FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Live Docker Containers

David Alexandre Gomes Reis

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Filipe Figueiredo Correia, Assistant Professor

July 27, 2020

Live Docker Containers

David Alexandre Gomes Reis

Mestrado Integrado em Engenharia Informática e Computação

July 27, 2020

Abstract

The use of containerization technologies for software development, such as Docker, is now widespread, with over 70000 Dockerfiles being found in projects from the GitHub platform as of October 2016. The use of containerization provides a secure, portable and efficient environment where applications can be executed.

Currently, the usual workflow of a developer configuring a Docker environment consists of writing a Dockerfile, building the Dockerfile into a Docker image, instantiating that Docker image in a Docker container and verifying if the container is working as expected (using a tool or the command-line). If the container is not behaving as expected, then the developer has to make changes to the Dockerfile and repeat the process, until the desired behaviour is achieved. A survey, answered by students and professionals, showed that this process is often perceived as time-consuming.

Live programming refers to the ability to obtain continuous feedback on a program while that program is being developed. The level of liveness in IDEs is related to the type and update frequency of the feedback provided. Currently, the only Dockerfile development tools which provide live feedback are some static analysis tools. Therefore, by increasing the live feedback available in the Dockerfile development environment we can shorten the feedback loop and reduce the time spent in Dockerfile development.

We propose an approach where dynamic analysis is performed on the Dockerfile under development and the results of that analysis are displayed and updated in the IDE as the developer edits the Dockerfile. By automatically collecting and presenting live dynamic feedback about the Dockerfile under development, the developer has faster access to potentially helpful information than in a non-live environment.

This approach is implemented as Dockerlive: an extension for Visual Studio Code which provides live dynamic feedback for Dockerfile development. Dockerlive automatically builds, instantiates and extracts information from a Docker image and a Docker container as the developer edits a Dockerfile, providing continuous feedback on the changes that the developer makes. The implementation of Dockerlive follows the reference architecture described in this dissertation.

In order to measure the impact of the presence of live dynamic feedback in a Dockerfile developer's performance, a controlled experiment with users has been designed and conducted. This experiment showed evidence that the presence of live dynamic feedback in the IDE can significantly improve the efficiency of developers working in Dockerfiles and significantly impact their behaviour during the development process.

Keywords: Docker, Dockerfile, Containerization, Live Programming, IDE

Resumo

O uso da tecnologias de *containers* no desenvolvimento de software, como *Docker*, é agora generalizado, tendo sido encontrados acima de 70000 *Dockerfiles* em projetos da plataforma GitHub à data de outubro de 2016. Estas tecnologias têm a vantagem de proporcionar um ambiente seguro, isolado e eficiente onde se podem executar aplicações.

Atualmente, os passos que um programador segue para configurar um ambiente em *Docker* são escrever um *Dockerfile*, compilar o *Dockerfile* numa *Docker image*, instanciar essa *image* num *Docker container* e verificar se o *container* funciona como desejado. Se o *container* não estiver a funcionar como desejado, o programador terá de alterar o *Dockerfile* e repetir este processo, até que o comportamento desejado seja alcançado. Este processo pode ser lento, baseado em tentativa e erro, pelo que se pode tornar demorado, como comprovado através de um inquérito realizado com alunos e profissionais com experiência em Docker.

Live programming refere-se à capacidade de obter *feedback* contínuo sobre um programa enquanto este é desenvolvido. O nível de *liveness* em *IDEs* corresponde ao tipo e à frequência de atualização do *feedback* exibido. Atualmente, as únicas ferramentas de desenvolvimento em *Dockerfile* que fornecem *live feedback* são algumas ferramentas que apenas fazem análise estática. Deste modo, ao aumentar o *live feedback* disponível no ambiente de desenvolvimento de *Dockerfiles*, é possível encurtar o ciclo de *feedback* e reduzir o tempo gasto no desenvolvimento de *Dockerfiles*.

Propomos uma abordagem em que se executa análise dinâmica no *Dockerfile* em desenvolvimento e os resultados dessa análise são exibidos e atualizados no ambiente de desenvolvimento à medida que o *Dockerfile* é editado. Ao gerar e exibir *live feedback* dinâmico sobre o *Dockerfile* em desenvolvimento, o programador tem acesso mais rápido a informação potencialmente útil do que num ambiente sem este tipo de *feedback*.

Esta abordagem é implementada em *Dockerlive*: uma extensão para *Visual Studio Code* que fornece *live feedback* dinâmico durante o desenvolvimento em *Dockerfile*. O *Dockerlive* compila, instancia e extrai informação a partir de uma *Docker image* e de um *Docker container* à medida que um *Dockerfile* é editado, fornecendo *feedback* contínuo sobre as alterações que são feitas. A implementação do *Dockerlive* segue a arquitetura de referência descrita nesta dissertação.

De forma a medir o impacto da presença de *live feedback* dinâmico na eficiência de um programador de *Dockerfile*, uma experiência com utilizadores foi desenhada e realizada. Esta experiência mostrou indícios de que a presença de *live feedback* dinâmico no ambiente de desenvolvimento pode aumentar significativamente a eficiência de programadores de *Dockerfile* e pode ter um impacto significativo no seu comportamento durante o processo de desenvolvimento.

Acknowledgements

I would like to express my gratitude and appreciation to my supervisor, Prof. Filipe Correia, as well as Prof. João Pedro Dias, for always being available to shine the beacon of wisdom through the unclear path that a dissertation entails.

I would also like to express my gratitude to all the teachers who, over the course of my years at *Faculdade de Engenharia da Universidade do Porto*, have constructively challenged me to overcome unique obstacles and learn valuable lessons.

To my family, especially my mother, father and sister, thank you for the constant support over the years, for always allowing me to pursue my ambitions and, above all, for shaping me into who I am today and teaching me kindness, fortitude and discipline.

I would also like to thank my friends and colleagues who, over the years, have stayed by my side and helped me relieve the tensions of life.

Finally, I would like to thank Cláudia, for being my partner in crime and having the world record on how fast anyone can make me laugh.

David Alexandre Gomes Reis

“Out of clutter, find simplicity;
From discord, make harmony;
In the middle of difficulty, lies opportunity.”

John A. Wheeler on Albert Einstein’s rules of work

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
AUFS	Advanced Multi-layered Unification File System
COW	Copy-On-Write
CPU	Central Processing Unit
CSV	Comma-Separated Values
Dev	Development
DSL	Domain Specific Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
ID	Identifier
IP	Internet Protocol
IT	Information Technology
JDT	Java Development Tools
JS	Javascript
JSON	Javascript Object Notation
LSP	Language Server Protocol
NFS	Network File-system
Ops	Operations
OS	Operating System
RAM	Random-Access Memory
REPL	Read-Eval-Print Loop
RPC	Remote Procedure Call
RQ	Research Question
TCP	Transmission Control Protocol
TF	Temporary File
TSV	Tab-Separated Values
UI	User Interface
URL	Uniform Resource Locator
VS	Visual Studio
VSCode	Visual Studio Code
YAML	YAML Ain’t Markup Language

Chapter 1

Introduction

This chapter presents a general overview of this dissertation. Section 1.1 provides some context regarding the technologies and fields of work most relevant to this dissertation. Section 1.2 introduces the main issues and challenges that currently exist in Dockerfile development, with Section 1.3 focusing on the issues that this project intends to solve, as well as the objectives and general approach. Section 1.4 presents the main contributions of this dissertation. Finally, Section 1.5 provides some guidance on the structure of this document.

1.1 Context

DevOps is a set of practices which focus on promoting communication and cooperation between development teams (Dev) and operations teams (Ops), as well as employing agile development methodologies, with the purpose of optimizing the process of deployment. This optimization consequently allows a company to adapt and respond faster to its needs [13]. In order to allow the Dev and Ops teams to develop the infrastructure in an agile way, solutions which allow the infrastructure to be defined using code have been developed and are currently available.

Containerization, particularly with Docker [26], has become a widely adopted technology which enables the practice of DevOps [19, 35]. Docker allows developers to define the desired infrastructure in a file called Dockerfile, which can then be built into a Docker image and instantiated as a Docker container [5]. All the instructions specified in the Dockerfile, including the development team's product, are executed inside the container. This allows developers to deploy their products in an environment with the following characteristics [33]:

- **Secure** — Applications run inside a container with strictly restricted access to the host's resources.
- **Portable** — A Dockerfile can create the same container under different host environments and does not pollute the host machine with dependencies.

- **Efficient** — As opposed to virtual machines, containers do not require an hypervisor, which generally results in increased performance.

Despite these benefits, development with Docker containers is not perfect, since the Dockerfile development process can be slow [15]. This dissertation aims at employing live software development techniques with the objective of improving the current state of Dockerfile development. Live software development is often associated with live programming, where a developer, usually programming with a general-purpose programming language, has live feedback in his development environment. However, liveness can be applied to other areas of software development, such as testing or deployment [1]. The solutions and approaches presented in this dissertation go beyond live programming by providing live dynamic feedback in the development environment of the Dockerfile language, which is usually used in the deployment phase of software development [33].

1.2 Current Problem

Developing a Dockerfile can be a hard and slow process [15]. According to the analysis performed in Section 3.1, there are some tools available which can help developers working on Dockerfiles, such as static analysers (e.g. linters) or infrastructure testing tools.

However, most of these tools don't provide feedback during development in an automatic way and, instead, must be manually launched by the developer. In other words, the majority of the existing solutions don't provide live feedback to the developer. Providing live feedback is a technique that has proven to be effective at reducing the time required for a developer to fix bugs [22], since the developer has access to information about his code more often and with less effort. The tools which do provide some live feedback about Dockerfiles only provide static analysis information, leaving most of the debugging process up to the developer.

We believe that the lack of live dynamic analysis feedback in the IDE leads developers to work in an iterative workflow which can be slower than desired. The survey presented in Chapter 4, answered by students and professionals with some experience in Dockerfile development, further strengthens this belief by showing that developers regard most activities of the Dockerfile development process as considerably time-consuming.

A more detailed description of the problems that this work intends to tackle is presented in Section 5.1.

1.3 Motivation and Objectives

This dissertation attempts to solve some of the issues in the Dockerfile development process by implementing a live development environment for Dockerfiles.

Live development environments aim at providing continuous feedback when developing software, reducing the typical 4-stage development cycle (edit, compile, link, run) to a single stage development cycle. Live development environments can be classified according to their liveness, which represents the extent and responsiveness of the feedback that is provided to developers [36]. Steven L. Tanimoto proposes a 6-level liveness classification system. This system is in ascending order, meaning that each level has higher liveness than the previous [37].

Given the current issues that exist within the Dockerfile development process and the benefits that live programming can potentially bring to it, there is an argument to be made that by bringing more live feedback to the developer's environment it would be possible to mitigate some of these issues and reduce the time required to develop a Dockerfile.

As such, the main objective of this dissertation is to provide live dynamic feedback to developers working with Dockerfiles and analyse the impact of the increased live feedback in the developer's performance and behaviour.

Section 5.2 provides a deeper description of these topics, including the main hypothesis of this dissertation.

1.4 Contributions

This dissertation aims to improve the current Dockerfile development process and analysing the impact of liveness in the developer's IDE. In order to achieve these objectives, the following contributions are made:

- A survey, answered by students and professionals, with the purpose of identifying the main issues that exist in the Dockerfile development process.
- A novel approach which enables the delivery of live dynamic feedback to a developer working on Dockerfiles.
- A thoroughly documented reference architecture for the aforementioned approach, implemented as an extension for the Visual Studio Code IDE.
- A controlled experiment with users which analyses the performance and behaviour of developers working with and without an implementation of the reference architecture. This user study has the the purpose of measuring the impact of a higher level of liveness in the efficiency of developers working in Dockerfiles.

1.5 Document Structure

This chapter focuses on providing an overview of the concepts which support this dissertation, as well as the problem it attempts to solve, its motivations and its objectives.

Chapter 2 presents the concepts and technologies which are relevant to this dissertation, focusing mainly on containerization and live programming.

Chapter 3 analyses and documents the state of the art tools and solutions from two distinct areas of software development: working on Dockerfiles and live programming environments. The analysis of the current Dockerfile work environment focuses on identifying and categorizing tools and solutions which help a developer working on Dockerfiles, paying special attention to the liveness levels of each tool. The analysis of live programming environments aims at documenting different approaches to liveness and feedback in IDEs, as well as identifying the main threats to the success of a live environment. Ultimately, this chapter intends to explore the current development environment that is available to Dockerfile developers.

Chapter 4 describes a survey answered by 68 students and 109 developers with the main objective of understanding the current Dockerfile development process. For this purpose, the survey focuses on assessing which activities of Dockerfile development are perceived as time-consuming and understanding the approaches that developers most use to work on a Dockerfile.

After analysing the tools and approaches that Dockerfile developers have at their disposal, as described on Chapter 3, and understanding the current Dockerfile development process through the survey described in Chapter 4, Chapter 5 focuses on describing the main problems detected in the Dockerfile development process and presenting the main hypothesis that this dissertation aims to validate.

Chapter 6 describes an approach which aims to bring live dynamic analysis feedback to the Dockerfile development environment. This chapter also describes a reference architecture and implementation of this approach.

Chapter 7 presents a controlled experiment with users performed with the aim of analysing the impact of live dynamic feedback in the Dockerfile development environment. The methodology, data collection techniques, data analysis and threats to validity are discussed in this chapter.

Chapter 8 presents the conclusions of this document, addressing the main hypothesis and providing an overview of the work and contributions performed. Finally, this chapter also presents the work to be developed in the future with the aim of expanding the work performed in this dissertation.

Chapter 2

Background

This chapter presents relevant concepts and technologies for this dissertation. Section 2.1 describes the concepts of infrastructure and DevOps, describing Docker's general architecture with particular emphasis on the Docker image building process and layered storage system. Section 2.2 describes the concepts of live programming and liveness, as well as describe some of its potential benefits and shortcomings.

2.1 Infrastructure and DevOps

Infrastructure can be defined as the characteristics and configurations of underlying technologies which allow software to run and provide the features intended by the stakeholders [2]. Being able to develop and deploy a software product in a reliable, fast and reproducible way can be a very important objective for companies nowadays, since it allows a company to quickly ship changes to its products. Most of the time, this can be is achieved through a symbiotic relationship between the development team (which focuses on developing software) and the IT operations team (which focuses on building and preparing the infrastructure), with continuous communication and cooperation. By combining this communicational practice with agile development methods, which promote fast development and deployment of software, the aforementioned objective can be achieved. This organizational approach to the development and deployment of software and infrastructure is defined as **DevOps**. The term DevOps refers to the development team ("Dev") and the IT operations team ("Ops"), although this approach can be applied to other teams, such as teams focused on security and networking [2, 32, 13].

2.1.1 Containerization with Docker

A container is an independent and isolated environment which runs within a host machine and is strictly configured to use a restricted amount of its host's resources. There are multiple implementations of this concept, but in recent years Docker [26] has become a *de facto* standard [42]. Docker is a platform which allows developers to specify the desired container configuration in a file called the Dockerfile. By using Docker to specify the desired application infrastructure within a container, teams are able to deploy their products in an environment with the following characteristics [33]:

- **Secure** — Applications run inside a container with strictly restricted access to the host's resources.
- **Portable** — A Dockerfile can create the same container under different host environments and does not pollute the host machine with dependencies.
- **Efficient** — As opposed to virtual machines, containers do not require an hypervisor, which generally results in increased performance.

These characteristics, combined with the practice of DevOps, allow the teams to develop their infrastructure safely in a similar way to the development of the application code — using agile methodologies and version control [19].

In recent years, cloud solutions have seen a large rise in popularity, with over 49% of the world's population having access to at least one Internet enabled device [34]. Containers can be very useful in this type of solutions since the characteristics of the containerization pattern enable an isolated and programmatic deployment at scale [34, 35].

In addition, the isolation that containers provide between the application's dependencies and the host machine reduces the potential incompatibility between different containerized applications, since each one runs within it's own environment and dependencies [33].

A common alternative to containers in the deployment of some cloud solutions is virtualization. However, when compared to virtualization, containerization can offer a performance advantage, reduced startup time and smaller deployment size. This makes containerization better suited for some cloud solutions where performance is a key requirement [12, 5].

```
FROM node:13.1.0

WORKDIR /app/
COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3333

ENTRYPOINT ["node","index.js"]
```

Figure 2.1: Example Dockerfile. Generates a container with NodeJS v13.1.0 and dependencies installed.

Figure 2.1 presents an example of a Dockerfile which launches a NodeJS v13.1.0 application. It starts by defining the base image using the `FROM` instrution, which specifies that the base image should be the *node* image at the version 13.1.0. This image, which is retrieved from DockerHub [1], comes with NodeJS preinstalled. Instructions on line 3 to 8 are responsible for installing the required dependencies to run the NodeJS application. The instruction `EXPOSE` exposes a port from the container to the host machine. Finally, the instruction `ENTRYPOINT` specifies the command which will be executed when the container is raised. In this case, it starts the NodeJS application.

2.1.2 Docker Engine

The Docker Engine consists of the Docker containerization technology along with some tools and interfaces to help developers manage the Docker ecosystem. It can be divided in three main components [2]:

- **dockerd** — Daemon which runs in the background and directly controls images, containers and other artifacts.
- **Engine API** — Interface which allows other applications to interact with dockerd.
- **'docker' command** — Command line tool which allows a user to manage Docker artifacts.

Using Docker, developers can define an environment by writing instructions in a file, called Dockerfile. A Dockerfile can then be compiled into a Docker image, which contains a file system resulting from the build process as well as an `ENTRYPOINT` instruction. A developer can then instantiate the Docker image as a Docker container, which will load the image file system and execute the `ENTRYPOINT` instruction.

With these components, developers can not only use the command line but also develop their own programs which communicate with the Engine API in order to interact with containers and images and perform actions such as building a Docker image from a Dockerfile or stopping a running container [16].

2.1.3 Docker Image Building and Storage

Docker uses a Copy-On-Write (COW) mechanism, meaning that each layer of the image only contains the files that the layer creates or modifies [24]. During the image building process, the instructions present in the Dockerfile are read sequentially and each instruction in the Dockerfile corresponds to a layer in the image. As such, layers are sequentially created and each layer only stores the file system changes it performs [15]. For example, the layer generated from an `ADD` instruction, which adds files from the host machine to the image, will contain just the added files.

This mechanism can reduce the image building time in some situations, since layers which have not been changed do not need to be rebuilt. For example, if the programmer changes an

[1] *node* image in DockerHub available at `https://hub.docker.com/_/node/`

[2] Docker Overview — Docker Documentation available at `https://docs.docker.com/engine/docker-overview/`

instruction on line 8 of the Dockerfile, Docker can take the last intermediate layer generated before line 8 and only execute instructions from line 8 onward. Every change to a file or directory during the container's execution is also written to a new layer which does not belong to the Docker image, meaning that other instances of the same image will not have those changes. This also saves disk space since files are only copied when modified, avoiding data duplication [15].

This mechanism is possible since Docker containers use a union file system such as AUFS [11] or OverlayFS [27] which are able to merge different layers in a process called *union mount* [31, 11, 8]. Figure 2.2 represents how the COW system is used in order to allow each image layer to be as small as possible and to allow multiple isolated containers to use the same image without duplicating data.

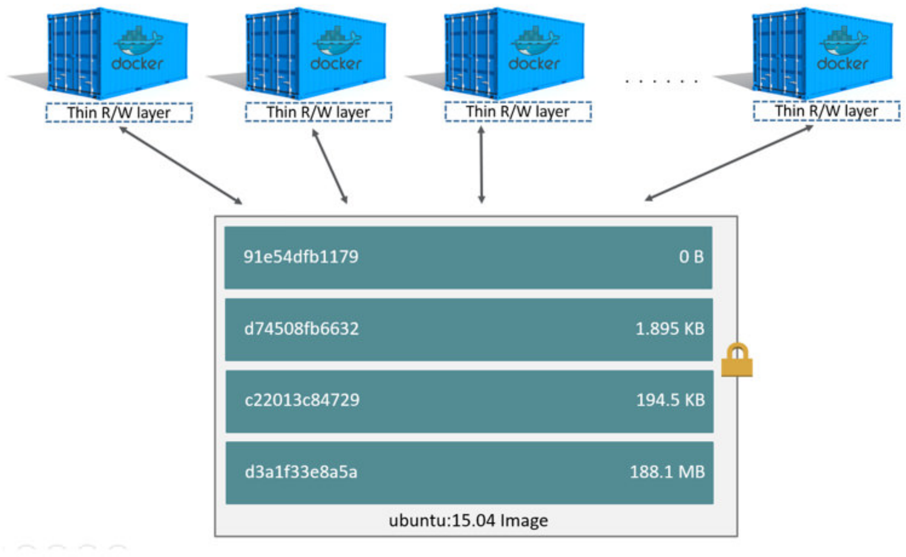


Figure 2.2: Visual representation of the COW system in Docker [a].

[a]Taken from https://docs.docker.com/storage/storagedriver/

However, despite the benefits provided by this mechanism, the image building process can still be slow and disrupt the developer's workflow [15].

2.2 Liveness and Live Programming

The concept of liveness refers to the ability to modify a running program, providing constant accessibility to evaluation and nearly instantaneous feedback on actions performed by a user in an environment. If an environment is able to provide these features, it can be considered a live environment [1].

Although the concept of liveness is commonly associated with programming (i.e. live programming) [37], liveness can be applied to other activities of software development, such as testing or deployment [1], or even to live artistic performances [4]. The solutions and approaches presented in this dissertation go beyond live programming by providing live dynamic feedback in the development environment of the Dockerfile language, which is usually used in the deployment phase of software development [33].

Live programming environments aim to continuously update and provide feedback about a system as it is being developed. Live programming can be achieved by constantly executing the developer's code, updating the running program any time the source code is changed, and providing feedback to the developer according to the changes verified in the program's execution. Ideally, this changes the software development cycle from a 4-stage process (*edit*, *compile*, *link*, *run*) to a single-stage cycle [37]. Since the purpose of live programming is to constantly keep the developer informed about the system under development, feedback is usually tightly coupled to environment where the code is written, either as a standalone IDE or as a plugin to an IDE [23]. Read-eval-print loops (REPLs) can also provide some liveness, since the developer obtains regular feedback as the code is entered in the shell [40].

Following these principles, live programming can promote **technical agility**, as the short and frequent feedback loop it creates allows developers to practice the *inspect and adapt* methodology used in agile programming [17].

The concept of live programming has been implemented in various fields of work, such as 3D graphics programming [10], general-purpose programming [22] and data science [41]. A deeper analysis of multiple live programming solutions is performed in Section 3.2.

