 2023-01-24.

[23] IBM. What are Microservices? | IBM. https://www.ibm.com/topics/microservices. Accessed: 2023-01-30.

[24] IBM. What is DevOps? | IBM. https://www.ibm.com/topics/devops. Accessed: 2023-02-02.

[25] Susan Jamieson. Likert scales: How to (ab)use them. *Medical Education*, 38(12):1217–1218, 2004.

[26] Stefan Kehrer, Florian Riebandt, and Wolfgang Blochinger. Container-Based Module Isolation for Cloud Services. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 177–17709, 2019.

[27] Shinya Kitajima and Atsuji Sekiguchi. Latest Image Recommendation Method for Automatic Base Image Update in Dockerfile. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari, editors, *Service-Oriented Computing*, Lecture Notes in Computer Science, pages 547–562. Springer International Publishing, 2020.

[28] Emna Ksontini and Marouane Kessentini. Refactorings and Technical Debt for Docker Projects. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[29] Mingjie Li, Xiaoying Bai, Minghua Ma, and Dan Pei. DockerMock: Pre-Build Detection of Dockerfile Faults through Mocking Instruction Execution, 2021.

[30] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55–55, 1932.

[31] Markus Linnalampi. Outdated software in container images. Master's thesis, Aalto University, 2021.

[32] Pedro Lourenço, João Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Cloudcity: A live environment for the management of cloud infrastructures. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2019.

[33] Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. An Empirical Case Study on the Temporary File Smell in Dockerfiles. *IEEE Access*, 7:63650–63659, 2019.

[34] João Carlos Cardoso Maduro. Automatic Service Containerization with Docker. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2021.

[35] John H. Maloney and Randall B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology - UIST '95*, pages 21–28. ACM Press, 1995.

[36] Gloria Mark, Daniela Gudith, and Ulrich Klocke. The cost of interrupted work: More speed and stress. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems*, pages 107–110, 2008.

[37] Scott McMillan. MAKING CONTAINERS EASIER WITH HPC CONTAINER MAKER.

[38] Jenny Morales, Cristian Rusu, Federico Botella, and Daniela Quiñones. Programmer eXperience: A Systematic Literature Review. *IEEE Access*, 7:71079–71094, 2019.

[39] Emanuel Moreira, Filipe F Correia, and João Bispo. Overviewing the liveness of refactoring for energy efficiency. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 211–212, 2020.

[40] Abi Noda, Margaret-Anne Storey, Nicole Forsgren, and Michaela Greiler. DevEx: What Actually Drives Productivity: The developer-centric approach to measuring and improving productivity. *Queue*, 21(2):Pages 20:35–Pages 20:53, 2023.

[41] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J. Eglen, Tim Head, Tony Hirst, and Benjamin D. Evans. Ten simple rules for writing Dockerfiles for reproducible data science. *PLOS Computational Biology*, 16(11):e1008316, 2020.

[42] Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-driven management of docker containers. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 718–725, 2016.

[43] Bruno Piedade, João Pedro Dias, and Filipe F Correia. An empirical study on visual programming docker compose configurations. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10, 2020.

[44] Bruno Piedade, João Pedro Dias, and Filipe F Correia. Visual notations in container orchestrations: an empirical study with docker compose. *Software and Systems Modeling*, 21(5):1983–2005, 2022.

[45] Paolo Ernesto Prinetto, Dott Riccardo Bortolameotti, and Giuseppe Massaro. Security Misconfigurations Detection and Repair in Dockerfile. Master's thesis, Politecnico di Torino, 2022.

[46] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 476–486. Association for Computing Machinery, 2017.

[47] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New Directions for Container Debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '17, pages 51–56. Association for Computing Machinery, 2017.

[48] David Reis and Filipe F. Correia. Dockerlive : A live development environment for Dockerfiles. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–4. IEEE, 2022.

[49] David Reis, Bruno Piedade, Filipe F. Correia, João Pedro Dias, and Ademar Aguiar. Developing Docker and Docker-Compose Specifications: A Developers' Survey. *IEEE Access*, 10:2318–2329, 2022.

[50] David Alexandre Gomes Reis. Live Docker Containers. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2020.

[51] Giovanni Rosa, Antonio Mastropaolo, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises. Technical report, STAKE Lab - University of Molise, Pesche, Italy and Software Institute - Universit‘ a della Svizzera Italiana (USI), Switzerland, 2023.