Steven L. Tanimoto proposes a classification system for live programming environments according to the liveness that it provides. Tanimoto defines liveness as the ability to modify a running program, taking into consideration the responsiveness and timeliness of the feedback provided by the environment. The system initially consisted of 4 levels [36] but has since been extended to 6 levels [37], where each level can be considered to provide more liveness than the previous level. The 6 levels are [37]:

- **Liveness Level 1** — Static visualization, such as a flowchart, with the ability to inform.
- **Liveness Level 2** — In this level, the developer can perform an action and manually request feedback. Feedback may arrive some time after the request is performed. For example, in an environment with this level of liveness, a developer can manually compile and execute a program and then observe the program's output in order to understand if it's behaving as expected.
- **Liveness Level 3** — In this level, the system waits for the developer to finish a section of code. When the system detects that a section of code has been finished, it automatically updates the running program with the new code and automatically provides feedback relative to the updated execution. For example, in an environment with this level of liveness, a developer can trigger an update of the program and obtain feedback just by changing and saving a file in the IDE.
- **Liveness Level 4** — In this level, the system continuously updates the running program with changes in the source code as they are introduced by the developer. In contrast with

the previous level, this level does not need an implicit trigger to update the running program, doing so continuously. As a consequence, the feedback provided is also continuously updated.

- **Liveness Levels 5 and 6** — In these levels, the system integrates some predictive capabilities, proactively generating multiple possible code changes, executing those changes and providing feedback accordingly. Possible code changes could be generated using machine learning algorithms, which try to predict the most likely code changes that the developer will perform next.

Working in a live programming environment has 3 main expected benefits [1]: **immediacy**, **exploration** and **stability**. Immediacy refers to the fact that feedback about the actual state of the program is delivered immediately during development in a live environment. Exploration refers to the fact that it's easier for a developer to quickly explore the system under development. Immediacy and exploration are two advantages which go hand in hand, since by having immediate feedback about the actual behaviour of their program, instead of having to speculate on the program's behaviour after a change, developers can swiftly explore and iterate on different ideas. This can help developers converge more quickly to a better solution. Since the behaviour of the system under development is constantly being observed it can also be easier to stabilize it (i.e. ensure that the behaviour of the system is as expected for every possible input state) [1].

However, there can be obstacles to the success of a live environment, such as [1]: **increased complexity**, **unsuitable abstractions**, **transient semantics** and **lack of interoperability with existing tools**. By providing constant feedback to the developer there is an increased amount of data for the developer to process, which can inadvertently increase the complexity of the developer's task. On the other hand, the abstractions used by the live programming environment may not be useful for the system under development, especially when the system's scale increases. Transient semantics may also be a problem, since the system under development might often be in a state that is not safely executable, making it hard to generate valuable live feedback from dynamic analysis. This may interrupt the ideally continuous feedback loop. Furthermore, some tools, such as external service APIs, may not be well suited for the repeated execution required by live programming [1].

Given the characteristics of live environments described in this section, there are a few different motivations for the implementation of liveness in programming. Some of these motivations are [30]:

- **Accessibility** — Make programming more accessible, particularly to less experienced programmers.

- **Comprehension** — Facilitate the comprehension of the system under development.

- **Exploration** — Enabling programmers to explore the system under development.

- **Productivity** — Increase the productivity of the developer, often through accelerating the execution of some development activities.

These motivations are not mutually exclusive and can even enable each other. For example, accessibility is commonly achieved through a facilitated comprehension of the system under development [30].

Chapter 3

State-of-the-Art

This chapter analyses and categorizes existent solutions in two different (albeit interconnected) topics. Section 3.1 analyses and categorizes existent tools and solutions which provide helpful information or functionalities to developers working on Dockerfiles, describing their features and level of liveness. Section 3.2 analyses existent development environments, taking into account their liveness and the mechanisms they use to deliver feedback to the developer.

3.1 Working with Dockerfiles

After understanding and exploring the concepts introduced in Chapter 2, this section presents the current state-of-the-art practices used by developers when writing a Dockerfile. Since those practices rely heavily on the features and characteristics of the tools available to developers, an analysis and categorization of those tools is performed and documented in this section. Each section explores a category of tools which represents a set of functionalities that can help a developer working on Dockerfiles. Since some of those functionalities could potentially benefit from having a higher level of liveness, one of the main objectives of this analysis is to evaluate the level of liveness of each tool, according to the hierarchy proposed by Tanimoto [37] and described in Section 2.2. Tools can enable multiple practices which belong to different categories, which means that some tools are listed under multiple categories.

In sum, this analysis aims to help answer the following questions:

- **What features are currently not available within an IDE?**
- **What level of liveness do these tools and features provide?**
- **What development practices do these tools enable?**

The tools analysed in these sections were found by searching on the platforms Google, GitHub, Visual Studio marketplace, JetBrains Plugins Repository, Scopus and IEEE Xplore, using the search queries *"Docker"*, *"Dockerfile"* and *"Docker container"*. If multiple results were deemed too similar in terms of features and characteristics, the star count (GitHub) and download count (Visual Studio marketplace and JetBrains Plugins Repository) were used to select one of the tools. For the results provided by the platforms Google, GitHub, Visual Studio marketplace and JetBrains Plugins Repository, the documentation provided was used in order to analyse each tool. In the cases where the documentation was insufficient, the tools were installed and tested manually. For the results provided by Scopus and IEEE Xplorer, the article's full-text was used in order to analyse each tool. A total of 37 tools have been analysed and placed in one or more of the created categories:

- Container Status
- Performance Monitoring
- Container Management
- Infrastructure Testing
- Static Analysis
- Image Build Optimization

3.1.1 Container Status

Some tools have as their main goal to display information about the status of one or more containers. The different types of information that these tools gather and display to the user are:

- **Running status** — Show if a container is running, stopped or restarting.
- **Ports** — Show container port mappings.
- **Log** — Show the output log of a container.

Most of the tools analysed provide these types of information since they are some of the most basic information that can be retrieved from a container. This information can be useful to identify potential problems. For example, a developer who just instantiated a container that should run a HTTP server can use one of these tools to check if the container is in fact running, or if it isn't, alerting the developer to a potential crash. Log visualization can also be very useful since the programs and services running inside a container could be outputting valuable information to the container's logs.

Figure 3.1 shows a screen capture of the plugin Docker [1] for the Intellij IDEA IDE [1]. This plugin presents container status information graphically to the developer. The icons on the left side of the container's name are used to represent whether the container is running, stopped or restarting. The user can select a container from the list and obtain the port bindings and logs

[1]Links to the mentioned tools can be found in Appendix A

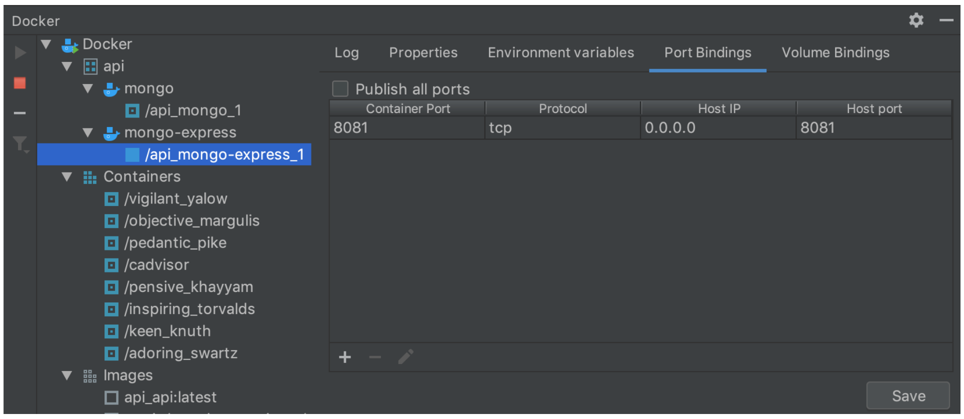
Table 3.1: Comparison of analysed Container Status tools. Links to the mentioned tools can be found in Appendix A.

Tool	Liveness Level			IDE Plugin
	Running Status	Ports	Log	
Visual Studio Container Tools Extensions (Preview)	2	2	2	VS
Docker	2	2	2	Intellij IDEA
Docker for Visual Studio Code (Preview)	2	—	2	VSCode
Docker Explorer	2	—	—	VSCode
Docker WorkSpace	2	—	—	VSCode
Docker Runner	2	—	—	VSCode
Haven	2	2	2	No
DockStation	2	2	2	No
Portainer	2	2	2	No
Seagull	2	2	—	No
Dozzle	2	—	2	No
docker_monitoring_logging_alerting	2	—	2	No
Weave Scope	2	—	2	No
Dockeron	2	—	2	No
ctop	2	—	2	No
lazydocker	2	—	2	No
Wharfee	2	—	2	No
cAdvisor	2	—	—	No
AppOptics	2	—	—	No
Docker-Alertd	2	—	—	No
monit-docker	2	—	—	No
Captain	2	—	—	No

for the selected container, by clicking on the *Port Bindings* and *Log* tab respectively. The plugin retrieves this information by performing requests to the Docker Engine [2], which must be running in the target machine.

Table 3.1 synthesizes the result of analysing a set of container status tools according to the level of liveness that they enable. As an example, for the practice of checking the running status of a container, every tool provides liveness at the level 2. This table shows that, to the best of our knowledge, there are no tools which provide a liveness level above 2 in this category, since the identified tools do not react autonomously to changes in the Dockerfile. Despite the lack of high levels of liveness, we can also see that there are tools which enable all the practices identified in this category in an IDE.

[2]Links to the mentioned tools can be found in Appendix A



- Docker
  - api
    - mongo
      - /api_mongo_1
    - mongo-express
      - /api_mongo-express_1
  - Containers
    - /vigilant_yalow
    - /objective_margulis
    - /pedantic_pike
    - /cadvisor
    - /pensive_khayyam
    - /inspiring_torvalds
    - /keen_knuth
    - /adoring_swartz
  - Images
    - api_api:latest

☐ Publish all ports

Container Port	Protocol	Host IP	Host port
8081	tcp	0.0.0.0	8081

Figure 3.1: Docker plugin for Intellij IDEA

3.1.2 Performance Monitoring

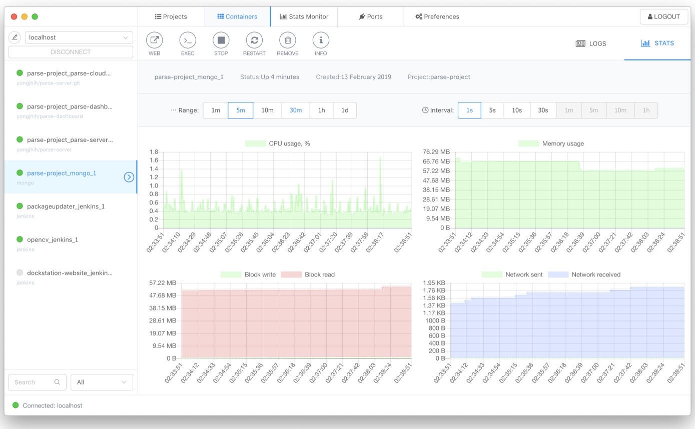


Figure 3.2: Dockstation performance monitoring [a].

[a]Taken from Dockstation documentation listed in Appendix A.

Containers can use some of the resources of the host machine, such as CPU and RAM usage, secondary memory usage and network bandwidth. If the developer knows that the desired behaviour from a container should trigger a certain level of resource usage, analysing the resource usage of the container can provide helpful insight into whether the container is working as intended. For example, a developer starts a container which should have a RAM usage of 1GB when running as intended. If the reported secondary memory usage is different than the expected, then it can be concluded that the container is not running as intended. Figure 3.2 shows a screen capture from the tool Dockstation, which collects and displays information about the CPU and RAM usage, secondary memory reads/writes and network activity [3].

[3]Links to the mentioned tools can be found in Appendix A

Table 3.2: Comparison of analysed Performance Monitoring tools. Links to the mentioned tools can be found in Appendix A.

Tool	**Liveness Level Monitoring**				**Alerting**	**IDE Plugin**
	CPU	**RAM**	**Secondary Memory**	**Network**		
Sysdig	2	2	2	2	Yes	No
AppOptics	2	2	2	2	Yes	No
monit-docker	2	2	2	2	Yes	No
docker_monitoring _logging_alerting	2	2	2	2	Yes	No
docker-alertd	2	2	—	—	Yes	No
cAdvisor	2	2	2	2	No	No
DockStation	2	2	2	2	No	No
Haven	2	2	—	2	No	No
ctop	2	2	—	2	No	No
Portainer	2	2	—	2	No	No
Weave Scope	2	2	—	—	No	No
lazydocker	2	2	—	—	No	No
Dockeron	—	—	2	—	No	No

Some of these tools also allow the developer to configure certain actions or alerts which are triggered automatically, based on the information extracted from a container. For example, this allows a developer to automatically send an email to himself when the CPU usage of a container goes above 95%, or to automatically restart a container if the container crashes.

In order to provide performance related feedback, tools usually start by checking if the target container is running and display that information to the developer. Since checking if a container is running is one of the types of information that is categorized in Section 3.1.1, which analyses container status tools, every tool categorized as a performance monitoring tool is also categorized as a container status tool.

Table 3.2 synthesizes the result of analysing a set of performance monitoring tools according to the level of liveness that they enable. For example, for the practice of monitoring the RAM usage of a container, every tool provides liveness at the level 2. This table shows that, to the best of our knowledge, there are no tools which provide a liveness level above 2 in this category, since the identified tools do not react autonomously to changes in the Dockerfile. It's also worth noting that there are no tools in this category that run within an IDE, as plugins. This is possibly because these tools are of greater importance during operations, to continuously inspect the status of a system in production, and their use during development is not as widespread. However, as mentioned, this information and features could potentially be useful during development.

3.1.3 Container Management

Some tools aim at providing the ability to easily manage the state of the containers in a local machine, offering the options to start, stop, build and remove a local container. These tools enable the following practices:

- **Start container** — Start an existing container
- **Stop container** — Stop an existing container
- **Build container** — Build a container from a Dockerfile
- **Attach Shell** — Attach an interactive shell into the container

These tools are useful to a developer working on Dockerfiles since without them, container management relies on the built-in Docker console commands which take time and effort to write. Therefore, these tools provide the ability to manage the containers on your machine in a easier and faster way. Figure 3.3 shows a screen capture of the Docker plugin for Intellij IDEA, which provides these features in the IDE. This tool allows a developer to start, stop and remove an existing container in the developer's system. However, it does not allow a developer to build a container from a Dockerfile and does not perform any actions automatically as the developer edits the code. This means that the developer must edit the Dockerfile and manually use the tool in order to progress in his development process. On the other hand, the fact that this tool is available directly in the IDE, where the developer writes the Dockerfile, is a valuable aspect that contributes to a more fluent work environment.

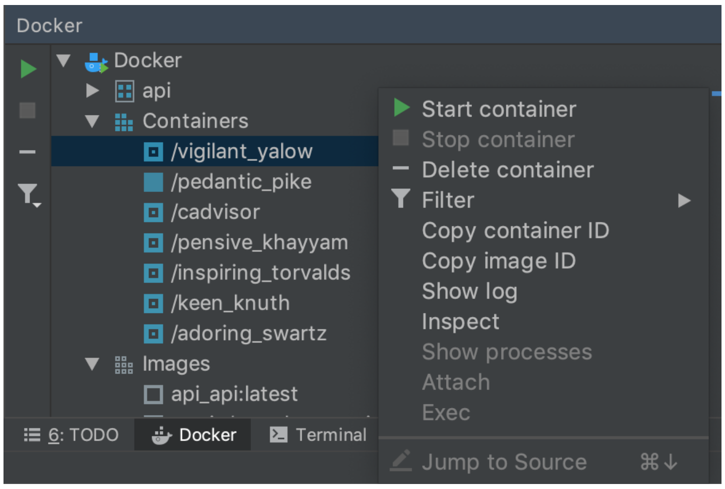


Figure 3.3: Docker plugin for Intellij IDEA

Table 3.3 synthesizes the result of analysing a set of tools according to the level of liveness that they enable in a given practice. For example, in the practice of starting a container, every tool provides liveness at the level 2. This table shows that, to the best of our knowledge, there are no tools which provide a liveness level above 2 in this category, since the identified tools do not react autonomously to changes in the Dockerfile. Despite the lack of higher levels of liveness, we can also see that there are tools which enable all the practices identified in this category in the IDE of the developer.

Table 3.3: Comparison of analysed Container Management tools. Links to the mentioned tools can be found in Appendix A.

Tool/Plugin	**Liveness Level**				**Attach Shell**	**IDE Plugin**
	Start	**Stop**	**Remove**	**Build**		
Docker	2	2	2	2	Yes	Intellij IDEA
Visual Studio Container Tools Extensions (Preview)	2	2	2	—	Yes	VS
Docker for Visual Studio Code (Preview)	2	2	2	—	Yes	VSCode
Wharfee	2	2	2	2	Yes	No
Docker.el	2	2	2	2	Yes	No
DockStation	2	2	2	—	Yes	No
Portainer	2	2	2	—	Yes	No
Weave Scope	2	2	—	—	Yes	No
GoInside	—	—	—	—	Yes	No
Docker Runner	2	2	—	2	No	VSCode
Docker Explorer	2	2	—	—	No	VSCode
Captain for Mac	2	2	2	—	No	No
Dockeron	2	2	2	—	No	No
Seagull	2	2	2	—	No	No

3.1.4 Infrastructure Testing

Infrastructure testing solutions aim at allowing developers to execute tests on an infrastructure configuration. The Docker development process can benefit from these tools, since they can be used to test whether an infrastructure configuration, written in a Dockerfile, is generating a container which behaves as intended [6].

RSpec. This tool allows developers to write tests in the Ruby programming language. It was created for the purpose of allowing for behaviour-driven development, since it allows for developers to clearly state the intended behaviour within a test [4].

Serverspec. This tool, which extends RSpec, allows developers to write RSpec tests in a Ruby domain specific language (DSL) to verify if a server is working as intended [4]. As an example, Figure 3.4 shows a test which verifies if the port 3333 of the target server is listening to *tcp6* traffic. Despite being directed at servers, Serverspec provides a wide range of testable resource types [5] which make it useful for other kinds of systems.

[4]Links to the mentioned tools can be found in Appendix A

[5]List of Serverspec Resource Types available at https://serverspec.org/resource_types.html

```
describe port(3333) do
  it { should be_listening.with('tcp6') }
end
```

Figure 3.4: Serverspec test which verifies if the port 3333 is listening to *tcp6* traffic

Chef Inspec. This tool is a testing framework which was initially an extension to Serverspec but is now a separate standalone tool. Inspec allows developers to write tests in a Ruby DSL named Chef InSpec DSL. These tests allow a developer to verify the infrastructure of a system [6]. Figure 3.5 shows a test written in the Chef InSpec DSL which verifies if the port 3333 of the target server is listening to *tcp* traffic.

```
describe port(3333) do
    it { should be_listening }
    its('protocols') {should include 'tcp'}
end
```

Figure 3.5: Inspec test which verifies if the port 3333 is listening to *tcp* traffic

Chef Inspec and Serverspec are very similar in their functionality, but support different types of resources [7].

Goss. This tool allows the validation of a server configuration. Tests are specified in the YAML language [6]. As an example, Figure 3.6 shows a test written in YAML which verifies if the port 3333 of the target server is listening to *tcp6* traffic.

```
port:
  tcp6:3333:
    listening: true
```

Figure 3.6: Goss test which verifies if the port 3333 is listening to *tcp6* traffic

These solutions provide a fast and replicable way for developers to test the underlying software infrastructure of their systems, and some effort has been done in order to simplify their usage with Docker. For example, **Dgoss** [6] builds a Docker image, instantiates a container and runs Goss tests in that Docker container with a single command. **Dockerspec** [6] performs similar actions, but also allows the developer to perform some static analysis on the Dockerfile. Nevertheless, none of these tools provide a level of liveness above 2, since the developer must run them manually. Furthermore, these tools do not run within an IDE. The tests must also be previously written by the developer. As such, there is still some manual work that must be done by the developer in order to run these tests.

[6]Links to the mentioned tools can be found in Appendix A

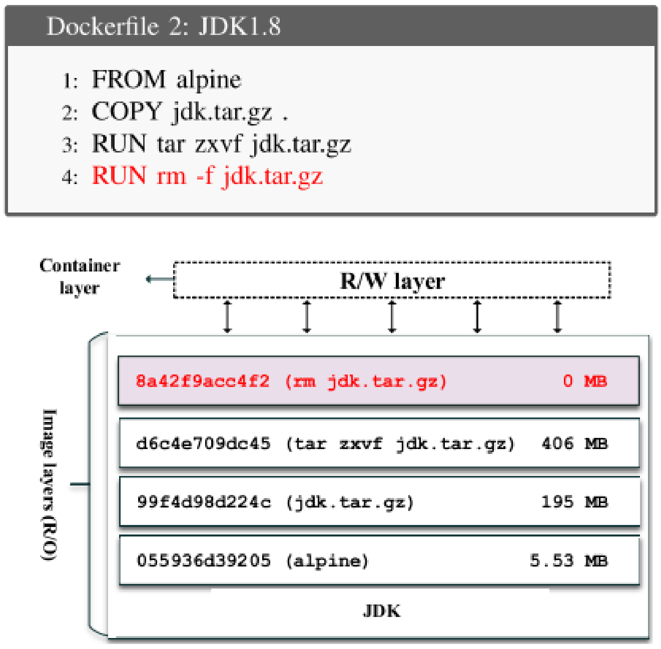
[7]Comparison between the resources supported by Chef Inspec and Serverspec available at `https://www.inspec.io/docs/reference/migration/`

3.1.5 Static Analysis

Static analysis tools, in contrast with dynamic analysis tools, are able to scan a Dockerfile for potential errors without instantiating a Docker container or building a Docker image. As such, while their scope is more restricted when compared to the other solutions, they can also be much faster to run.

J. Xu et al [39]. propose a static analysis approach to detect a code smell named Temporary File Smell (TF Smell). During the Docker image building process, some files are only used during one or more steps of the building process and are not necessary during runtime. As such, these files should be deleted during the building process, after being used, so that the final image size is kept as small as possible. Failing to delete such files is what J. Xu et al. coined as a TF Smell.

Figure 3.7 shows an example of a TF Smell. In this example, the developer copies a temporary file called `jdk.tar.gz` (instruction 2), uses the file (instruction 3), and deletes the file (instruction 4). However, given the layered nature of the Docker image building process, as described in Section 2.1.3, the 4th instruction does not remove the file from the image, since on the layer created by the 3rd instruction the file will still be present.



Dockerfile 2: JDK1.8

```
1: FROM alpine
2: COPY jdk.tar.gz .
3: RUN tar zxvf jdk.tar.gz
4: RUN rm -f jdk.tar.gz
```

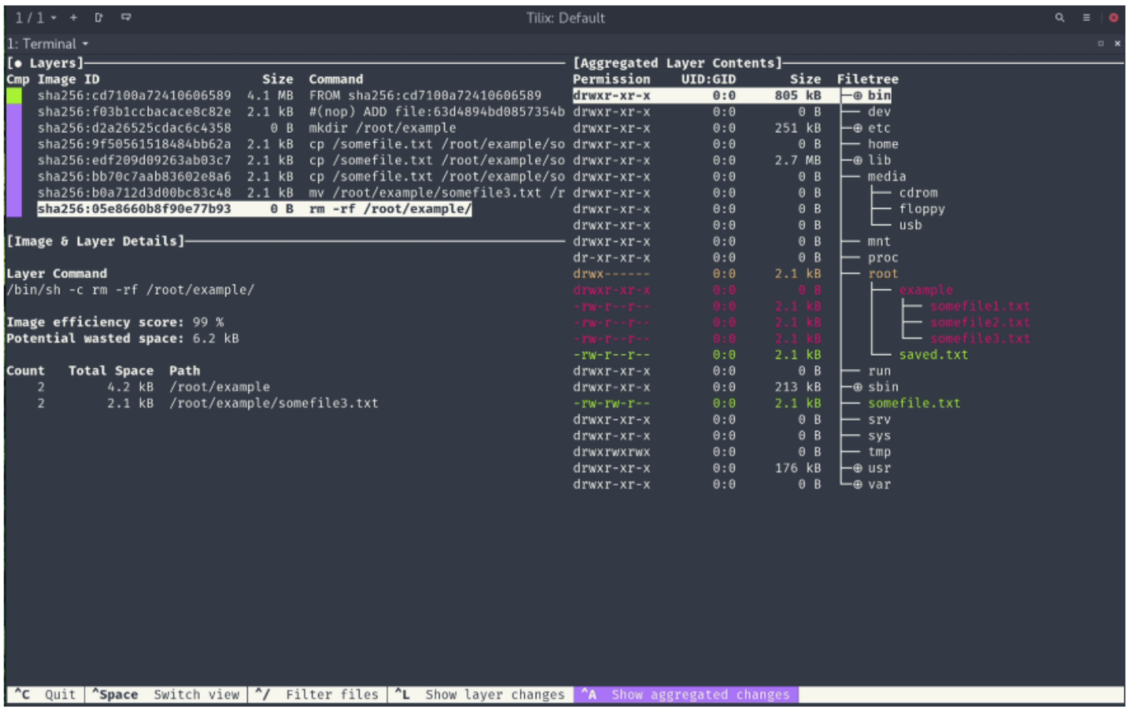
```
8a42f9acc4f2  (rm jdk.tar.gz)          0 MB
d6c4e709dc45  (tar zxvf jdk.tar.gz)  406 MB
99f4d98d224c  (jdk.tar.gz)           195 MB
055936d39205  (alpine)              5.53 MB
```

Figure 3.7: TF Smell caused by not removing a temporary file [39].

The static analysis method proposed by J. Xu et al. to detect TF Smells in a Dockerfile consists of three steps. On the first step, an Abstract Syntax Tree (AST) of the target Dockerfile is created. This AST is used in the following steps. On the second step, a semantic analyser is used in order to create a list containing every file that is created (e.g. using a *wget* command) and another list containing every file that is deleted (e.g. using a *rm* command). The line where each creation/deletion occurs is also stored. In the third step, created and deleted files are compared. A

TF Smell exists if this comparison reveals that a file is deleted in an instruction different from the one where it is created [39].

dive is a tool which allows a developer to navigate through the file system of each layer of a docker image [8]. Developers can see which files are added, removed or modified in each layer of the Dockerfile, allowing them to easily detect potentially unnecessary files still present in the image. The size and command of each layer is also presented to the user. Furthermore, in order to facilitate the exploratory work of the developer, filtering and agreggation of files are also supported.



```
[● Layers]──────────────────────────────────────────────────── [Aggregated Layer Contents]──────────────────────────
Cmp Image ID                    Size  Command                           Permission     UID:GID       Size  Filetree
    sha256:cd7100a72410606589  4.1 MB  FROM sha256:cd7100a72410606589    drwxr-xr-x         0:0     805 kB  ├─⊕ bin
    sha256:f03b1ccbacace8c82e  2.1 kB  #(nop) ADD file:63d4894bd0857354b drwxr-xr-x         0:0        0 B  ├── dev
    sha256:d2a26525cdac6c4358     0 B  mkdir /root/example               drwxr-xr-x         0:0     251 kB  ├─⊕ etc
    sha256:9f50561518484bb62a  2.1 kB  cp /somefile.txt /root/example/so drwxr-xr-x         0:0        0 B  ├── home
    sha256:edf209d09263ab03c7  2.1 kB  cp /somefile.txt /root/example/so drwxr-xr-x         0:0     2.7 MB  ├─⊕ lib
    sha256:bb70c7aab83602e8a6  2.1 kB  cp /somefile.txt /root/example/so drwxr-xr-x         0:0        0 B  ├── media
    sha256:b0a712d3d00bc83c48  2.1 kB  mv /root/example/somefile3.txt /r drwxr-xr-x         0:0        0 B  │   ├── cdrom
    sha256:05e8660b8f90e77b93     0 B  rm -rf /root/example/             drwxr-xr-x         0:0        0 B  │   ├── floppy
                                                                         drwxr-xr-x         0:0        0 B  │   └── usb
[Image & Layer Details]─────────────────────────────────────── drwxr-xr-x         0:0        0 B  ├── mnt
                                                                         dr-xr-xr-x         0:0        0 B  ├── proc
Layer Command                                                            drwx------         0:0     2.1 kB  ├── root
/bin/sh -c rm -rf /root/example/                                         drwxr-xr-x         0:0        0 B  │   ├── example
                                                                         -rw-r--r--         0:0     2.1 kB  │   │   ├── somefile1.txt
Image efficiency score: 99 %                                             -rw-r--r--         0:0     2.1 kB  │   │   ├── somefile2.txt
Potential wasted space: 6.2 kB                                           -rw-r--r--         0:0     2.1 kB  │   │   └── somefile3.txt
                                                                         -rw-r--r--         0:0     2.1 kB  │   └── saved.txt
Count   Total Space  Path                                                drwxr-xr-x         0:0        0 B  ├── run
    2        4.2 kB  /root/example                                       drwxr-xr-x         0:0     213 kB  ├─⊕ sbin
    2        2.1 kB  /root/example/somefile3.txt                         -rw-rw-r--         0:0     2.1 kB  ├── somefile.txt
                                                                         drwxr-xr-x         0:0        0 B  ├── srv
                                                                         drwxr-xr-x         0:0        0 B  ├── sys
                                                                         drwxrwxrwx         0:0        0 B  ├── tmp
                                                                         drwxr-xr-x         0:0     176 kB  ├─⊕ usr
                                                                         drwxr-xr-x         0:0        0 B  └─⊕ var

 ^C Quit │ ^Space Switch view │ ^/ Filter files │ ^L Show layer changes │ ^A Show aggregated changes
```

Figure 3.8: Dive tool presenting the file system of a layer and the image potential wasted space [a].

[a]Taken from dive's documentation listed in Appendix A.

Dive is also able to present to the user the files that might be potentially wasting space in the image (i.e. detect and localize TF Smells [39]). This is performed by automatically comparing every file across every layer and detecting any files which are deleted in a layer different from the one where the files are created. As an example, Figure 3.8 presents the output of the Dive tool for a Docker image, showing to the user the multiple layers of the image, the file system of the selected layer and the files which are potentially wasting space.

[8]Links to the mentioned tools can be found in Appendix A

Despite its usefulness, all the features mentioned require manual work from the developer, since the developer must still navigate and analyse the file system manually. Furthermore, this tool does not automatically update according to changes in the Dockerfile.

Hadolint [Standalone] is a tool which performs static analysis on a Dockerfile, presenting to the developer a list of possible errors and bad practices that the target Dockerfile may contain [9]. The rules which this linter applies [10] are based on the best practices list available in the official Docker documentation [11].

The analysis of the Dockerfile consists of three main steps. The first step is to generate an AST of the target Dockerfile which is used in the following steps. The second step is to use the AST in order to verify if any of the rules is being broken. The third step is to perform the linting of any Bash code that exists inside a `RUN` instruction. This last step is optional, since it only applies when a Dockerfile contains a `RUN` instruction.

Hadolint is a standalone tool which is not live nor integrated, by default, into an IDE.

Hadolint [VSCode plugin] is a plugin which integrates Hadolint [9] into Visual Studio Code [9]. This plugin automatically runs Hadolint on the Dockerfile that the developer is editing every time that the developer saves the Dockerfile, and presents the output of Hadolint in the user interface of the IDE [9]. The plugin presents the detected issues by underlining the respective instructions and allows the developer to see details on the issues by hovering over the instructions. It also lists all the issues in a window below the text editor.

This plugin, by automatically updating its state every time the user saves the Dockerfile, achieves the **3rd level of liveness**.

Docker for Visual Studio Code (Preview) and **Docker (JetBrains)**, previously mentioned in Section 3.1.1, also perform linting of the Dockerfile while it's being edited and present the detected issues in the development environment. Since these tools automatically update every time the Dockerfile is edited without requiring the developer to save changes, they achieves the **4th level of liveness** [9].

In sum, Table 3.4 presents all the tools analysed, as well as the level of liveness with which they provide static analysis. In this case, we can see that some tools are able to provide static analysis in the IDE with a high level of liveness — these tools update their static analysis feedback as the user edits the Dockerfile. However, this feedback can be quite limited when compared to specialized live programming solutions which perform dynamic analysis since only static analysis is performed.

[9] Links to the mentioned tools can be found in Appendix A

[10] Hadolint's list of rules available at `https://github.com/hadolint/hadolint#rules`

[11] Docker Docs — Best practices for writing Dockerfiles available at `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`

Table 3.4: Comparison of analysed Static Analysis tools. Links to the mentioned tools can be found in Appendix A.

Tool	**Static Analysis Liveness Level**	**IDE Plugin**
J. Xu et al.	2	No
Dive	2	No
Hadolint [Standalone] Dockerlint Dockerfile_lint Dockerfilelint	2	No
Hadolint [Plugin] Dockerfilelint [Plugin]	3	VS Code
Docker for Visual Studio Code (Preview)	4	VS Code
Docker	4	Intellij IDEA

3.1.6 Image Build Optimization

Docker images can become large and take a long time to build [38, 15, 14]. Given the iterative nature of the Dockerfile development process, some solutions attempt to improve the development process by reducing the time required to build a Docker image.

The image building time is also a critical factor to the success of live programming environments, since feedback is delivered in a rapidly changing environment. If the feedback takes too long to be generated and displayed to the developer, it can become outdated and, therefore, less useful [29]. As such, when attempting to create a live programming environment, it could be useful to implement solutions that can help lower the often excessive image compilation times.

FastBuild is a solution which implements a caching system for remotely accessed files during the Docker image building process. Using this cache system reduces the need to retrieve files from remote systems, resulting in a considerable reduction of the time required to build a Docker image, particularly in systems connected to a network with a low network bandwidth (up to 10.6x faster) [15].

FastBuild achieves this by implementing three main components. The first component is a caching mechanism which extends the existing caching mechanism in Docker. This mechanism intercepts every network request performed during the image building process and if the network request has been performed in a previous build then, instead of performing a new network request, it satisfies the request using cached data. It also tests if the cached data is out of date, by performing a request for the modification time. If the file is not stored in cache or the cached data is out of date, then the network request is performed as usual. The second component is responsible for overlapping instructions execution. The third component is responsible for building base images locally instead of pulling them from Docker Hub. This is possible and faster than downloading the base images due to the caching system implemented [15].

Figure 3.9 shows that this tool can greatly reduce the required time to build and launch a Docker container, and as such can reduce the amount of time that a developer has to wait for a Dockerfile to build during the development process.

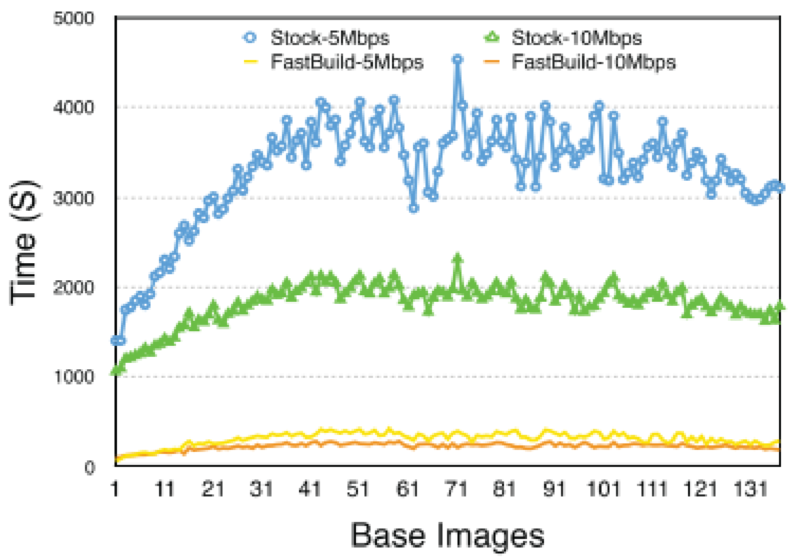


Figure 3.9: Build and launch time comparison on a low network bandwidth environment between FastBuild and the Docker default building technique for 137 sample base images with more than 100,000 downloads on the Docker Hub platform [15].

Slacker [14] is a replacement for the default Docker storage driver which aims at reducing the time required to pull and start a container without significantly decreasing the performance during the execution of the container. Tests revealed that the two steps which usually take up most of the time during the process of building and starting a container are (1) pulling an image hosted on a remote repository (such as DockerHub) and (2) file system operations, particularly when writing to a file which is on a deep layer of the AUFS file system. It was also noted that usually most of the data on a container is not read immediately at startup. According to tests performed on 57 Docker images from DockerHub, the median size of an image is 329MB. However, the median size of the files read until the container is running (i.e. server listening, printing a "system ready" message, etc.) is only 20MB. As such, Slacker was designed with the aim of reducing the time spent in these two steps.

Slacker consists of a centralized Network File System (NFS) which all the Docker daemons and registries have access to, where containers are stored as files and images are stored as snapshots. When a developer on his machine starts a container, the container files are actually stored in the NFS and all file system operations are performed over the NFS. This allows a container to lazily access the image data, reading and writing the data only as needed. Since images are stored as read-only snapshots and the containers data is located in the NFS, pulling an image no longer requires a network transfer to the developers machine, consisting of an efficient clone action on the NFS server which creates a clone of the read-only snapshot. With Slacker, image layers are flattened, however a copy-on-write (COW) system is implemented in the snapshot/clone system, which means that it still has the advantage of not requiring full copies of images every time an image is pulled [14]. Figure 3.10 provides a visual representation of this architecture.

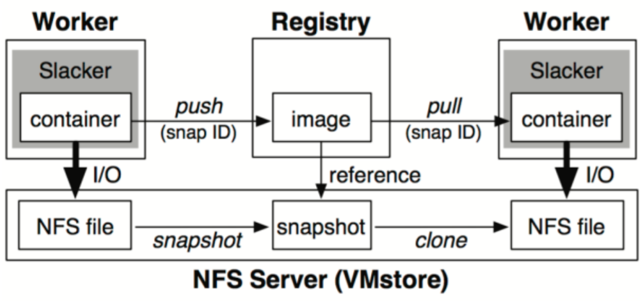


Figure 3.10: Slacker Architecture [14].

According to further testing, with Slacker, pulling an image becomes 72x faster than the regular image pulling with AUFS. The run phase, however, becomes 17% slower. Taking into consideration the median times that each phase of the development cycle usually takes, this provides significant improvements, reducing the median time on the push, pull and run cycle by 20x [14].

3.1.7 Discussion

Over the years, multiple efforts have been led with the purpose of improving the development process of Dockerfiles, allowing developers to have faster and easier access to the functionalities and metrics that are typically required to develop Dockerfiles. However, the vast majority of these tools do not provide any automatic feedback while the developer works and, as such, are not *live* enough to create a tight feedback loop in the developer's environment. To the best of our knowledge, the only tools which provide a level of liveness above 2 are some of the static analysis tools described in Section 3.1.5, which only give a very shallow level of information to the developer compared to the tools in other categories, as they only inspect the Dockerfile itself and not the image or the container.

Liveness in IDEs requires a short feedback loop, as the developer's changes to the code should be reflected on his environment shortly after being made [29]. Since Dockerfile builds can take some time, latency problems may arise when trying to increase the liveness in tools which require the Docker image to be built. Even when no liveness is involved, this can be detrimental to the developer's workflow [15]. As a way to mitigate this issue, there are some potentially viable solutions which focus on reducing the time required to build a Docker image. The authors of the solutions described in this section agree that network requests are one of the biggest culprits of long image compilation times. As such, these solutions rely mainly on caching mechanisms, which save time and network bandwidth and have the potential to be useful in live programming environments aimed at Dockerfiles.

3.2 Liveness and Feedback in IDEs

This section presents and analyses multiple existent IDEs or plugins/extensions to IDEs which focus on providing visual feedback during the development process. The search performed to find these solutions was not restricted to a specific activity within software development or liveness level. In order to come to these results, a search on IEEEXplore and the ACM Digital Library was performed using the query "Live Programming OR Liveness" and the query "integrated development environment OR IDE". Visual Studio Code's marketplace was also searched using the query "docker". Results were filtered by name, abstract, number of downloads (when applicable) and citation count (when applicable). The main purpose of this analysis is to understand different approaches to the implementation of liveness in IDEs and to the delivery of feedback to the developer. For that reason, IDEs with different liveness levels and feedback mechanisms were selected.

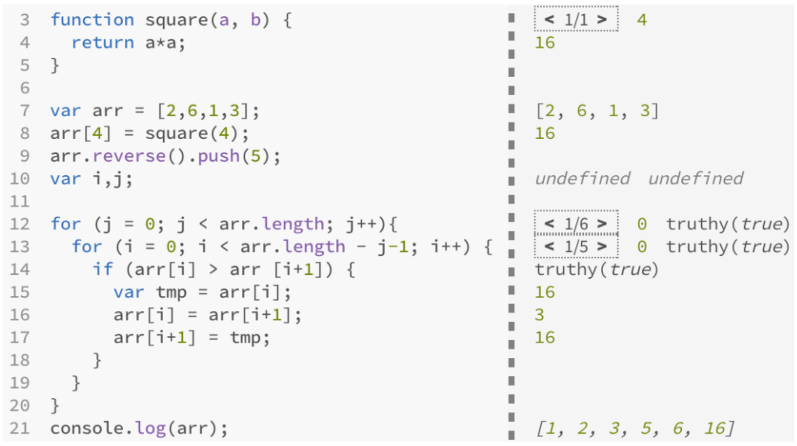
IDEs usually provide feedback about the program that is being developed. This feedback can consist of errors, warnings, suggestions and other information which might be useful for the developer to have during the development process. Live IDEs differ from non-live IDEs by providing this kind of information in a continuous way [37]. As such, it's important to understand how existing solutions tackle the issue of implementing liveness and present information in a way that is useful and comfortable to the developer. As such, an analysis has been perform in order to answer the questions:

- **What are the approaches currently used to implement liveness and display feedback in IDEs?**
- **What are the main threats to the success of liveness in IDEs?**

J. Kramer et al [22]. performed a controlled experiment with users in order to analyse the developers' behaviour when working on a live programming environment. Namely, they intended to demonstrate that using a live environment lowers the time spent between introducing a bug and fixing it and the overall time necessary to complete a task, and the participants would find and fix bugs throughout the development of the task instead of just at its end.

In order to validate these hypotheses, a prototype was developed which provided information about the code continuously, updating as the developer wrote the code. This prototype consisted of a plugin for the Brackets IDE. It consists of two main components, the back-end and the front-end. The front-end sends the JavaScript code that the developer is writing as a *string* to the back-end. The back-end then executes the developer's code in a sand-boxed environment, gathering information about the code while it runs. This information consists of variable values, log messages, function call return values and parameters and also supports the inspection of different loop iterations [22]. As an example, Figure 3.11 shows the output of the tool for a snippet of JavaScript code, showing the multiple types of possible feedback displayed to the user.

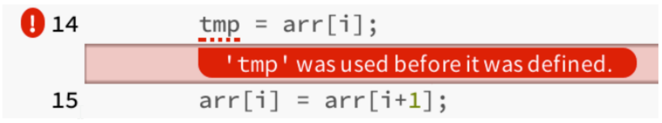
As shown in Figure 3.11, the information is displayed at the right side of the corresponding line. This tool also provides some feedback when an error is detected on the developer's code.



```
function square(a, b) {                              < 1/1 >  4
  return a*a;                                        16
}

var arr = [2,6,1,3];                                 [2, 6, 1, 3]
arr[4] = square(4);                                  16
arr.reverse().push(5);
var i,j;                                             undefined  undefined

for (j = 0; j < arr.length; j++){                    < 1/6 >  0  truthy(true)
  for (i = 0; i < arr.length - j-1; i++) {           < 1/5 >  0  truthy(true)
    if (arr[i] > arr [i+1]) {                        truthy(true)
      var tmp = arr[i];                              16
      arr[i] = arr[i+1];                             3
      arr[i+1] = tmp;                                16
    }
  }
}
console.log(arr);                                    [1, 2, 3, 5, 6, 16]
```

Figure 3.11: Example output for Javascript code on Kramer's environment [22].

Figure 3.12 shows an example of the interface presented when an error is found in the developer's code. In this tool, a small red error icon is placed at the left of the line where the error is detected and the faulty part of the code is underlined with a dotted red line. When the user clicks this icon, a small red window is displayed below the faulty line which contains a short description of the error that was detected. This tool does not perform any attempt to automatically correct or suggest potential fixes for the errors found in the code [22].