[52] Corrado Santoro, Fabrizio Messina, Fabio D'Urso, and Federico Fausto Santoro. Wale: A Dockerfile-Based Approach to Deduplicate Shared Libraries in Docker Containers. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 785–791, 2018.

[53] Sid Palas. This is a valid Dockerfile for a NodeJS application. https://twitter.com/sidpalas/status/1634194026500096000, 2023. Accessed: 2023-06-01.

[54] Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations: The Case for Microservice Architecture. *SN Computer Science*, 2(6):459, 2021.

[55] Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Engineering software for the cloud: patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, pages 1–8, 2016.

[56] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[57] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, 2013.

[58] Yujing Wang and Qinyang Bao. A Code Injection Method for Rapid Docker Image Building, 2019.

[59] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. *IEEE Access*, 8:34127–34139, 2020.

[60] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An Empirical Study of Build Failures in the Docker Context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 76–80. ACM, 2020.

[61] Jiwei Xu, Yuewen Wu, Zhigang Lu, and Tao Wang. Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 185–190, 2019.

[62] Hongjie Ye, Jiahong Zhou, Wei Chen, Jiaxin Zhu, Guoquan Wu, and Jun Wei. DockerGen: A Knowledge Graph based Approach for Software Containerization. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 986–991, 2021.

[63] Mehdi Zarei. Investigating the inner workings of container image vulnerability scanners. Master's thesis, Oslo Metropolitan University, 2022.

[64] Shuaihao Zhong, Duoqiang Wang, Wei Li, Feng Lu, and Hai Jin. Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph. *Wireless Communications and Mobile Computing*, 2022:e4592428, 2022.

Appendix A

Questionnaires

This appendix contains the questionnaires mentioned in Sections 4.5.3 and 4.5.4. Section A.1 contains the recruitment questionnaires, Section A.2 contains the questionnaire given to the control group, and Section A.3 contains the questionnaire given to the experimental group.

A.1 Recruitment Questionnaire

Research on Dockerfile Development

We invite you to participate in our study evaluating a novel Dockerfile development environment. To qualify you should have some experience developing Dockerfiles, and participate in an online session where you may be asked to try out innovative tools for working with Dockerfiles (max. 1h duration).

The results will be used exclusively to support our research, and not for any commercial purpose. We shall publish any collected data in anonymized and aggregated forms only. By participating you will be contributing to the latest research in this topic, and will have first-hand access to the results of this study.

If you have any questions, feel free to get in touch with the researchers:
João Matos <up201703884@edu.fe.up.pt>
Filipe Correia <filipe.correia@fe.up.pt>

* Indica uma pergunta obrigatória

1. Are you willing and interested to participate in this study? *

   Marcar apenas uma oval.

   - ◯ Yes
   - ◯ No

Personal Information

2. What is your name? *

3. What is your email address? *

4. What is your current role? *

Marcar apenas uma oval.

- ◯ Developer
- ◯ Architect
- ◯ Operations
- ◯ QA
- ◯ Outra: _____

Experience

5. Approximately how many years of professional experience do you have in software development and related tasks? *

Marcar apenas uma oval.

- ◯ 5 or less
- ◯ 6-10
- ◯ 11-15
- ◯ 16-20
- ◯ 21 or more

6. Approximately how many years of professional experience do you have developing Dockerfiles? *

Marcar apenas uma oval.

- ◯ 1 or less
- ◯ 2
- ◯ 3
- ◯ 4
- ◯ 5 or more

7. Approximately how many Dockerfiles have you written from scratch into a working first version? *

This means you created the Dockerfile and worked on it until it became functional.

Marcar apenas uma oval.

- ◯ 1-3
- ◯ 4-6
- ◯ 7-9
- ◯ 10 or more

8. Approximately how many Dockerfiles have you edited? *

This means you were not responsible for the creation of the Dockerfile but you made some modifications to it.

Marcar apenas uma oval.

- ◯ 1-3
- ◯ 4-6
- ◯ 7-9
- ◯ 10 or more

9. Have you ever used a Visual Studio Code extension to edit Dockerfiles? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

10. If your answer to the previous question was "Yes", please write the name of the extension(s) that you used.

_____

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

A.2 Control Questionnaire

Improving the Developer Experience of Dockerfiles

This questionnaire aims to evaluate your experience regarding aspects that are relevant to the study that you are participating in. Please fill the questionnaire according to the instructions that are provided.