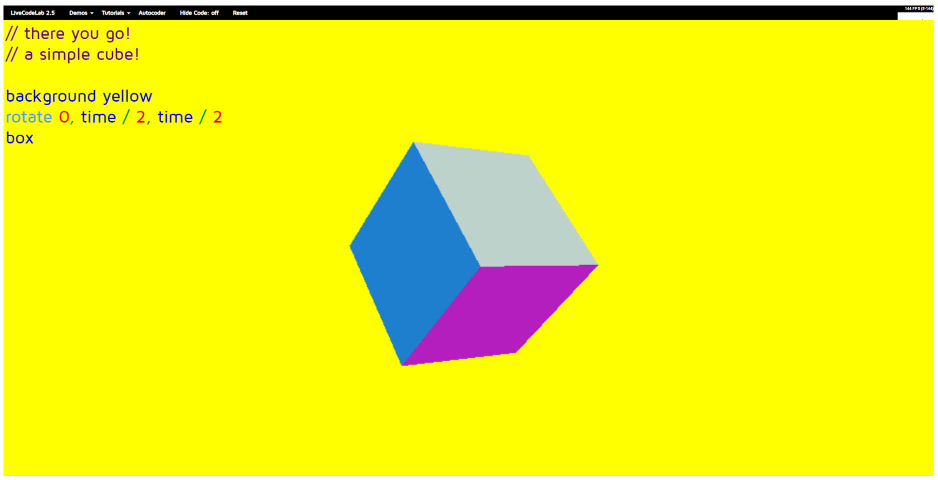
```
      tmp = arr[i];
      'tmp' was used before it was defined.
      arr[i] = arr[i+1];
```

Figure 3.12: Javascript error detected in user's code on Kramer's environment [22].

Since the feedback is updated continuously and the developer does not need to perform any explicit or implicit action in order to trigger the update, this prototype provides the 4th level of liveness. Furthermore the live feedback provided results from both dynamic and static analysis, since some feedback is generated from the execution of the program and some feedback is generated from the raw source code [22].

The results from the user study for this prototype showed that using a live coding environment significantly lowered the total time spent fixing bugs and promoted testing throughout the course of the development. However, there was no decrease in the total time required for the completion of the task. J. Kramer et al. believe that this can be due to a small sample size. Overall these results show that this implementation of liveness can have some benefits on the improvement of the development process [22].

LiveCodeLab [10] is a web-based framework that allows developers to write code in the LiveCodeLang language and visualize the result of the code in real-time. The environment is tailored towards the development of 3d graphics and audio sample sequencing, providing a live visualization and playback of the code that is being written. The user does not need to perform any action in order to update the output of the tool, meaning that the tool is updated every time that the code changes into a different valid state. Figure 3.13 presents the interface of LiveCodeLab, executing the sample code provided in the website called "Simple cube".



```
// there you go!
// a simple cube!

background yellow
rotate 0, time / 2, time / 2
box
```

Figure 3.13: LiveCodeLab interface with the example code "Simple cube" being executed.

Initially, LiveCodeLab used JavaScript as its language. Since the accessibility and intuitiveness were core principles of the project, the next iterations of the framework moved to the LiveCodeLang language, which is based in CoffeeScript and was designed with accessibility in mind, having short syntax and intuitive keywords which facilitate the exploration process of the developer without compromising the language's flexibility and potential [10].

Internally, LiveCodeLab performs 4 different actions independently and at different intervals [10]:

- When the developer edits the code via keyboard or mouse, the code is translated, parsed and hot-swapped over the old code.
- Since the developer can write code which depends on the current timestamp, the graphics rendering and other data structures are updated at regular time intervals. This happens up to 60 times per second.
- The playback of audio samples is handled by a component which runs at a specific interval. This interval is specified in "beats per minute" and is customizable using an instruction in the code.
- LiveCodeLab also includes an optional feature called AutoCoder which randomly swapping parts of the code every second. This feature, despite being undesired during critical development, can be useful for a novice developer to explore the possibilities of the tool at a faster pace.

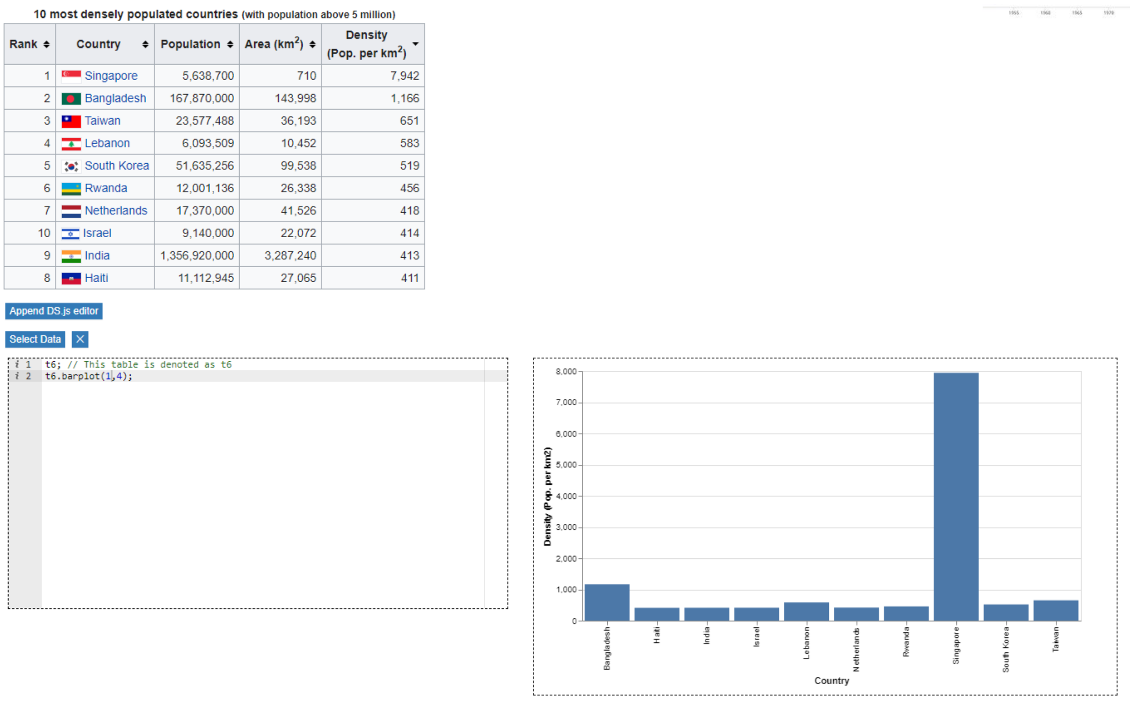
In order to ensure that the environment does not crash or stop its output at any time due to the developer writing faulty code, 2 verifications are made. First, the developer's code is syntactically analysed, and if a syntax error is found, the code is not updated and the last working state keeps running. In order to prevent runtime errors from affecting the flow of the environment, the previous known working state of the environment is kept for the first few seconds of execution of a new code. If that code runs for long enough, then it is considered stable and becomes stored as the last working state. If that code crashes within a certain time frame, then it is discarded and the last working state of the environment is restored. This does not guarantee that no runtime errors will occur, but relies on the assumption that most runtime errors will occur within the first few seconds of execution [10].

Since the feedback is updated continuously and the developer does not need to perform any explicit or implicit action in order to trigger the update, LiveCodeLab provides the 4th level of liveness. Furthermore the live feedback provided results from both dynamic and static analysis, since some feedback is generated from the execution of the program and some feedback is generated from the raw source code [10].

DS.js [41] is a tool which enables a live data science programming environment in any web-page which contains data in the form of HTML tables or CSV/TSV datasets. DS.js detects the data in the webpage and injects itself into the page near the tables, giving to a developer the option to manipulate the data, derive tables from the data and perform other data science tasks. The interface provided to the user is embedded directly into the web-page, consisting of a box where the user writes JavaScript code which manipulates the data and a box where the result of the data manipulation (i.e. a new table, a graph or other visualization) is displayed. Figure 3.14 shows an example of this interface, where data is extracted from an HTML table of a website and a bar plot is drawn using the data present in that table. DS.js is also able to suggest certain actions to the user and to save the work that a user has done in a website. These functionalities help the developer explore and discover more functionalities from the provided data manipulation API.

DS.js updates it's live visual output pane every time that the developer finishes writing a line of code, written in the editor pane. The end of a line of code is detected by the insertion of a newline or a semicolon. The live visual output only renders the output of a single line of code at a time. As such, developers need to be able to switch between the lines to be able to choose at any moment which one is being displayed. Line switching is done by placing the cursor on the line which is meant to be rendered in the live pane. Despite only the output of a single line being displayed at a time, the whole block of code is executed. This means that DS.js provides the 3rd level of liveness, since some user action is required in order to update the live visual output pane [41].

Other mechanisms are used in order to facilitate the work of the developer. For instance, static analysis is performed and potentially problematic statements related to the declaration and assignment of global variables are highlighted to the developer by displaying a small warning icon next to the respective statements. Developers also have access to previews for individual method calls. As illustrated in Figure 3.15 this is particularly useful in statements where many method calls are chained, since it allows the developer to evaluate each one of these method calls individually, therefore potentially making it easier to determine which of the chained method calls



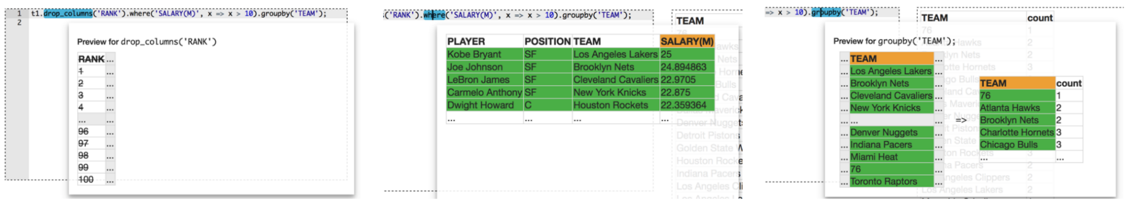
10 most densely populated countries (with population above 5 million)

Rank	Country	Population	Area (km²)	Density (Pop. per km²)
1	Singapore	5,638,700	710	7,942
2	Bangladesh	167,870,000	143,998	1,166
3	Taiwan	23,577,488	36,193	651
4	Lebanon	6,093,509	10,452	583
5	South Korea	51,635,256	99,538	519
6	Rwanda	12,001,136	26,338	456
7	Netherlands	17,370,000	41,526	418
10	Israel	9,140,000	22,072	414
9	India	1,356,920,000	3,287,240	413
8	Haiti	11,112,945	27,065	411

```
t6; // This table is denoted as t6
t6.barplot(1,4);
```

Figure 3.14: DS.js interface drawing a bar plot with data extracted from a html table.

is behaving unexpectedly in case of a bug. This mechanism also provides the 3rd level of liveness, since the previews are live but require a mouse click on the desired method call in order to update and be displayed. Both these mechanisms have in common the fact that they help the developer by providing more feedback.



```
t1.drop_columns('RANK').where('SALARY(M)', x => x > 10).groupby('TEAM');
```

Preview for drop_columns('RANK')

~~RANK~~	...
~~1~~	...
~~2~~	...
~~3~~	...
~~4~~	...
...	...
~~96~~	...
~~97~~	...
~~98~~	...
~~99~~	...
~~100~~	...

(a) previews the column to be dropped

PLAYER	POSITION	TEAM	SALARY(M)
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.894863
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.359364
...	...	...	...

(b) previews the selected rows

Preview for groupby('TEAM');

...	TEAM	...
...	Los Angeles Lakers	...
...	Brooklyn Nets	...
...	Cleveland Cavaliers	...
...	New York Knicks	...
...	...	...
...	Denver Nuggets	...
...	Indiana Pacers	...
...	Miami Heat	...
...	76	...
...	Toronto Raptors	...

=>

TEAM	count
76	1
Atlanta Hawks	2
Brooklyn Nets	2
Charlotte Hornets	3
Chicago Bulls	3
...	...

(c) previews the grouping operation

Figure 3.15: DS.js preview panes on three chained method calls [41].

User studies revealed that developers were able to successfully complete data science tasks in 30 minutes. Subjects had some data analysis experience, but had almost no introduction to DS.js beforehand. The overall response from the subjects to the live development experience was positive and they liked its responsiveness, although they wished that it was possible to keep more than one visualization on the screen at a time. Additionally, despite the benefits of having a live environment, all subjects agreed that DS.js would be less suited for more complex analysis tasks, since the API for data manipulation provided by the tool is too simple for those tasks [41].

Omnicode [20] is a live IDE which runs the developer's Python code and, using *scatterplots*, displays all the values that the program's variables had, as a function of the execution step or another variable. Figure 3.16 shows two *scatterplots*. The left one shows the values of the variable `temp` as a function of the number of execution steps, while the second one shows the values of the variable `temp` as a function of the value of `sum(nums)` (i.e. the sum of all the variables inside the list `nums`).

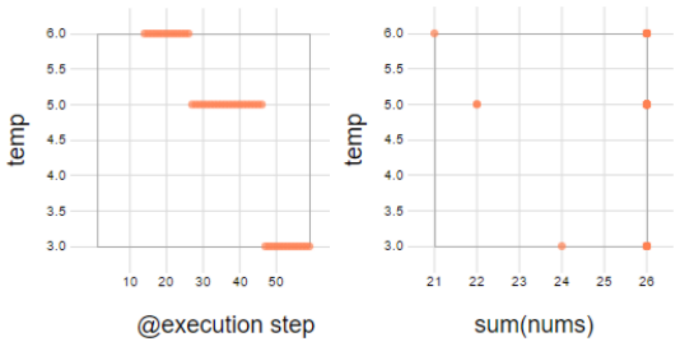


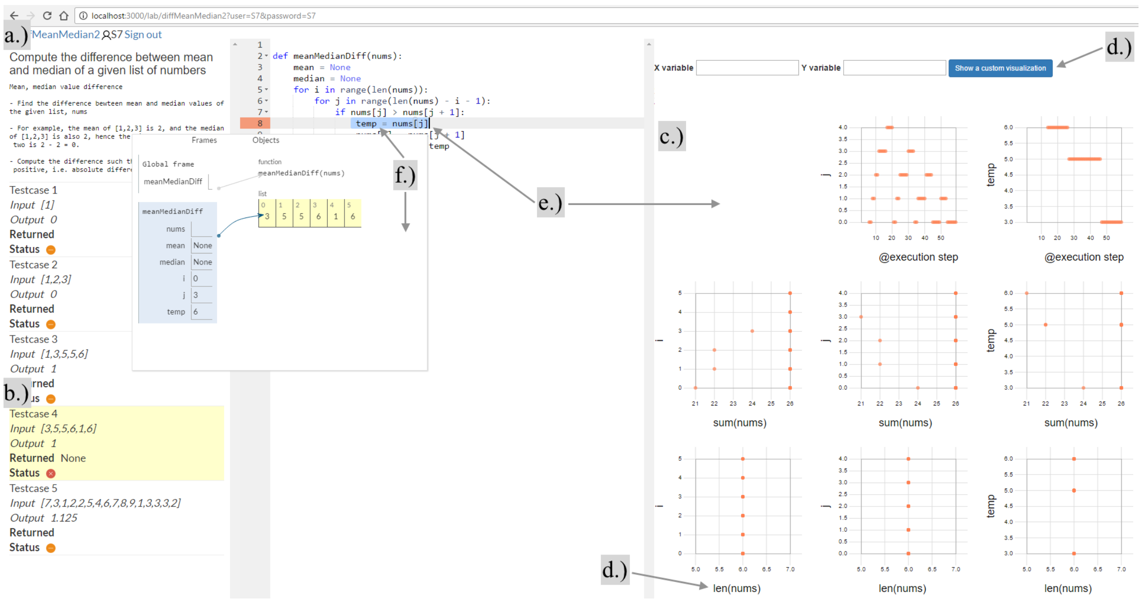
Figure 3.16: *Scatterplots* which display the values of runtime variables [20].

Omnicode was specially designed for environments where the developer has a set of test cases consisting of known inputs and respective expected outputs and is tasked with writing the code that processes the inputs and generates the expected outputs. The developer can see the test cases and select one of them at a time. After selecting a test case, Omnicode automatically runs the code while it's being written by the developer, feeding it with the input from the selected test case, and providing feedback to the developer. Tests are run when the developer stops writing for 2 seconds and the code has a valid syntax, which means that this tool implements a form of live programming which can be defined as *continuous testing* with the 3th level of liveness [20].

The feedback provided in real-time consists of:

- The return value of the code.
- A color indicator which shows if that value matches the expected output from the selected test case.
- *Scatterplots* which display the values of runtime values during execution.
- By selecting a line of code, a popup displays a visualization of all the data structures existent when the selected line is executed.

The user can also add custom *scatterplots* which allow him to visualize the value of a runtime value as a function of another runtime value. This volume of feedback being provided in real-time to the developer is what makes Omnicode stand out from other solutions, as it can be considered an extreme implementation of live programming. In order to reduce clutter and provide more meaningful information, the developer is also able to select a few lines of code or sections of a *scatterplot* and the whole IDE updates to display just the information relative to the selected execution segment. This selection works bidirectionally, since by selecting some lines of code only the corresponding sections of the *scatterplots* are highlighted, and vice-versa [20]. Figure 3.17 shows the interface with the main points of interested marked with letters *a.)* to *f.)*.



Compute the difference between mean and median of a given list of numbers

```
Mean, median value difference

- Find the difference bewteen mean and median values of
the given list, nums

- For example, the mean of [1,2,3] is 2, and the median
of [1,2,3] is also 2, hence the
 two is 2 - 2 = 0.

- Compute the difference such th
 positive, i.e. absolute differe
```

Testcase 1
Input [1]
Output 0
Returned
Status

Testcase 2
Input [1,2,3]
Output 0
Returned
Status

Testcase 3
Input [1,3,5,5,6]
Output 1
rned
us

Testcase 4
Input [3,5,5,6,1,6]
Output 1
Returned None
Status

Testcase 5
Input [7,3,1,2,2,5,4,6,7,8,9,1,3,3,3,2]
Output 1.125
Returned
Status

```

def meanMedianDiff(nums):
    mean = None
    median = None
    for i in range(len(nums)):
        for j in range(len(nums) - i - 1):
            if nums[j] > nums[j + 1]:
                temp = nums[j]
```

Figure 3.17: Omnicode example interface [20].

After an exploratory user study, subjects reported that the live visualizations helped debugging the code and validate their mental models of how the program works, reducing the need to write print statements in order to do so, and that having a rich visualization helped them explain their code to other people. However, subjects also reported that visual overload was a problem, which is to be expected giving that one of the core objectives of Omnicode was to push live programming to the extreme by displaying every value of every variable as a function of other variables at the same time [20].

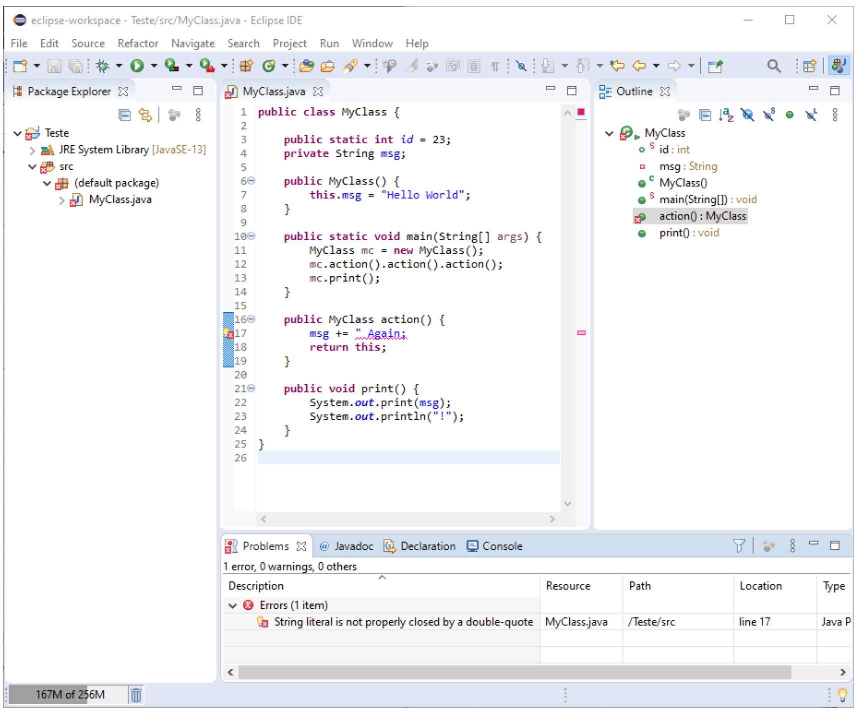
As a way to reduce the visual overload, the authors highlights a few possibilities [20]:

- Merge redundant information.
- Use color to help differentiate information.
- Allow user to selectively hide, show, resize and move information.
- Analyse the information and automatically highlight the most important segments.

By implementing these possibilities, the system would still be able to present the same information, but would do so in a way that would be less overwhelming to the developer. It should also be noted that despite the negative feedback regarding the visual overload, users were still generally satisfied with this particular section of the tool, rating it at an average of 4.11 out of 5 [20].

Omnicode's *scatterplots* as well as the automated test results and any syntax and runtime errors detected in the code via static analysis are displayed in panes located on both sides of the code editor. These elements are always visible. By hovering the cursor over a line, a temporary pane displays execution information for that line. Omnicode makes no attempt to automatically correct or suggest potential fixes for the errors found in the code [20].

Eclipse IDE [28] is an integrated development environment which was a popular choice for developers working in the Java programming language in 2006 and still is fairly common nowadays [12]. It provides support for the Java Development Tools (JDT) and a solid platform for third-party plug-ins, allowing for the extensive customization of the functionalities available during the development process. Despite supporting other languages, the analysis of this tool focuses on the feedback provided to the developer when writing Java code with the default interface configuration.



```java
public class MyClass {

    public static int id = 23;
    private String msg;

    public MyClass() {
        this.msg = "Hello World";
    }

    public static void main(String[] args) {
        MyClass mc = new MyClass();
        mc.action().action().action();
        mc.print();
    }

    public MyClass action() {
        msg += " Again;
        return this;
    }

    public void print() {
        System.out.print(msg);
        System.out.println("!");
    }
}
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
String literal is not properly closed by a double-quote	MyClass.java	/Teste/src	line 17	Java P

Figure 3.18: Default Eclipse IDE interface for Java

Figure 3.18 displays the main interface that this analysis focuses on. The left pane contains the Package Explorer tab which displays the resources available in the project, such as files and dependencies, organized in packages. The center pane is the code editor, where the developer can read and write code. The code editor also displays any problems detected in the code, displaying a small icon near the line count and underlining the relevant code in the respective line. By hovering over the underlined portion of code, a short description of the problem is displayed. For some of

[12]Stack Overlow Developer Survey 2019 available at https://insights.stackoverflow.com/survey/2019

the detected problems, clicking on the icon displays a list of suggested changes in order to fix the problem. The right pane contains the Outline tab which presents an overview of the file currently visible in the code editor. This overview consists of a list of the classes present in the file, as well as the instance variables, methods and constructors of each class. Each element is labeled with a representative icon. This tab is updated every time that the developer stops writing code for a few seconds. The bottom pane contains the Problems, Javadoc, Declaration and Console tabs. The Problems tab provides a textual description of any problems detected in the developer's code, as well as the resource, path, location and type of each problem. This tab is updated every time the user saves the current file. The Console tab displays the output of the developer's program, updating every time the developer manually triggers an execution of the program. The Javadoc tab displays documentation relative to the currently opened file. The Declaration tab displays information relative to the code where the cursor is currently placed [28].

Eclipse does not perform any live dynamic analysis. However, syntax errors are detected and highlighted continuously, without requiring any explicit request from the developer. As such, Eclipse provides some feedback obtained via static analysis with the 4th level of liveness.

Code Bubbles [7] is an integrated development environment for the Java programming language which implements a user interface which is organized in sets of bubbles. Each individual bubble is an independent element which can contains a piece of code (such as a method, constructor or a class's variables), documentation, search results, developer notes or other artifacts. Bubbles can be grouped by dragging a bubble to a group or a bubble to another bubble. This allows the developer to organize his IDE in a customizable way. Furthermore, a developer can navigate through the code by right clicking a segment of code and pressing the "Open Declaration" or "Find All References" buttons and by searching through the source code in the right pane of the interface. Each of these interactions opens a new bubble that contains just the portion of code that the developer asked for, such as a class or a function. This approach, as opposed to the more common file-based approach, allows the developer to navigate through the code in a working set-based IDE which follows more naturally the code's structure. In addition, by hovering over certain sections of code, a temporary bubble shows the developer a preview of their containing method. The developer can also assign titles and icons to bubbles or bubble sets. Figure 3.19 displays the aforementioned features and the interface of Code Bubbles.

Code Bubbles' user interface is implemented using Microsoft Windows Presentation Foundation, while it's back-end runs as a plugin for the Eclipse IDE, using some of Eclipse's features in order to process the code. Features such as build, run and debug options, finding references and declarations, error and warning management and syntax highlighting are provided by Eclipse and displayed in Code Bubble's interface. As such, the feedback provided by Code Bubbles when an error is detected in the code is similar to the feedback provided by Eclipse in the code editor [7], meaning that Code Bubbles also provides some feedback obtained via static analysis with the 4th level of liveness.

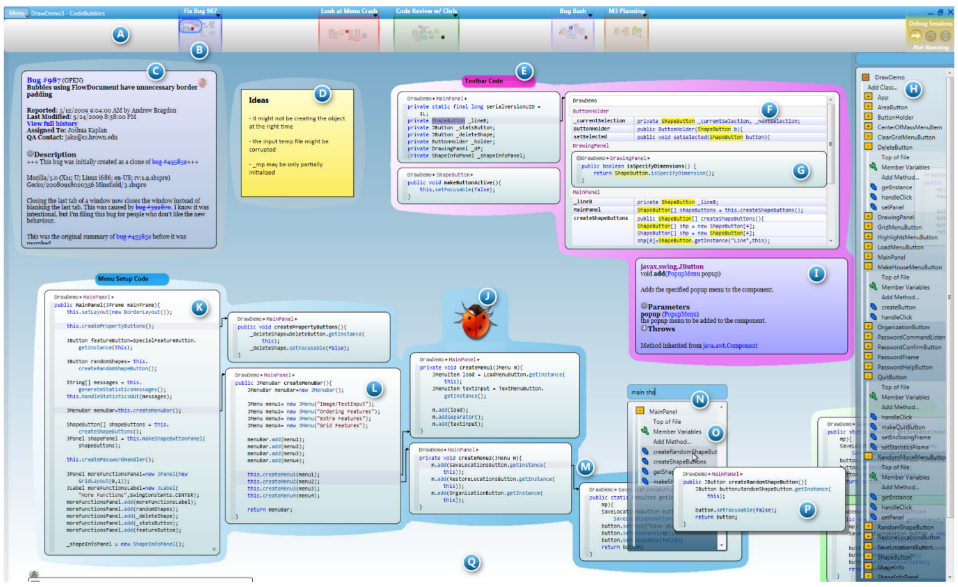


Figure 3.19: Code Bubbles IDE interface. Each letter represents one of the main elements of the user interface [7].

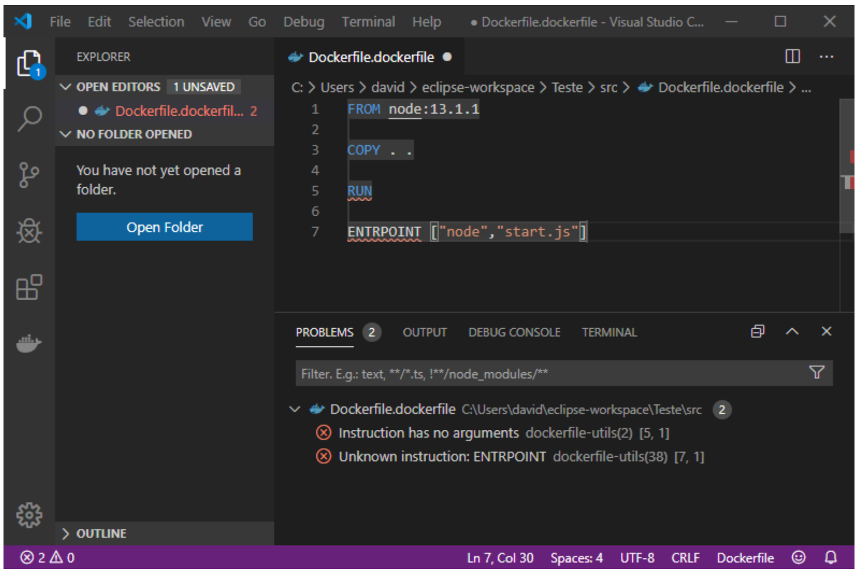
User studies were conducted in order to understand how beneficial is the Code Bubbles code visualization technique and also to gauge the opinion of experienced professional developers about the features implemented. Overall, the participants felt positive about the environment. They valued the ability of being able to see multiple functions side-by-side even if they were in different files and generally agreed that reviewing code in Code Bubbles was easier than their preferred IDE [7].

Visual Studio Code [13] is a code editor which can be used for many languages and purposes, supporting a wide range of languages out-of-the-box and providing a plug-in marketplace where developers can distribute and download plug-ins which further expand the capabilities of the environment. This analysis focuses on the development of Dockerfiles in Visual Studio Code with the Docker plugin [13] installed.

The feedback provided by this plugin regarding the file opened in the code editor consists of syntax highlighting, problem detection and documentation. Problems detected in the currently opened Dockerfile are displayed in a few different ways: the segments of code where each problem was detected are underlined, a red mark indicating the location of the problems in the file is displayed in the scroll bar and a textual description of each detected problem is displayed in a pane below the code editor, providing also a count of all the detected problems. Furthermore, in the file browser, if there are any problems detected then the file's name is displayed in red. By hovering over any underlined segment of code, a small window also displays a textual description of the respective problem. Documentation can be accessed by hovering over valid Dockerfile instructions. This action triggers the display of a small window containing a short description of

[13]Links to the mentioned tools can be found in Appendix A

the hovered instruction and simple examples of its usage, as well as a link to a web-page where the developer can obtain more extensive documentation for that instruction. Figure 3.20 displays the interface of Visual Studio Code with the Docker plug-in where some of the described features can be observed.


File
Edit
Selection
View
Go
Debug
Terminal
Help
● Dockerfile.dockerfile - Visual Studio C...
EXPLORER
OPEN EDITORS 1 UNSAVED
● Dockerfile.dockerfil... 2
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
OUTLINE
Dockerfile.dockerfile ●
C: > Users > david > eclipse-workspace > Teste > src > Dockerfile.dockerfile > ...
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter. E.g.: text, **/*.ts, !**/node_modules/**
⊗ 2 ⚠ 0
Ln 7, Col 30
Spaces: 4
UTF-8
CRLF
Dockerfile

```
FROM node:13.1.1

COPY . .