* Indica uma pergunta obrigatória

1. What is your name? *
   Please include your first and last names.

2. What is your email address? *

Skills

3. I am confident in my English skills. *
   Mark one of the circles.

   Marcar apenas uma oval.

   1 2 3 4 5

   Str Strongly Agree

4. I feel comfortable programming in Python. *
   Mark one of the circles.

   Marcar apenas uma oval.

   1 2 3 4 5

   Str Strongly Agree

5. I feel comfortable programming in JavaScript and Node.js. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

6. I feel comfortable working with Firefox. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

7. I feel comfortable working with Visual Studio Code. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

8. I feel comfortable working with Linux-based desktop-focused operating systems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

9. I feel comfortable using Visual Studio Code to edit Dockerfiles. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

10. I feel comfortable interacting with Docker through a terminal. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 1 Experience

11. I was able to understand what to do in Task 1. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

12. I was able to finish Task 1 without any problems. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 2 Experience

13. I was able to understand what to do in Task 2. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

14. I was able to finish Task 2 without any problems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 3 Experience

15. I was able to understand what to do in Task 3. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

16. I was able to finish Task 3 without any problems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Environment Considerations

17. I felt comfortable with the remote environment. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Additional Feedback

18. I felt like the tasks took a long time to carry out. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

19. I felt like the tasks would have been easier to carry out if I had access to other tools. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

20. Please write the name of any tool(s) that you believe would have helped you carry out the tasks.

_____

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

A.3 Experimental Questionnaire

Improving the Developer Experience of Dockerfiles

This questionnaire aims to evaluate your experience regarding aspects that are relevant to the study that you are participating in. Please fill the questionnaire according to the instructions that are provided.

~~* Indica uma pergunta obrigatória~~

1. What is your name? *

   Please include your first and last names.

2. What is your email address? *

Skills

3. I am confident in my English skills. *

   Mark one of the circles.

   Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

4. I feel comfortable programming in Python. *

   Mark one of the circles.

   Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

5. I feel comfortable programming in JavaScript and Node.js. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

6. I feel comfortable working with Firefox. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

7. I feel comfortable working with Visual Studio Code. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

8. I feel comfortable working with Linux-based desktop-focused operating systems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

9. I feel comfortable using Visual Studio Code to edit Dockerfiles. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

10. I feel comfortable interacting with Docker through a terminal. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 1 Experience

11. I was able to understand what to do in Task 1. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

12. I was able to finish Task 1 without any problems. *
Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

13. Dockerlive helped me do Task 1. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 2 Experience

14. I was able to understand what to do in Task 2. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

15. I was able to finish Task 2 without any problems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

16. Dockerlive helped me do Task 2. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Task 3 Experience

17. I was able to understand what to do in Task 3. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

18. I was able to finish Task 3 without any problems. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

19. Dockerlive helped me do Task 3. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Environment Considerations

20. I felt comfortable with the remote environment. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

Additional Feedback

21. I felt like the tasks took a long time to carry out. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

22. I felt like the tasks would have been easier to carry out if I had access to other tools. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

23. Please write the name of any tool(s) that you believe would have helped you carry out the tasks.

Additional Feedback about Dockerlive

24. Dockerlive made the tasks easy to carry out. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

25. Dockerlive was easy to use. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

26. Dockerlive's suggestions were useful in carrying out the tasks. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

27. I would like to use Dockerlive in the future. *

Mark one of the circles.

Marcar apenas uma oval.

	1	2	3	4	5	
Str	◯	◯	◯	◯	◯	Strongly Agree

28. Which of Dockerlive's features helped you the most? And why?

You can check the instructions we provided in the beginning, if you need to.

29. What problems did you run into while using Dockerlive to do the tasks?

30. If you could, how would you improve Dockerlive?

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

Appendix B

Dockerfiles

This appendix contains the Dockerfiles mentioned in Section 4.4.

```
FROM python

COPY . .

RUN pip install -r requirements.txt

RUN apt-get update && apt-get install -y httpie

RUN http http://example.com -o file.html

CMD python3 main.py
```

Listing B.1: Dockerfile for Task 1

```
FROM node

COPY package*.json .

COPY important_info.txt .

COPY important_info2.txt /

COPY file.js .

CMD node file.js
```

Listing B.2: Dockerfile for Task 2

```
1 FROM node
2
3 RUN apt-get update && apt-get install -y nodejs
4
5 COPY . .
6
7 CMD node index.js
```

Listing B.3: Dockerfile for Task 3