RUN

ENTRPOINT ["node","start.js"]
```

```
Dockerfile.dockerfile C:\Users\david\eclipse-workspace\Teste\src 2
  ⊗ Instruction has no arguments dockerfile-utils(2) [5, 1]
  ⊗ Unknown instruction: ENTRPOINT dockerfile-utils(38) [7, 1]
```

Figure 3.20: Visual Studio Code with Docker plug-in interface.

All the feedback described is obtained via static analysis of the Dockerfile and is instantly updated on every code change, therefore providing static analysis feedback with the 4th level of liveness.

3.2.1 Discussion

This analysis presents a few different approaches to live feedback and shows that the presence of liveness in the development environment can have some benefits, such as leading to a reduction of the average time required for a developer to create a working program or to fix existing bugs [22]. Furthermore, the increased accessibility to relevant information in the IDE can be valuable to developers, providing easier access to information which otherwise would only be reachable by manually retrieving it. Increasing the understandability of the code can also be one of the benefits of providing live feedback to the developer. It's also noticeable that all of the analysed environments provide live feedback generated from static analysis, while only some perform live dynamic analysis. Environments which provide live feedback from dynamic analysis usually display a larger amount of valuable information to the developer [20] and promote a more continuous workflow where the developer is able to test his code more often [22] without having to leave the IDE.

It can also be noticed that while some tools focus on providing feedback based on the plain execution of the developer's code, Omnicode adopts a test-based strategy where the user code is automatically tested against pairs of known inputs and respective expected outputs [20].

Table 3.5: Comparison of analysed development environments. The *Quick fix* column refers to the ability to suggest code changes to the developer.

Environment	**Target language**	**Navigation**	**Quick fix**	**Docs access**	**Static analysis liveness level**	**Dynamic analysis liveness level**
Omnicode	Python	File-based and test-based	No	No	4	4
LiveCodeLab	LiveCodeLang	Single-file	No	No	4	4
J. Kramer et al.	Javascript	File-based	No	No	4	4
DS.js	Javascript (with Data Mining API)	Table-based	No	Yes	3	3
Eclipse IDE	Java	File-based	Yes	Yes	4	2
Code Bubbles IDE	Java	Working set-based	Yes	Yes	4	2
Visual Studio Code with Docker plug-in	Dockerfile	File-based	No	Yes	4	2

Furthermore, although the majority of IDEs adopt a file-based approach, other approaches such as the working set-based [7] and table-based [41] are also evaluated positively by developers for a few reasons, such as fitting the code's logic more tightly [7].

In regards to the visibility of the feedback provided, all the analysed environments provide partial visibility, where a lighter version of the feedback information is always displayed to the developer and the full textual feedback generated by the IDE is visible only in certain views or by performing certain actions, such as hovering over code. Furthermore, feedback that is always displayed to the developer usually consists of small, unobtrusive and easily identifiable elements, such as small icons placed near the line where a problem is detected or a red underline. Allowing the developer to toggle the visibility of certain fragments of information is a way to ensure that the user interface is not overwhelming while still providing all the desired feedback. Live programming environments such as Omnicode and the environment proposed by J. Kramer et al. are exceptions, constantly displaying a large amount of information to the developers. For that reason, these tools can be overwhelming at times, although there are some mechanisms that can help mitigate this issue [20] [22].

It should also be noted that features such as generated code fix suggestions and quick access to the documentation have mixed support on the analysed environments. Both these features can be helpful to the developer, particularly in cases where the developer is working with libraries beyond the standard libraries of a programming language, as seen in DS.js [41]. Once again, the design of these features is very important, so that the developer has a fluid interaction with his environment and does not feel overwhelmed with information.

Table 3.5 presents a direct comparison between the analysed environments, including the liveness level of the dynamic and static analysis feedback offered by each environment.

Chapter 4

Survey on Working with Dockerfiles

This chapter presents a survey which aims to understand the issues of the current Dockerfile development process. Section 4.1 focuses on identifying the motivation for the execution of this survey, as well as how its results can be relevant for professionals and researchers. Section 4.2 presents the primary goal of this survey and the research questions that this survey attempts to answer. Furthermore, Section 4.3 describes the procedure used in order to collect the data. Section 4.4 presents and analyses the results from the survey. Finally, Section 4.5 identifies threats to the validity of the conclusions drawn, while Section 4.6 presents a brief summary of those conclusions.

The description of the procedures and approaches present in this chapter aims at allowing other researchers to perform this user study in other contexts. Along the same line, all the documents, resources and artifacts mentioned throughout this chapter (i.e. the questionnaire, all the collected data and data analysis scripts) are available in a **replication package**, stored as a public repository on GitHub [1].

4.1 Motivation

In recent years, Docker has become a *de facto* standard in containerization technologies. Docker uses a file, called *Dockerfile*, which contains text instructions, in order to generate an isolated environment called a *container* [33]. Given Docker's popularity and, consequently, the importance of developing Dockerfiles, it becomes relevant to characterize and optimize the Dockerfile development process that developers currently follow. This survey focuses on the Dockerfile development process and attempts to identify some of its flaws, which may then be improved upon with further work.

[1] Resources and artifacts related to this user study are available at `https://github.com/davidreis97/challenges_with_docker`

4.2 Goals and Research Questions

In Section 3.1, a set of tools currently available to Dockerfile developers is analysed. Through this analysis it was found that, to the best of our knowledge, the only tools which provide live feedback during Dockerfile development are some static analysis tools. On the other hand, Section 3.2 analyses a set of general-purpose development environments, some of which provide live dynamic analysis feedback. Through this analysis we found that the presence of live dynamic analysis feedback in the IDE can bring some improvements to the development process.

As such, this survey aims to expand on the conclusions drawn in Section 3.1 by further analysing the current Dockerfile development process. Its specific goal is to to verify if there are actual issues with the current process or if it generally works well as it is, without live dynamic analysis feedback.

Furthermore, this survey aims to understand if the issues that developers experience and the approaches that they use during development vary according to their level of experience. For this purpose, the survey was distributed to two different target audiences — one generally more experienced than the other. Section 4.3 presents more information about the distribution of the survey.

This survey also aims to guide the implementation of a solution which brings live dynamic feedback to the developer's environment. This is achieved by dividing the Dockerfile development process into the different main activities which constitute it and identifying the most time-consuming activities.

Therefore, this survey aims to help answering the following research questions:

- **RQ1** — How time-consuming is each activity of Dockerfile development?
- **RQ2** — Which approaches are used to overcome issues during Dockerfile development?

4.3 Data Collection

Two runs of this survey were performed with different respondents. The first run of the survey was answered by 68 students, most of whom were novices in Docker. The second run of the survey was targeted to professional software developers and answered by 110 respondents.

In the first run, the survey was distributed in person during classes of the 4th year, 1st semester, integrated masters in Informatics and Computer Engineering at the Faculty of Engineering of the University of Porto. We started by explaining to each class the goals of the study. After that we provided a *URL* where they could answer the survey in digital form. Students were asked to answer the survey immediately and we stayed in the classroom until everyone had completed their submission, helping students resolve any technical problems accessing the questionnaire. This distribution strategy had the purpose of reaching a wide number of respondents, as well as keeping at a minimum the class time spent on the survey. Since the survey's estimated completion time is 5 minutes and 6 classes were targeted, the survey's distribution took approximately 30 minutes. The survey was also distributed through email to the same target audience to allow for students who were absent from the classes to answer as well.

In the second run, the survey was first distributed in the context of the on-site research track of the XP 2020 Conference [2], an international conference on agile software development. The survey was also distributed in communities and forums directed towards Docker, DevOps and general-purpose programming. This distribution strategy had the purpose of reaching developers with some professional experience and a wide range of software engineering backgrounds.

In some questions of the survey, Likert-type rating scales are used. Likert scales are used to measure a subject's attitude — the personal level of agreement towards a certain statement [18]. In our implementation, this scale consists of 5 options: Strongly Disagree, Disagree, Neutral, Agree and Strongly Agree, which indicate the level of agreement of a respondent towards a statement.

The survey is divided into two sections: *Participant Characterization* and *Working with Docker technologies*.

In order to measure the Dockerfile development experience of the respondents, the section *Participant Characterization* of the survey focuses on gathering information about the professional experience of each respondent. In this section, using a Likert scale, respondents must state their opinion on their own level of experience working with Dockerfiles. In order to validate their response with a less opinionated parameter, respondents must also specify how many projects they have worked on that had a Dockerfile, used Dockerfiles created by others or created/updated a Dockerfile. They are also required to select the Dockerfile instructions that they've used during development. In the second run of the survey, the participants also specified their country, their professional sector and their main professional responsibilities.

The survey section *Working with Docker technologies* focuses on discovering which activities in Dockerfile development take longer than desired and gather some of the approaches used during Dockerfile development. In this section, using a Likert scale, respondents must indicate if they agree that they spend a lot of time in each of the development activities identified.

Furthermore, respondents can optionally specify in a text field what steps or strategies do they usually take in order to diagnose and solve bugs in Dockerfiles. Respondents must also specify which plugins/tools they use when developing a Dockerfile, other than a general-purpose text editor, if any.

A full version of the questionnaire can be found in Appendix B. This survey was developed in collaboration with another master's thesis which focused on docker-compose development. For that reason, the survey also contains questions regarding software development with docker-compose. However, since this dissertation focuses solely on Dockerfile development, only the set of questions regarding Dockerfiles are analysed.

[2]XP 2020 Conference Website available at `https://www.agilealliance.org/xp2020/`

4.4 Data Analysis

By measuring the attitude of the respondents towards the statement that a considerable time is spent in each Dockerfile development activity, we are able to identify the activities in which most of the developers believe to be considerably time-consuming, therefore answering the research question RQ1. By collecting responses about the general approach taken to solve problems in Dockerfiles as well as the plugins/tools used during development, it is possible to identify a set of approaches used, as well as their popularity, therefore answering the research question RQ2.

Questions regarding the professional experience of each respondent may help narrow down the conclusions extracted from the survey to a specific group or range of developers.

4.4.1 First Run

We only considered the responses of participants who had created/edited a Dockerfile in at least one project. As such, from the 68 total responses obtained, 19 were excluded, and the 49 remaining participants were considered in this analysis.

Regarding the professional experience of the respondents, we verified that the average number of projects where respondents had created/updated a Dockerfile was 2.4, showing that respondents had some experience, as expected from students. The median response was 2 and the standard deviation was 2.2.

In this run, the questions regarding the identified Dockerfile development activities were:

- **A1** — When I write a Dockerfile, I spend a lot of time *finding out what parent image is the most suitable.*
- **A2** — When I write a Dockerfile, I spend a lot of time *finding out what are the dependencies of the system that must be added to the docker image.*
- **A3** — When I write a Dockerfile, I spend a lot of time *finding out what are the Dockerfile commands that I need.*
- **A4** — When I write a Dockerfile, I spend a lot of time *trying to understand if the resulting container is working as intended (e.g., running commands and tests on the container).*
- **A5** — When I write a Dockerfile, I spend a lot of time *trying to understand why the resulting container is not working as intended.*
- **A6** — When I write a Dockerfile, I spend a lot of time *finding out which commands are responsible for the container misbehaviour.*
- **A7** — When I write a Dockerfile, I spend a lot of time *rebuilding the image and re-running the container to confirm that it is working as intended.*

The results shown in Figure 4.1 present the attitude of respondents towards the statement that a lot of time is spent on each of the identified development activities. These answers make it possible to answer the research question RQ1, defined in Section 4.2: respondents generally agreed that a lot of time is spent on most development activities. In particular, it can be observed that activities A2-A7 were considered time-consuming, with over half the respondents answering at least *"Agree"*. Among those, A4, A5 and A7 were considered particularly time-consuming, with over 60% of the respondents answering at least *"Agree"*. A1 was not considered to be particularly time-consuming. In A1 there was also a larger proportion of *"Neutral"* answers. We believe that this may be due to the fact that this activity is performed less times than other activities (i.e. a base image can be chosen just once at the start of the project, whereas finding and fixing can be a more iterative process).

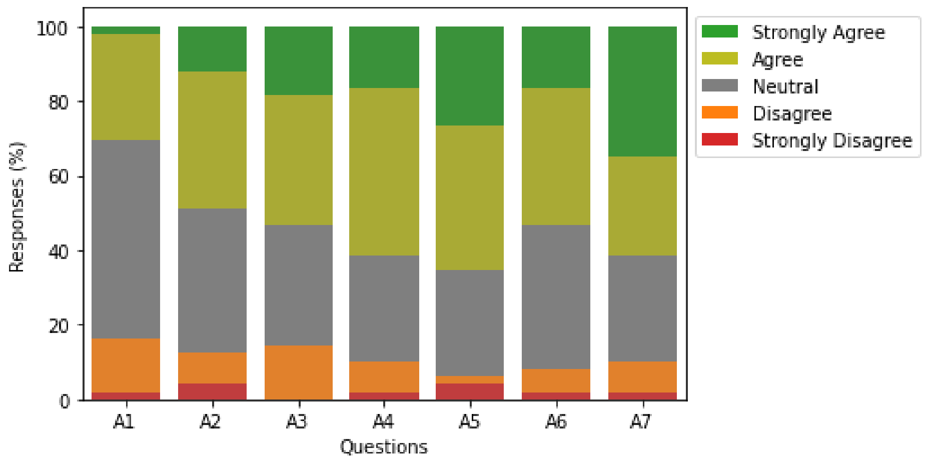


Figure 4.1: Attitude from respondents in the first run towards a lot of time being spent in each of the development activities

It was also noticed that 90% of the respondents do not use any plugin/tool other than a general-purpose IDE during Dockerfile development. From the remaining 10%, the only tool mentioned was the Docker plugin for Visual Studio Code, previously analysed in Chapter 3.

Answering the research question RQ2, a large majority of the approaches to solving problems in Dockerfiles that were mentioned by respondents mention *"trial and error"*, using Google and other online platforms, *"random guessing"* or entering the container to manually execute commands which may help diagnose the issue. We believe that these approaches can be slow and frustrating. This belief is further sustained by the fact that activities A3-A7, typically related to the debugging process of Dockerfiles, were found to be perceived as particularly time-consuming.

4.4.2 Second Run

As expected, the second run was answered by developers with a wider range of backgrounds and slightly more professional experience than the respondents of the first run. This analysis only takes into consideration the responses of the participants who had created/edited a Dockerfile in at least one project. As such, from a total of 110 responses, 1 response was discarded and the remaining 109 responses are considered in this analysis.

Regarding the professional experience of the respondents, we verified that the average number of years of experience in creating/updating a Dockerfile was 3.4. The median response was 3 and the standard deviation was 1.64. These results reinforce our belief that the respondents from this run are slightly more experienced than participants from the first run. Furthermore, a vast majority of the participants worked in the industry, with most of the remaining participants working in the academic sector. Participants were from 23 different countries, with Portugal (21%), the USA (19%) and Spain (9%) being the most represented countries. Most of the participants had responsibilities as a software developer, with a significant number also working in operations.

In this run, the questions regarding the identified Dockerfile development activities were:

- **A1** — When I write a Dockerfile, I spend considerable time *finding out what parent image is the most suitable.*
- **A2** — When I write a Dockerfile, I spend considerable time *finding out what are the dependencies of my system that must be added to the docker image.*
- **A3** — When I write a Dockerfile, I spend considerable time *finding out what are the right Dockerfile commands that I need.*
- **A4** — When I write a Dockerfile, I spend considerable time *confirming if the resulting container is working as intended.*
- **A5** — When I write a Dockerfile, I spend considerable time *trying to understand why the resulting container is not working as intended (e.g., running commands and tests on the container).*
- **A6** — When I write a Dockerfile, I spend considerable time *finding out which commands are responsible for the container misbehaviour.*
- **A7** — When I write a Dockerfile, I spend considerable time *rebuilding the image and re-running the container to confirm that it is working as intended.*
- **A8** — When I write a Dockerfile, I spend considerable time *reading Docker documentation.*

These questions are very similar to the questions presented in the first run — questions A1-A7 from both runs target the same Dockerfile development activities. However, in this run the questions were slightly modified to be more clear and objective. Nevertheless, we believe that these small changes should not significantly change the participants' interpretations of the questions. Question A8 was introduced in this run.

Figure 4.2 displays the attitude from respondents towards the statement that a considerable amount of time is spent in each of the identified development activities. These results allow us to answer the research question RQ1, defined in Section 4.2. The respondents generally agreed that considerable time is spent in most Dockerfile development activities. Activities A2, A4, A5 and

A7 were regarded as particularly time-consuming, with over 60% of the participants answering with at least "Agree". In A4 and A7 in particular, less than 10% of the participants answered with "Disagree" or less.

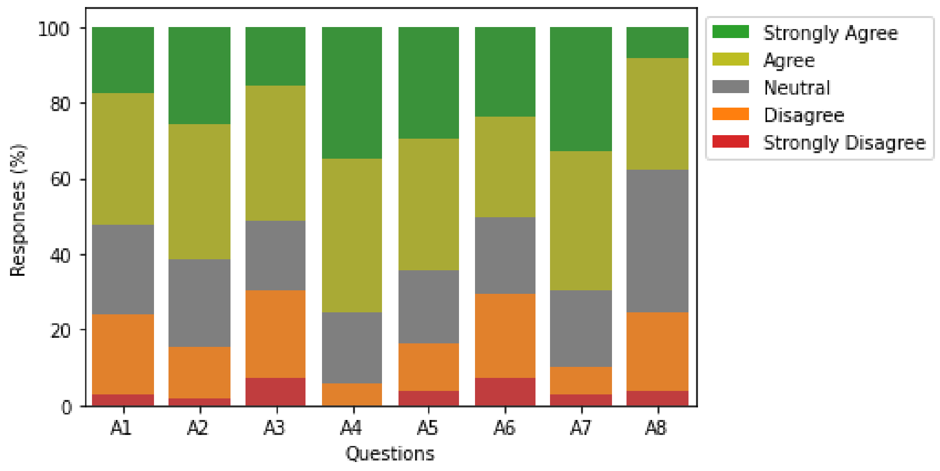


Figure 4.2: Attitude from respondents in the second run towards a considerable amount of time being spent in each of the development activities

It was also observed that only 25 out of the 109 respondents (i.e. 23%) mentioned using tools or plugins in order to develop Dockerfiles, with a vast majority consisting of static analysis tools such as the ones described in Section 3.1.5.

Directly answering RQ2, many participants described a trial and error approach, where they manually inspect the container using the command line after each Dockerfile change and iterate until the container works as intended. Some participants also mentioned structured approaches such as continuous integration pipelines and end-to-end testing.

4.5 Threats to validity

Survey differences. The survey published in the second run had a few changes when compared to the first run. This could affect any conclusions drawn from comparing the results from both runs. However, the results are consistent between the two runs of the survey, leading us to believe that the changes did not have a relevant effect on the answers.

Participant's geographic distribution. Even though in the second run participants had a much broader background, certain countries may have been overly represented due to the fact that some country-specific communities were approached in order to broadcast the survey. This could negatively affect the generalizability of the results to the population of Dockerfile developers at large. However, to the best of our knowledge, there is no evidence that a participant's country should significantly influence the conclusions drawn in this chapter.

Selection of activities. The subdivision of the Docker development process into multiple approaches was performed based on our experience with Dockerfile development. As such, other developers might find a different division of approaches as more suitable. However, participants did not raise this issue in neither of the runs, which gives us confidence that our set of approaches was generally well accepted.

4.6 Discussions

This section focuses on discussing the results presented in Section 4.4, directly answering the research questions defined in Section 4.2.

The research question RQ1 aims to understand how time-consuming each activity of Dockerfile development is. With the exception of the A1 activity in the first run of the survey and A8 in the second run of the survey, participants in general found most Dockerfile development activities to be time consuming. However, it should be noted that participants also displayed a slight tendency to identify the debugging activities of Dockerfile development, such as understanding why a container isn't working or re-building the Docker image, as the most time-consuming.

The research question RQ2 aims to identify the approaches used to solving bugs in Dockerfiles. In general, participants commonly mentioned approaches which rely on *"trial and error"* and manually building, raising and inspecting the images and containers under development. However, some participants from the second run mentioned using more structured and formal approaches, such as testing and pipelines. Furthermore, the tools and plugins used by the participants were consistent with the analysis on the state-of-the-art Dockerfile static analysis tools, presented in Section 3.1.5.

Overall, these results provide a better understanding of the most critical activities and the problems that developers face during Dockerfile development. This knowledge can help increase the impact of the live environment designed and implemented in Chapter 6 since the feedback generated by the environment can be tailored towards helping the developer during the activities perceived as most time-consuming.

Chapter 5

Problem Statement

This chapter describes the problems that currently exist in Dockerfile development, particularly focusing on the problems that this dissertation proposes to address. Section 5.1 describes these problems and elaborates on the current Dockerfile development process. Section 5.2 presents the main hypothesis that this dissertation aims to validate, while Section 5.3 presents the methodology used in order to validate this hypothesis, including a brief description of each step in the methodology.

5.1 Current Problem

Currently, developing Dockerfiles can be a slow task, since developers must repeatedly modify and test the Dockerfile until it produces a container that behaves as intended [15].

As thoroughly analysed in Chapter 3, there are some tools and approaches that can help during Dockerfile development and, therefore, may improve the efficiency of a developer working with Docker containers and Dockerfiles.

However, as previously mentioned, most of these tools don't provide live feedback, as they require explicit actions from the developer in order to function. For example, the Goss framework [1], introduced in Section 3.1.4, allows a developer to perform infrastructure tests which can help determine if the instructions written in a Dockerfile produce a Docker image and container with the intended behaviour. However, this requires some effort, as the developer needs to write the tests and execute the framework manually, often outside of his IDE. To the best of our knowledge, the tools which provide some live feedback only perform static analysis.

We believe that the aforementioned limitations restrict developers working in Dockerfiles to an iterative workflow which can often be slower than desired. As an example, Figure 5.1 presents a possible workflow of a Dockerfile developer.

[1]Links to the mentioned tools can be found in Appendix A

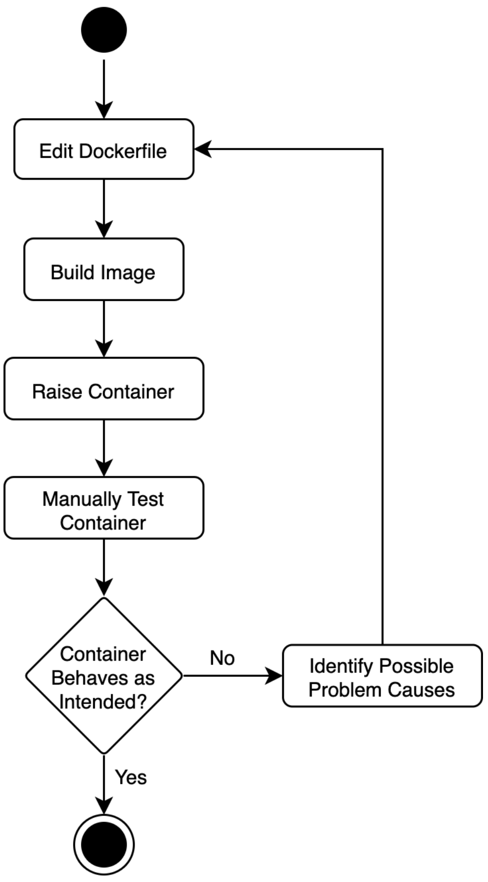


Figure 5.1: Possible workflow of a Dockerfile developer.

The survey presented in Chapter 4 further reinforces this belief. In this survey, developers generally indicated spending a considerable amount of time in most activities of Dockerfile development, particularly in the activities related to the debugging process of a Dockerfile. Furthermore, this survey also showed that developers commonly use a *"trial and error"* approach when solving problems related to Dockerfiles, regularly having to manually build, raise and inspect a container to verify its behaviour.

In addition, a study, which evaluated a representative sample of 560 open-source projects from the GitHub platform, showed that 34% of the Dockerfiles found in those projects could not be built without errors [9]. The same study showed that bad practices, such as the lack of a `MAINTAINER` tag or the lack of version pinning in the base image and dependencies, were also observed in a considerable amount of Dockerfiles, including Dockerfiles in the top 100 repositories according to GitHub's star rating [9]. These results show that the emergence of errors and bad practices during the Dockerfile development process is not uncommon.

Overall, the evidence presented in this section gives us confidence that the Dockerfile development process could be significantly improved by the implementation of live dynamic analysis feedback in the IDE.

5.2 Hypothesis

Taking into consideration the limitations identified in the current Dockerfile development process, described in Section 5.1, this dissertation aims to validate the following main hypothesis:

The Dockerfile development process can be more efficient if developers have a work environment with richer feedback and higher level of liveness than provided by the current state of the art tools and methodologies.

In this context, we define an increase in efficiency as a decrease in the time required for a developer to finish the Dockerfile development process. Furthermore, in the Dockerfile development process we include creating Dockerfiles from scratch, modifying existing Dockerfiles (either to add features or to fix bugs) and also diagnosing problems in existing Dockerfiles. Finally, according to the analysis performed in Section 3.1, only some static analysis tools provide more than the 2nd level of liveness when working with Dockerfiles. As such, having richer feedback and higher level of liveness can be achieved by providing feedback generated from dynamic analysis of the image and container generated from a Dockerfile, with the 4th level of liveness. It could also be possible to provide richer feedback by extending the current static analysis tools, although this dissertation focuses primarily in dynamic analysis.

As described in Section 2.2, there are multiple potential motivations for the implementation of liveness, such as improving the accessibility and comprehension of a system [30]. However, given the hypothesis described, it should be noted that the main motivation for liveness in this dissertation is productivity: accelerating some of the activities of the Dockerfile development process and, as a consequence, accelerating the Dockerfile development process as a whole.

5.3 Methodology

The primary objective of this dissertation is to validate the hypothesis defined in Section 5.2 and improve the Dockerfile development process.

As such, a **preliminary survey**, answered by students and professionals, was performed with the purpose of identifying the main issues that arise during the development of Dockerfiles and identifying the most time-consuming Dockerfile development activities. This preliminary survey is described in Chapter 4.

This survey and the analysis of the state-of-the-art, presented in Chapter 3, allowed us to design an **approach** which attempts to assist Dockerfile development and reduce the time spent in most of the Dockerfile development activities, particularly in those identified as time-consuming. This approach, which focuses on delivering live dynamic analysis feedback, allows the developer to have a workflow with shorter and more frequent feedback loops. According to the hypothesis presented in Section 5.2, this could improve the efficiency of the Dockerfile development process. Section 6.1 performs a detailed description of this approach.

In order to bring to life the aforementioned approach, a **reference architecture**, presented in Section 6.3, has been designed. This architecture is implemented as **Dockerlive** [2], an extension for the Visual Studio Code IDE. As designed in the approach, Dockerlive delivers live dynamic analysis feedback in the Dockerfile development environment.

Dockerlive was then used in a **controlled experiment with users**. In this experiment, the performance and behaviour of developers using Dockerlive were compared to the performance and behaviour of developers using an environment without live dynamic feedback. The results from this comparison allowed us to answer the main hypothesis of this dissertation, as well as to measure and understand the impact of live dynamic feedback in the Dockerfile development process. The methodology and the results obtained in this user study are thoroughly documented in Chapter 7.

[2]Dockerlive's VSCode Marketplace page is available at `https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive`

Chapter 6

Dockerlive: Approach and Reference Architecture

This chapter presents the solution implemented in order to generate live dynamic analysis feedback in an IDE. Section 6.1 defines an approach designed to generate and provide feedback with the 4th level of liveness. Section 6.2 presents a few key technologies that can be used in order to design an efficient architecture and implementation of the approach. Dockerlive Section 6.3 defines the architecture of the solution. Section 6.4 describes the features of Dockerlive, an implementation of the previously mentioned approach, which runs as an extension to the IDE Visual Studio Code. Dockerlive's architecture and implementation details are a reference architecture of the approach, presenting the architecture and mechanisms which can be used in order to deliver the approach in other contexts.

6.1 Approach

In order to validate the hypothesis described in Section 5.2, an approach has been designed with the aim of providing dynamic analysis feedback to the developer with the **4th level of liveness**. This approach allows the developer to have a workflow with shorter and more frequent feedback loops which, according to the hypothesis, could improve the efficiency of the developer. This approach consists of three steps: ***Identification of Interest Points***, ***Probing of Interest Points*** and ***Feedback Generation***.

In the ***Identification of Interest Points*** step, the Dockerfile is parsed in order to locate potential interest points that can be a source of valuable information to the developer. As such, this step relies on performing static analysis on the Dockerfile. For example, if the Dockerfile under development contains an `EXPOSE` instruction which exposes a certain port, then it could be valuable for a developer to know, without having to perform any manual checks, what service is running on the exposed port. On the other hand, some interest points are applicable to all

Dockerfiles and do not depend on the existence of certain instructions or patterns. For example, it can be useful for a developer to be able to visualise container performance statistics, regardless of the particular set of instructions in the Dockerfile. The heuristics which identify potential interest points are defined during the implementation of this approach.

In the ***Probing of Interest Points*** step, information is collected from the interest points identified in the previous step. For this purpose, an image is built and a container is raised from the Dockerfile under development. As such, this step performs dynamic analysis, gathering information about the Docker container and the Docker image.

Finally, in the ***Feedback Generation*** step, the information collected in the previous steps is processed and presented to the developer inside the IDE. Other IDEs have been analysed and documented in Section 3.2 in order to better understand how feedback can be provided in a way that is user-friendly.

These steps are executed every time that the developer makes a change to the Dockerfile in the IDE. For performance reasons, since these steps can be computationally intensive, if a user changes the Dockerfile before the end of the execution for the previous Dockerfile state, then the previous execution should be cancelled and a new one should be started.

This approach has the advantage of allowing the collection of data which regards the actual, live execution of the container. This allows the developer to have much richer information during development, as well as reducing the need to manually perform Docker-related actions such as building an image or raising a container. Another advantage of this approach is that it can be easily extended by identifying new interest points. However, an implementation of this approach must pay close attention to the performance, particularly on larger Dockerfiles, given the fact that image compilation times can be long [15].

The feedback generated by this approach can be coupled with the existing static analysis development tools for Dockerfiles which already provide up to the 4th level of liveness, documented in Section 3.1.5, allowing a developer to simultaneously take advantage of the richness of dynamic analysis and the immediacy of static analysis.

6.2 Main Technology Choices

This section provides some insight into some of the technologies that power the proposed approach and, therefore, are referenced in the architecture presented in Section 6.3 and in the description of the features presented in Section 6.4.

Visual Studio Code. In order to provide live feedback in the developer's development environment, we opted to implement an extension which runs in the general-purpose IDE Visual Studio Code [1]. This choice was made for two main reasons. Firstly, Visual Studio Code was the

[1]Links to the mentioned tools can be found in Appendix A

most popular IDE amongst DevOps developers in 2019 [2]. As such, by implementing liveness as an extension to Visual Studio Code, we provide liveness in an environment which the target users of our work are probably already familiar with. Secondly, Visual Studio Code offers an API which provides support to all the functionalities that our approach needs to work [3].

Language Server Protocol. The Language Server Protocol [4] (LSP) is a protocol which allows a single implementation of development assistive features, such as code auto-complete or syntax error highlighting, to be used across multiple different IDE's. The main advantage of this protocol is that it removes the need to have a unique implementation of these features for each language and for each IDE. Figure 6.1 presents a visual representation of the impact of the LSP in the implementation of development assistive features.

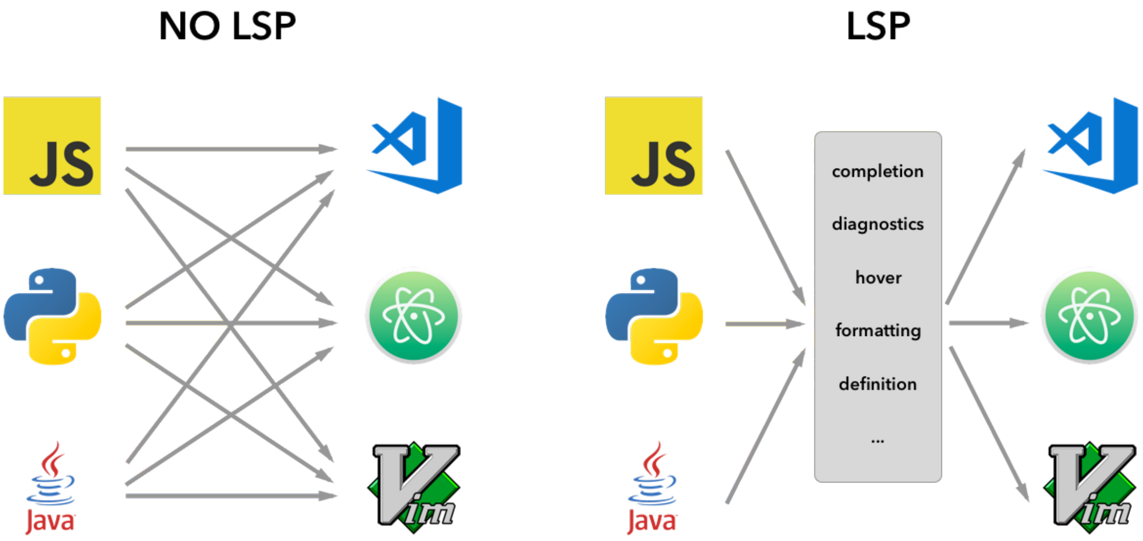


Figure 6.1: Impact of the LSP in the implementation of development assistive features through IDE extensions [a].

[a]Taken from https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

This protocol standardizes the communication between the IDE and a *Language Server*, which implements the development assistive features and can be used across multiple IDEs. Communication between the IDE and the Language Server is performed using the JSON-RPC protocol.

Dockerode One of the possibilities when it comes to interacting with Docker programmatically is to perform HTTP requests to the Docker API. This API works as an interface for performing

[2]Stack Overflow Developer Survey 2019 available at https://insights.stackoverflow.com/survey/2019

[3]Visual Studio Code Contribution Points available at https://code.visualstudio.com/api/references/contribution-points

[4]Documentation for the Language Server Protocol is avaliable at https://microsoft.github.io/language-server-protocol/

Docker-related actions (e.g. build an image) and is used, for example, by Docker's own command-line interface [5]. The *Dockerode* package [6] provides an interface for Javascript and Typescript solutions to perform these requests using object-oriented principles such as classes and objects instead of loose HTTP requests.

6.3 Architecture

Dockerlive implements this approach using a client/server architecture, where the client and server are two asynchronous entities which communicate in order to provide functionalities inside Visual Studio Code. Figure 6.2 presents an overview of Dockerlive's architecture using a component diagram.

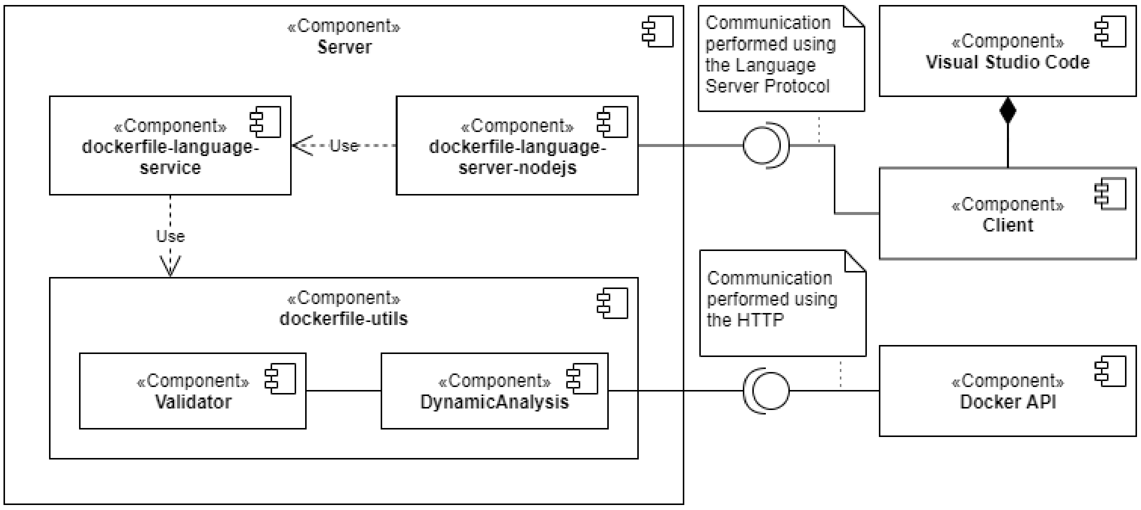

«Component» Server
«Component» dockerfile-language-service
Use
«Component» dockerfile-language-server-nodejs
Use
«Component» dockerfile-utils
«Component» Validator
«Component» DynamicAnalysis
Communication performed using the Language Server Protocol
«Component» Visual Studio Code
«Component» Client
Communication performed using the HTTP
«Component» Docker API

Figure 6.2: Dockerlive component diagram. Presents an overview of the main components which execute the dynamic analysis.

The *Server* component plays the role of the *Language Server*, as described in Section 6.2. Therefore, it is responsible for parsing Dockerfiles and generating the feedback and the functionalities that should be delivered to the user. It is completely separate from Visual Studio Code or any other IDE.

The server has three main components: *dockerfile-utils*, *dockerfile-language-service* and *dockerfile-language-server-nodejs*. These 3 libraries are also used by the *Docker for Visual Studio Code (Preview)* [7] extension, previously presented in Chapter 3. These libraries perform static analysis on Dockerfiles and report the results of the analysis using the Language Server Protocol. In Dockerlive, these libraries were extended in order to perform not just static analysis but also dynamic analysis, by automatically instantiating a container and performing the necessary actions to provide all the features detailed in Section 6.4. The *dockerfile-utils*

[5]Docker Overview — Docker Documentation available at https://docs.docker.com/engine/docker-overview/
[6]Dockerode NPM package available at https://www.npmjs.com/package/dockerode
[7]Links to the mentioned tools can be found in Appendix A

component, in particular, is responsible for analysing the Dockerfile and returning the results of the analysis to the *dockerfile-language-service* component. The *Validator* component of *dockerfile-utils* is responsible for executing the static analysis of the Dockerfile. If no errors are detected, the *DynamicAnalysis* component executes the dynamic analysis.

Extending the existent stack of libraries directed towards static analysis has a few performance benefits. Dynamic analysis, which can be computationally demanding for a consumer-grade machine, only needs to run if the Dockerfile is syntactically valid. For example, if the programmer mistyped the instruction `FROM` as `FRMO`, there's no need to try to build an image from the Dockerfile, as it's already clear that it won't compile successfully due to that typo. As such, Dockerlive skips the dynamic analysis if the static analysis found issues with the Dockerfile. This also reduces the probability of attempting to run a Dockerfile with transient semantics [1]. Another performance benefit consists of being able to use the same Abstract Syntax Tree (AST) that the static analysis uses to analyse the Dockerfile. As such, this AST is reused in the dynamic analysis with the purpose of deciding which actions and feedback should be generated (e.g. perform service discovery in each `EXPOSE` instruction, as described in Section 6.4.9).

In order to perform actions related to Docker artifacts, such as running a container or building an image, the server performs requests to the Docker API. Information is extracted from the container in ways that are supported by a wide range of container operative systems. As such, Dockerlive's features should work for a variety of Linux distributions. In order to provide a good user experience, if a certain feature is not supported in the operative system of the container that the developer is working on, the feature simply doesn't generate feedback, without throwing exceptions or crashing in a way that is visible to the user.

Furthermore, since containers can change their state during their execution, the server keeps making regular calls to the Docker API in order to update the information about the container.

The *Client* component serves as a bridge between the server and the IDE, presenting the feedback generated by the server to the user. The client is tied to and is executed within Visual Studio Code, as an extension, and is activated when a file of type *Dockerfile* is opened in the editor. The first action taken by the client, when it is activated by Visual Studio Code, is to instantiate the server. This client only interacts with the server, never directly interacting with the Docker API or with any Docker artifacts. During its execution, the client has two main duties:

- Send commands and events to the server, such as a document edit or a custom command, so that the server can generate new feedback or perform the action required by the custom command.
- Receive data from server and present it to the user as visual interface elements.

Dockerlive's architecture is based on the architecture of the *lsp-sample* example which is part of the official Visual Studio Code documentation [8].

[8] *lsp-sample* example project available at `https://github.com/microsoft/vscode-extension-samples/tree/master/lsp-sample`

By using the Language Server Protocol and performing all the analysis in a independent server, detached from the IDE, this extension could be implemented in other IDE's with little effort, as only a client would have to be developed. Further characteristics, advantages and disadvantages of the Language Server Protocol are detailed in Section 6.2.

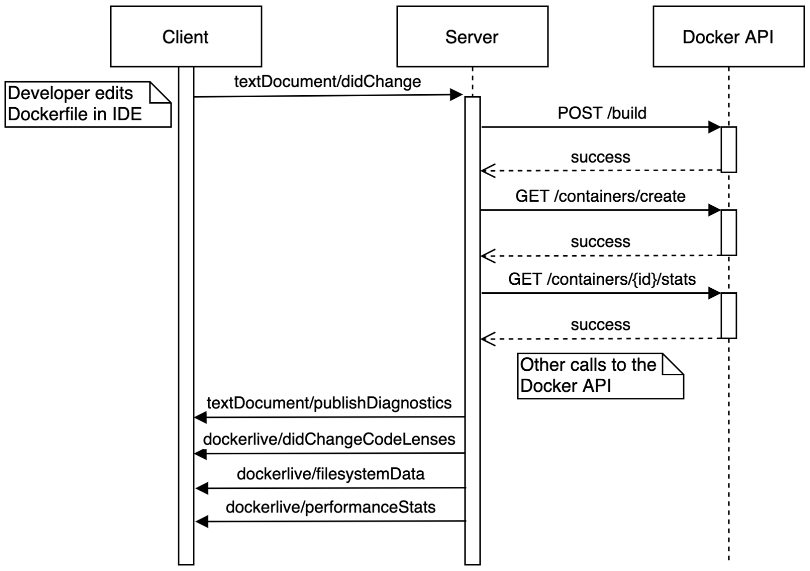

Client
Server
Docker API
Developer edits Dockerfile in IDE
textDocument/didChange
POST /build
success
GET /containers/create
success
GET /containers/{id}/stats
success
Other calls to the Docker API
textDocument/publishDiagnostics
dockerlive/didChangeCodeLenses
dockerlive/filesystemData
dockerlive/performanceStats

Figure 6.3: Dockerlive sequence diagram. Presents a simplified visualization of some of the messages shared between the client, the server and the Docker API.

Figure 6.3 presents a simplified visualization of a possible exchange of messages between the client, the server and the Docker API. In the sequence presented, the Dockerfile is edited by the developer in the IDE. As such, the client sends the new version of the document to the server. The server then builds an image, creates a container, gets the container's statistics and performs other calls to the Docker API in order to get more information about the container and the image. Further details about other calls performed to the Docker API are available in Section 6.4. After gathering and processing information about the image and the container created, the server sends the information back to the client, which displays it to the developer through the IDE. After this exchange of messages, the server would continue performing calls to the Docker API in order to update the information about the container, sending the updated information to the client.

6.4 Feature Design and Implementation

This section presents the features available in Dockerlive, describing their implementation and how they are presented to the developer within VSCode.

As described in Section 6.3, the Language Server Protocol is used to deliver feedback to the IDE. This implementation uses version v1.35 of the LSP. The following features of the LSP are used and referenced throughout this section:

- **Diagnostics** — Problems/potential suggestions targeting a specific part of the code, which can have a descriptive text and different severity levels: *Error*, *Warning*, *Information* or *Hint*, in decreasing order of severity. In VSCode, diagnostics are rendered as underlines in the targeted code section, where the color of the underline indicates the severity of the diagnostic. The descriptive text can be read by hovering over the targeted code section. As an example, the *Container Runtime Errors* feature of Dockerlive, described in Section 6.4.2, uses this mechanism.

- **Progress Reporting** — If the Language Server needs to perform any actions which may take a long time, it can use this mechanism to inform the IDE of the progress of the action, and the IDE can display that information to the user. In VSCode, progress is reported through the status bar. Dockerlive implements this feature, as described in Section 6.4.11, since some of its actions may take a while to be performed (e.g. building a Docker image).

- **Custom Messages** — Some of the features implemented in Dockerlive are not directly supported by the LSP. Therefore, the LSP allows the server and the client to communicate using their own custom messages. The downside of using custom messages is that unlike the natively supported features of LSP, which are implemented out-of-the-box in IDEs which support the LSP, the features implemented using this mechanism have to be adapted to each IDE.

As described in Section 6.1, a fundamental part of the approach consists in the identification and probing of interest points identified in the Dockerfile. Table 6.1 presents the main interest points where information is extracted from.

Table 6.1: Probing performed in each potential interest point.

Interest Point	**Probings**
`ENTRYPOINT`/`CMD` instructions	Container runtime errors (Section 6.4.2) Processes running in container (Section 6.4.3)
`ENV` instructions	Changes to environment variables (Section 6.4.4)
`FROM` instructions	Image OS information (Section 6.4.6)
`EXPOSE` instructions	Service discovery (Section 6.4.9)
Dockerfile as a whole	Container performance statistics (Section 6.4.5) Layer size and build time (Section 6.4.7) Layer file system explorer (Section 6.4.8) Image build and container output (Section 6.4.10)

In this implementation, the version of the Docker API used is v1.40. Throughout this section, multiple calls to the Docker API will be mentioned. Each call is further described in the official Docker API v1.40 documentation [9].

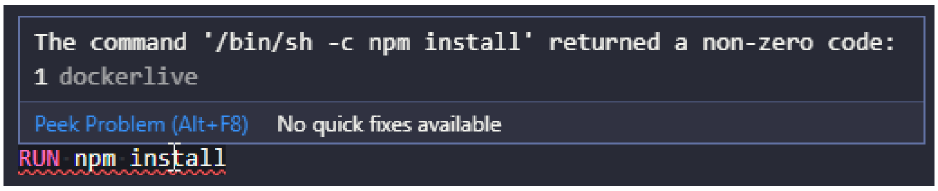
[9]Docker API v1.40 documentation available at `https://docs.docker.com/engine/api/v1.40/`

6.4.1 Continuous Image Build

In order to perform live dynamic analysis, every time the Dockerfile is edited inside VSCode the client sends the updated version of the Dockerfile to the server, which starts compiling an image from that Dockerfile. In order to build a new image, the contents of the directory where the Dockerfile is located are packed into a `tar` file — including the updated Dockerfile. The `tar` file is sent to the Docker API through the `POST /build` endpoint, which builds an image with the tag *TestImage*. Every time a new image is built, it is built with this tag, therefore ensuring that the image with tag *TestImage* is always the most recent one.

The output of the Docker API, which contains the image build logs, is used to monitor the progress of the build (e.g. *Step 3/5...*). The current build step is displayed using the Language Server Protocol progress reporting mechanism described in Section 6.4.11. The image build logs are also displayed in the *Output* panel, as described in Section 6.4.10. The intermediate image ids for each of the build steps, which, as mentioned in Section 6.4.7, are used to retrieve the size of each layer, are also extracted using the build output.

If the image building process fails for any reason, such as, for example, a RUN instruction returning with non-zero code, a diagnostic with `ERROR` severity is sent, targeting the instruction which caused an error. This causes that instruction to be underlined in red inside VSCode, as depicted in Figure 6.4. Dockerlive is able to determine the exact instruction which failed since each build step corresponds to an instruction in the Dockerfile and, as mentioned, the current step is monitored throughout the process. Hovering the underlined instruction shows more details about the error, which is also gathered from the Docker API's build output.



```
The command '/bin/sh -c npm install' returned a non-zero code:
1 dockerlive
```

```
RUN npm install
```

Figure 6.4: Error during the image build stage.

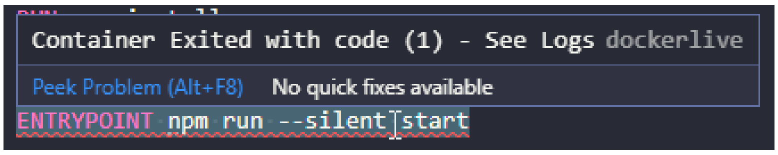
6.4.2 Container Runtime Errors

If the image is successfully built through the process described in Section 6.4.1, then a container is instantiated from that image. Since Docker containers must not have the same name, this container is named *testcontainer* followed by a large random number, to avoid any potential collision with existing containers.

Dockerlive continuously monitors this container by waiting for it to exit and checking its exit code. If the container exits with an exit code different than 0, then a diagnostic is sent with `ERROR` severity targeting the `ENTRYPOINT`/`CMD` instruction. In VSCode, this causes that instruction to be underlined in red, as shown by Figure 6.5. If there are no `ENTRYPOINT` nor `CMD` instructions in the Dockerfile, then the last instruction of the Dockerfile is underlined. Hovering the underlined instruction prompts the user to consult the logs, which might indicate the reason why the container exited with a bad exit code.

Dockerlive also attaches to the newly created container and displays its logs in the "Output" panel, as described in Section 6.4.10.

A similar diagnostic is also generated if there is an error creating the container, starting the container, getting the container's exit code or attaching to the standard output to the container, although these errors should not occur during Dockerfile development, as they are indicative of a problem related to Docker itself, rather than the Dockerfile being edited.

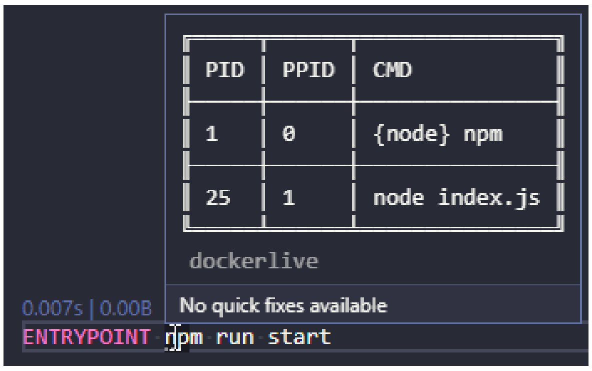


```
Container Exited with code (1) - See Logs dockerlive
Peek Problem (Alt+F8)    No quick fixes available
ENTRYPOINT npm run --silent start
```

Figure 6.5: Error after container exits with non-zero exit code.

6.4.3 Processes Running in the Container

Dockerlive allows the user to hover the `ENTRYPOINT`/`CMD` instruction (or the last instruction of the Dockerfile if there is no `ENTRYPOINT`/`CMD` instruction) in order to see the processes running in the container, as presented in Figure 6.6.

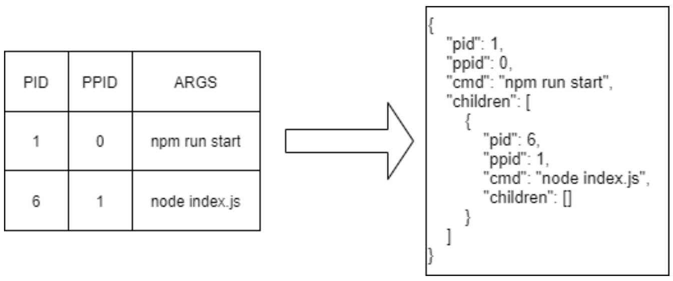


```
| PID | PPID | CMD           |
| 1   | 0    | {node} npm    |
| 25  | 1    | node index.js |
dockerlive
0.007s | 0.00B    No quick fixes available
ENTRYPOINT npm run start
```

Figure 6.6: Processes running inside container.

This information is retrieved by executing a `ps -eo pid,ppid,args` command inside the container. This command outputs the processes running in the container in a table format, providing the process ID (`pid`), parent process ID (`ppid`) and arguments (`args`) for each process running.

This information is then parsed, using the process IDs and parent process IDs in order to create a data structure which represents the process tree. This process tree is later used to analyse the environment variables of each process, as described in Section 6.4.4. Figure 6.7 shows an example of the conversion from the table of running processes to a tree.



PID	PPID	ARGS
1	0	npm run start
6	1	node index.js

```
{
  "pid": 1,
  "ppid": 0,
  "cmd": "npm run start",
  "children": [
    {
      "pid": 6,
      "ppid": 1,
      "cmd": "node index.js",
      "children": []
    }
  ]
}
```

Figure 6.7: Conversion to a tree data structure containing a container's running processes.

In order to continuously monitor the processes running in the container, this mechanism is executed every 500ms, allowing for the information in the developer's environment to be regularly updated. Furthermore, when a container exits, the running processes that were last detected before the container shut down are still available through this hover mechanism, along with a small text before the process table which indicates that the container has exited.

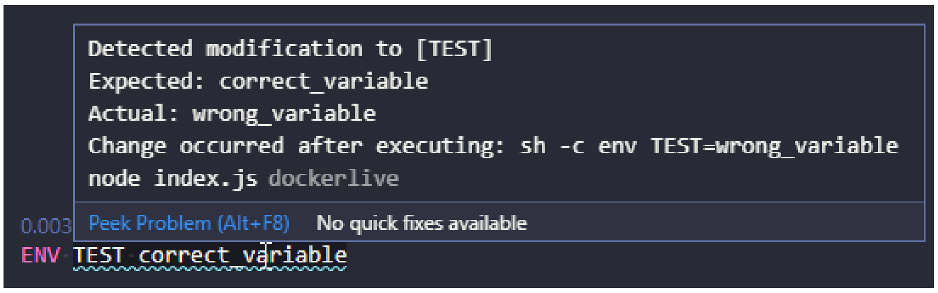
6.4.4 Changes to Environment Variables

Dockerlive continuously analyses the environment variables defined in the Dockerfile. The objective of this analysis is to identify situations where an environment variable is defined inside the Dockerfile and is redefined during the container's lifetime. As verified in the user study presented in Chapter 7, these situations can lead to bugs which may be time-consuming to detect and fix. In particular, the third task of the user study, detailed in Section 7.4, serves as an example of this kind of bug.

In order to detect and alert the developer of these potentially unwanted changes, the process tree described in Section 6.4.3 is used. In Unix systems, the initial environment variables of a process can be consulted by accessing a file called `/proc/<pid>/environ`, where `<pid>` is the process ID of the process under analysis [10]. By traversing the process tree and checking the environment variables of each process using the `/proc/<pid>/environ` file, it's possible to detect when a process changes environment variables, since these changes propagate to child processes. In other words, when a process is started with an environment variable with a value different than the one defined in the Dockerfile, its parent process is responsible for changing that environment variable. This allows Dockerlive not only to identify changes to environment variables, but also to pinpoint those changes to a culprit process, as seen in Figure 6.8.

As such, a diagnostic with `WARNING` severity is generated and sent to the client, underlining `ENV` instructions in blue when the value that they set is changed during the container's execution. Hovering the underlined instruction shows the expected value of the environment variable, the actual value of the environment variable in the container and the process which changed the environment variable. This mechanism is executed every 500ms, allowing for the information in the developer's environment to be regularly updated. Furthermore, this information persists in the user's interface even after the container exits.

[10]proc(5) — Linux manual available at https://man7.org/linux/man-pages/man5/proc.5.html



```
Detected modification to [TEST]
Expected: correct_variable
Actual: wrong_variable
Change occurred after executing: sh -c env TEST=wrong_variable
node index.js dockerlive
0.003 Peek Problem (Alt+F8)    No quick fixes available
ENV TEST correct_variable
```

Figure 6.8: Warning after change is detected in environment variable.

6.4.5 Container Performance Statistics

Dockerlive users can visualize performance statistics of the running container by clicking the "CPU" button, located in the upper right corner of the editor tab which contains the Dockerfile. By clicking this button, a new tab with performance graphs becomes visible. If the graphs are updated every second, then the container is running. If the graphs are stopped, then the container is stopped. When a new container starts, data is erased from the graphs. Figure 6.9 shows an example of these performance graphs.

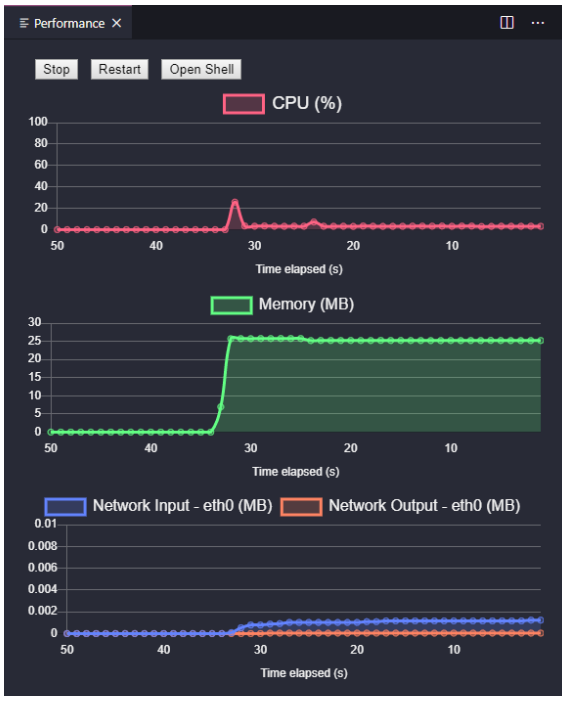


Figure 6.9: Performance monitoring webview displaying CPU, Memory and Network information.

In order to extract this information, a call to the Docker API is performed through the `GET /containers/{id}/stats` endpoint which provides performance information of a container. This information is then sent to the client. Since the Language Server Protocol does not define a payload to send graphical or performance data, this data is sent through a custom message, containing the latest performance statistics. The client, upon receiving this message, updates the data displayed in the graphs. The graphs are displayed in a tab inside VSCode, which contains a *webview* running HTML, CSS and JS code to render the graphs. Since this type of message does not belong to the Language Server Protocol specification, every client implementation needs to implement the retrieval and rendering of the performance data.

This feature is rendered in a *webview* inside VSCode, meaning that the file system data must be sent from the server to the client, and from the client to the *webview* in the form of an asynchronous message, as described in the VSCode documentation [11].

Furthermore, on this webview, there are also three buttons available:

- Stop — Stops the running container
- Restart — Restarts/Starts the container
- Open Shell — Open an interactive shell inside the container.

These buttons provide the developer with a quick way to interact with the running container. When a developer presses one of these buttons, a message is sent from the client to the server indicating which action it should take (e.g. pressing the *Stop* button sends a `dockerlive/stop` message to the server). The server then interacts with the Docker API through the `POST /containers/{id}/stop` to execute the action requested by the client.

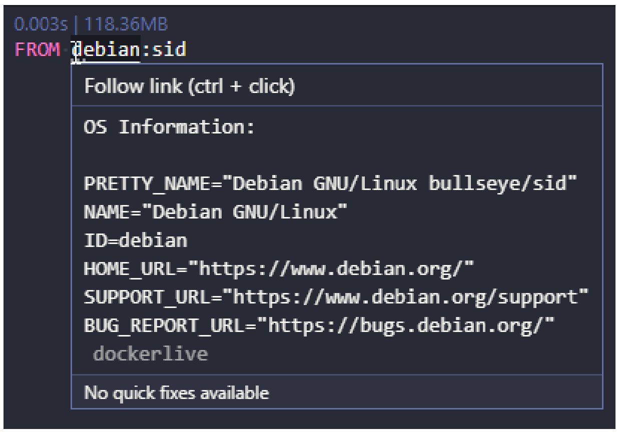
When a developer presses the *Open Shell* button, an editable text box becomes visible which contains the command `docker exec -it <containerName> /bin/sh`, where `<containerName>` is the name of the current running container. By pressing *Enter*, the command is executed in a terminal inside VSCode. This allows the developer to quickly open the `/bin/sh` shell, often present in Linux distributions, but still allowing the developer to run other executables.

6.4.6 Image OS Information

Dockerlive also attempts to extract information about the operative system running in the container. If that information is successfully retrieved, a diagnostic on the `FROM` instruction is generated and sent to the client with a severity of `INFORMATION`. In VSCode, the developer is able to hover over the image name and see some information about the OS that the container is running. As an example, Figure 6.10 shows the output of this feature for a Dockerfile with the base image `debian:sid`.

Since different operative systems have different footprints, multiple strategies have to be adopted in order to be able to provide support for a wide range of operative systems. These strategies consist of executing a command inside the container which, if successful, return information about the operative system. When a strategy is successful (i.e. the command exits with exit code 0), the remaining strategies do not need to be executed. In order of priority of attempts, the following commands are attempted:

[11] VSCode Webview API Documentation available at https://code.visualstudio.com/api/extension-guides/webview



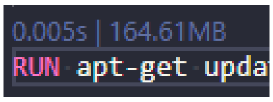
```
0.003s | 118.36MB
FROM debian:sid
Follow link (ctrl + click)
OS Information:

PRETTY_NAME="Debian GNU/Linux bullseye/sid"
NAME="Debian GNU/Linux"
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
 dockerlive
No quick fixes available
```

Figure 6.10: Operative System information indicating the distribution running in the container.

1. `cat /etc/os-release`
2. `cat /etc/lsb-release`
3. `cat /etc/issue`
4. `cat /proc/version`
5. `uname`

Since this mechanism relies on executing commands inside the container, these commands are attempted after the container is started, as described in Section 6.4.2.

6.4.7 Layer Size and Build Time



```
0.005s | 164.61MB
RUN apt-get upda
```

Figure 6.11: CodeLens showing layer size and build time.

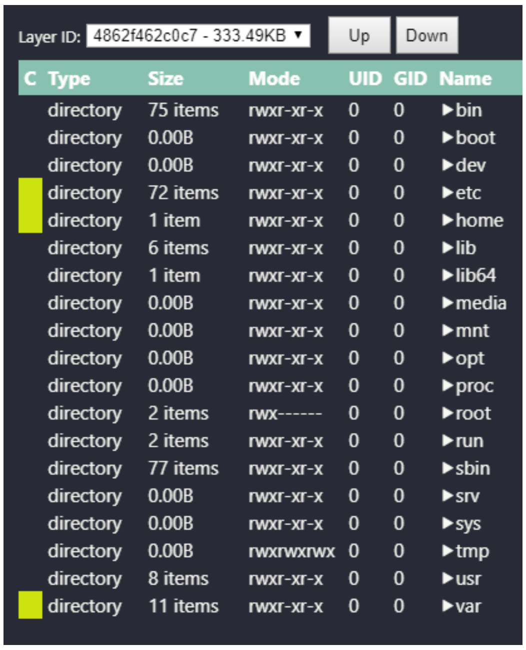
CodeLens is a feature of VSCode which allows for short text to be exhibited above certain lines of code. Dockerlive makes use of this feature in order to show, above every instruction in the Dockerfile, the time it took to build an intermediate image from that instruction and the size of that image. Figure 6.11 shows an example of this feature.

The time is calculated using the Docker API build output, which is received during the image build process. Since Dockerfile instructions are executed sequentially, the time that a certain instruction takes is the time that has elapsed since the previous instruction ended or since the start of the image build process in case the instruction is the first one in the Dockerfile.

The size of each layer is obtained with a call to the Docker API through the `GET /images/{name}/history` which reveals the intermediate images of a Docker image, as well as their size. This call is performed targeting the image tagged "TestImage", which through the mechanism described in Section 6.4.1 is the result of the compilation of the Dockerfile which has most recently been edited by the developer in VSCode. Each intermediate image with a valid ID corresponds to an instruction in the Dockerfile. As such, since layers are stored in the same order as the instructions in the Dockerfile, it's possible to identify which instruction gave origin to each intermediate image. After establishing this relationship, CodeLenses are created and sent to the client.

CodeLens functionallity is supported by the Language Server Protocol. However, in the Language Server Protocol it's not possible to add CodeLenses a document after the initial CodeLenses have been sent. Since we believe that, for a good user experience, CodeLenses should be visible almost immediately after a user edits a document, the implementation of CodeLens in Dockerlive relies on a custom CodeLensProvider in order to provide CodeLenses as soon as possible, while still being able to add new CodeLens at any time.

6.4.8 Layer File System Explorer



C	Type	Size	Mode	UID	GID	Name
	directory	75 items	rwxr-xr-x	0	0	▶bin
	directory	0.00B	rwxr-xr-x	0	0	▶boot
	directory	0.00B	rwxr-xr-x	0	0	▶dev
🟨	directory	72 items	rwxr-xr-x	0	0	▶etc
🟨	directory	1 item	rwxr-xr-x	0	0	▶home
	directory	6 items	rwxr-xr-x	0	0	▶lib
	directory	1 item	rwxr-xr-x	0	0	▶lib64
	directory	0.00B	rwxr-xr-x	0	0	▶media
	directory	0.00B	rwxr-xr-x	0	0	▶mnt
	directory	0.00B	rwxr-xr-x	0	0	▶opt
	directory	0.00B	rwxr-xr-x	0	0	▶proc
	directory	2 items	rwx------	0	0	▶root
	directory	2 items	rwxr-xr-x	0	0	▶run
	directory	77 items	rwxr-xr-x	0	0	▶sbin
	directory	0.00B	rwxr-xr-x	0	0	▶srv
	directory	0.00B	rwxr-xr-x	0	0	▶sys
	directory	0.00B	rwxrwxrwx	0	0	▶tmp
	directory	8 items	rwxr-xr-x	0	0	▶usr
🟨	directory	11 items	rwxr-xr-x	0	0	▶var

Figure 6.12: Webview displaying the file system of the layers inside an image.

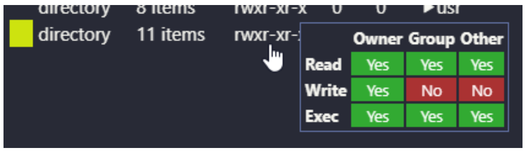
Dockerlive users can display and explore each layer's file system by clicking in the "FS" button, located in the upper right corner of the editor tab which contains the Dockerfile. By clicking this button, a new tab is displayed. This tab allows the developer to navigate through the aggregated file system of each of the intermediate layers. Figure 6.12 shows an example of this feature. A short explanation on the decomposition of an image in multiple intermediate layers is available in Section 2.1.3.

Each file system entry is described in 7 columns:

1. **C** — If a file system entry or one of it's descendants has been created, changed or deleted in the selected intermediate layer, then a yellow square is displayed in this column.
2. **Type** — Shows the type of a file system entry (e.g directory, file, symbolic link).
3. **Size** — Shows the size of a file system entry. If the entry is a directory, this column shows the number of items inside the directory.
4. **Mode** — Shows the mode (i.e. the permissions) of the file system entry in the Unix permissions format.
5. **UID** — Shows the UID of the owner of the file system entry.
6. **GID** — Shows the GID of the owner of the file system entry.
7. **Name** — Shows the name of the entry.

On the top of the table there is a dropdown menu which allows you to select the intermediate layer that is currently being displayed. You can also expand and collapse folders by clicking on their name.

By hovering over the mode of an entry, you can see a small window which shows a visual representation of the mode. Figure 6.13 shows an example of this feature.


directory 11 items rwxr-xr-

	Owner	Group	Other
Read	Yes	Yes	Yes
Write	Yes	No	No
Exec	Yes	Yes	Yes

Figure 6.13: Window displaying the mode of a file system entry.

If a layer deletes an entry, the entry is still visible in the layer which deleted it, but it has a red background and a type of 'removal'.

This feature is rendered in a *webview* inside VSCode, meaning that the file system data must be sent from the server to the client, and from the client to the *webview* in the form of an asynchronous message, as described in the VSCode documentation [12].

In order to retrieve the intermediate layers which compose the image tagged "TestImage", a call to the Docker API is performed through the endpoint `GET /images/{name}/get`. This call returns a `tar` stream which contains the multiple intermediate layers of the image, as `tar` files, as well as a JSON file which defines the correct order of the intermediate layers. Each intermediate layer `tar` file contains the changes that were performed to the file system in that layer. Therefore, in order to present to the developer the cumulative result of two or more layers, the layers are merged.

As an example, consider that a Docker API call to retrieve the image's intermediate layers returned a tar stream with the files *23fg298h3.tar*, *guh309fj3.tar* and *json*.

[12]VSCode Webview API Documentation available at https://code.visualstudio.com/api/extension-guides/webview

If the *json* file showed that the layer *23fg298h3* is the root layer, then if the user, in VSCode, selects the *guh309fj3* layer in the dropdown input field, then the user is able to see the result of merging both these layers (and the changes performed by layer *guh309fj3* marked with a yellow square in the *C* column in the table).

In order to merge the intermediate layers but still keep track of the changes made by each individual layer, at least two approaches can be used. The first approach is to keep all layers separate until the user selects a layer in the dropdown, and only then calculate the cumulative file system resulting from all the layers until the layer that the user selected. The two main advantages of this approach is that there is no need to make any file system merges until the user selects a layer in the dropdown and also that the changes made in each layer are also easily accessible. However, merging multiple layers when the user selects a layer in the dropdown can be computationally expensive and cause a delay to happen. For this reason, another approach was chosen in Dockerlive.

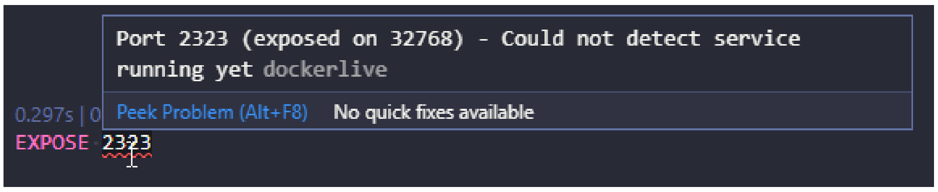
The approach implemented in Dockerlive consists of calculating the cumulative result for all layers at the end of the image build process. For example, in an image with layers A, B and C, this approach calculates the merge of layers A + B and A + B + C automatically at the end of the image build process. This way, when the user selects the layer B, for example, then the result of merging all the layers until the layer B (i.e. A + B) has already been calculated, therefore taking much less time to present it to the user than in the previous approach. The collection of changes made by each individual layer is also sent to the client, providing a way to visually mark the changes made by each particular layer. The main disadvantage of this approach is that more data needs to be sent from the server to the client. For example, in the first approach, a packet would be sent containing [A, B, C], where each entry contains just the changes performed in that layer. In the second approach, a packet would be sent containing [A, B, C, A+B, A+B+C]. For this reason, in Dockerlive, if packets exceed a size of 262144 bytes, they are split into multiple packets before being sent to the file system webview, where they are joined and displayed.

Each file system entry is stored in the following data structure, called FilesystemEntry:

```
{
    type: FileType,
    permissions: PermissionObject,
    uid: number,
    gid: number,
    size: number,
    children: FilesystemEntryCollection
}
```

In the context of this data structure, FileType represents a file system entry type (e.g. ’file’, ’link’, ’symlink’), PermissionsObject is an object which stores the permissions of the entry in the Unix permissions format and FilesystemEntryCollection is a list which stores multiple objects of type FilesystemEntry.

6.4.9 Service Discovery



```
Port 2323 (exposed on 32768) - Could not detect service
running yet dockerlive
```

Figure 6.14: Service discovery information showing a TCP service running on port 3000.

Dockerlive automatically tries to detect any services running on ports that are exposed with the `EXPOSE` instruction. If no service is detected, an error underline in the `EXPOSE` instruction is displayed. By hovering the port number on the instruction, you can see the name and protocol of the detected service. Figure 6.14 shows an example of this feature.

Service discovery is performed with the help of Nmap [13]. One of Nmap's features consists of detecting which services and versions are running in a specific port of a host. By using Nmap, Dockerlive can automatically perform service discovery at the ports which are exposed in the Dockerfile and present the output of Nmap as a diagnostic with `INFORMATION` severity level.

In order to be able to perform this scan, the container's exposed ports must be published. Publishing the ports ensures that all the exposed ports in the Dockerfile are mapped to random, available ports on the host machine. For the ports to be published, the flag *PublishAllPorts* is set to true in the Docker API call to the `GET /containers/create` endpoint, which creates the container. In order to discover which host ports were mapped to all the exposed ports, a call to the Docker API is performed through the `GET /containers/{id}/json` endpoint, which inspects and returns information about a container — including the port mappings. After receiving this information, the following Nmap command is executed:

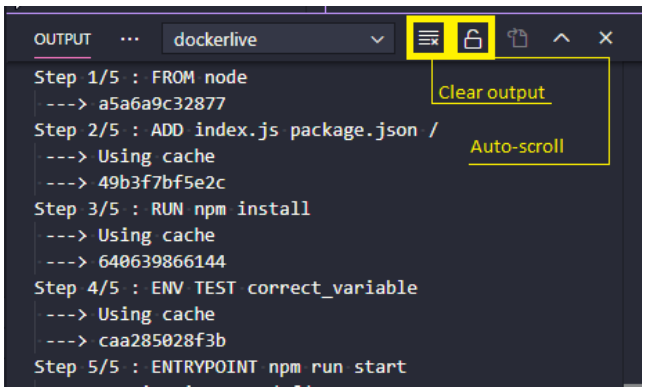
```
nmap -oX - 127.0.0.1 -p [port1,port2,...]  -sV -version-light
```

The flag `-oX` instructs Nmap to return results in the XML format. Since the scan must target the host machine, the target IP address for the scan is `127.0.0.1`. The flag `-p [port1,port2,...]` instructs Nmap to scan the specified ports. These ports are the ones that were obtained using the call to the Docker API. The flag `-sV` instructs Nmap to perform a Service and Version Detection scan. The flag `-version-light` instructs Nmap to run a light version of this scan, which can introduce some imperfect results (e.g. some services are less likely to be accurately identified). However, since Nmap can, even with this flag, always detect if a service is running and the speed of the feedback is very important in live development environments, we believed that the performance benefits are worth the slightly worsened accuracy.

The results from Nmap are parsed and mapped to the corresponding ports on the Dockerfile and diagnostics are created and sent, as previously described.

[13]Links to the mentioned tools can be found in Appendix A

6.4.10 Image Build and Container Log Output



```
Step 1/5 : FROM node
 ---> a5a6a9c32877
Step 2/5 : ADD index.js package.json /
 ---> Using cache
 ---> 49b3f7bf5e2c
Step 3/5 : RUN npm install
 ---> Using cache
 ---> 640639866144
Step 4/5 : ENV TEST correct_variable
 ---> Using cache
 ---> caa285028f3b
Step 5/5 : ENTRYPOINT npm run start
```

Figure 6.15: Image build and container log output. Two features are highlighted in yellow: clear all output and automatically scroll down as new output is displayed.

In order to improve the development flow while using Dockerlive, the output of the Docker image build and the Docker container is displayed in the *Output* panel in VSCode. This panel opens automatically when a Dockerfile is opened and may also be opened from the VSCode top bar (View → Output). Figure 6.15 shows an example of this feature. By using VSCode's *Output* panel, the user has access to features such as automatically scrolling down as new output is displayed and clearing all output.

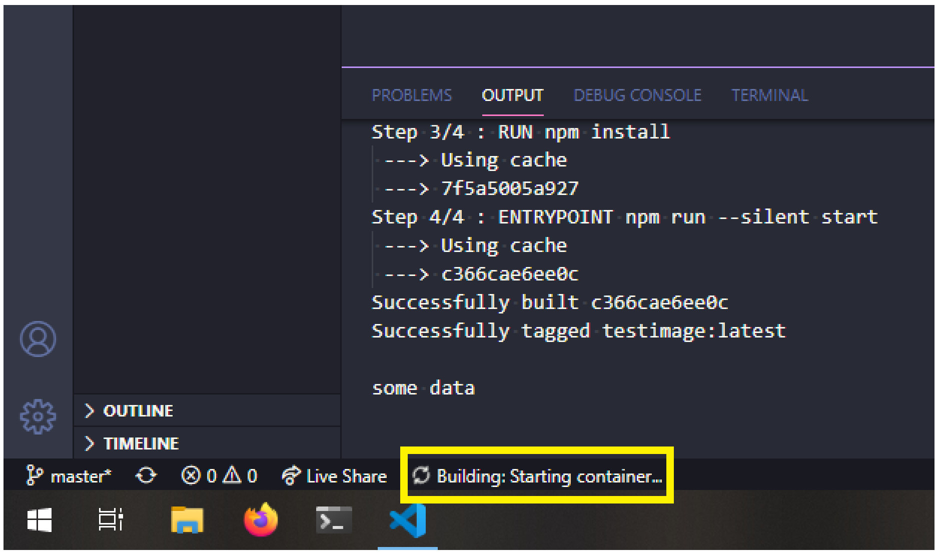
6.4.11 Progress Reporting

In order to efficiently and discretely inform the user about the progress of the analysis, a progress reporting mechanism has been implemented. The server sends the progress information using the `workDoneProgress` protocol which is part of the Language Server Protocol. In VSCode, this information is exhibited in the status bar, as shown in Figure 6.16.

This mechanism supports the following states:

- Build step information (e.g. "Step 1/4: FROM ubuntu")
- "Creating container..."
- "Starting container..." (as shown in Figure 6.16)
- "Running nmap..."

These states were chosen since they reflect the status of the dynamic analysis progress and the test container. Any time that Dockerlive is not performing one of these actions, the progress reporting interface is not visible in VSCode.



```
Step 3/4 : RUN npm install
 ---> Using cache
 ---> 7f5a5005a927
Step 4/4 : ENTRYPOINT npm run --silent start
 ---> Using cache
 ---> c366cae6ee0c
Successfully built c366cae6ee0c
Successfully tagged testimage:latest

some data
```

Figure 6.16: Progress report showing the status of the extension.

6.5 Deployment

Dockerlive has been deployed to the Visual Studio Code Marketplace [14] where it is available for download, free of charge. At the time of writing, Dockerlive has 272 installs and two 5-star reviews in the marketplace's rating system. In order to better understand the usage of the extension, Dockerlive also contains some telemetry features, implemented through Microsoft Azure [15], although at the time of writing not enough data has been collected to perform a significant analysis.

Dockerlive is also available as an open-source project at GitHub [16].

[14]Dockerlive's VSCode Marketplace page is available at https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive

[15]Links to the mentioned tools can be found in Appendix A

[16]Dockerlive's GitHub page is available at https://github.com/davidreis97/Dockerlive

Chapter 7

Empirical Evaluation

This chapter describes the controlled experiment with users performed in order to validate the main hypothesis of this dissertation, defined in Section 5.2. Section 7.1 and Section 7.2 start by presenting the main motivation, goals and research questions for this user study. The methodology and approaches used are presented in Section 7.3. Section 7.4 presents the experimental tasks that participants were asked to perform during the user study. Section 7.5 discusses the methods used to collect data during the user study, while Section 7.6 analyses the methods used to recruit participants with balanced characteristics. Section 7.7 presents a detailed analysis of the results of the user study, as well as the main conclusions. The main threats to the validity of this user study are discussed in Section 7.8. Finally, Section 7.9 summarizes the main findings of this user study and how they help to answer the research questions and the dissertation's main hypothesis.

The description of the procedures and approaches present in this chapter aims at allowing other researchers to perform this user study in other contexts. Along the same line, all the documents, resources and artifacts mentioned throughout this chapter (i.e. questionnaires, instruction documents, task code, all data collected and data analysis scripts) are available in a **replication package**, stored as a public repository on GitHub [1].

7.1 Motivation and Goals

The main motivation of this user study is to understand if the presence of live dynamic feedback in the development environment can positively influence the efficiency of a developer working in a Dockerfile. Understanding this makes it possible to validate the main hypothesis, presented in Section 5.2.

This study also aims to understand how live dynamic feedback can affect the behaviour of developers working in a Dockerfile. In other words, we wish to understand which Dockerfile

[1]Resources and artifacts related to this user study are available at https://github.com/davidreis97/DockerliveUserStudy

development activities are impacted by the presence of liveness, as well as how and why they are impacted.

Furthermore, this user study should make it possible to validate some of the design choices that were made during the design of the approach, described in Section 6.1, and its implementation, described in Section 6.4.

Another goal is to try to understand if the developers feel that the environment is comfortable and useful, regardless of the results obtained in other objective and precise measurements.

Finally, by combining the results of this study with the results of the survey described in Chapter 4, it's also possible to assess the perceived improvements brought by liveness in each activity of Dockerfile development, paying special attention to the activities that the participants of the survey identified as most time-consuming.

7.2 Research Questions

This user study aims to help answering the following research questions:

- **RQ1** — How can the presence of live dynamic analysis feedback *change the way that developers spend time while developing Dockerfiles?*
- **RQ2** — How can the presence of live dynamic analysis feedback *aid the developer in each Dockerfile development activity?*
- **RQ3** — How can the presence of live dynamic analysis feedback *change the user-perceived helpfulness and overwhelmingness of the IDE while fixing a Dockerfile?*

7.3 Methodology

In order to answer the research questions defined in Section 7.2 and validate the main hypothesis, presented in Section 5.2, a controlled experiment with users was designed and performed. This user study has the prospect of measuring and understanding the effect that an increase in live feedback can have in the Dockerfile development process. For this purpose, this study compares the use of two distinct Dockerfile development environments: an environment with live dynamic feedback, and an environment without live dynamic feedback. As such, this controlled experiment makes it possible to obtain the data required to compare the performance, development process and personal opinion of developers executing Dockerfile development tasks within a controlled environment, with and without live dynamic feedback. After obtaining this data, we can compare the two environments and, therefore, draw conclusions which verify the validity of the main hypothesis and answer the research questions.

Tasks. Participants should be asked to perform 3 independent tasks, which consist of editing a Dockerfile until the container built from that Dockerfile behaves as previously indicated to the participant. In other words, each of the 3 tasks consists of having the participant fix a Dockerfile. Tasks are individually described in Section 7.4.

Development Environment. Participants should be separated into two groups: a control group and an experimental group. Both groups should be given access to equivalent materials and asked to execute the same tasks. The difference between both groups is that the development environment provided to the control group only has the *Docker for Visual Studio Code (Preview)* [2] v1.2.1 extension (previously analysed in Chapter 3) while the experimental group also has, in addition to the aforementioned extension, *Dockerlive* [3] v1.0.17, which is an implementation of the reference architecture documented in Chapter 6. As such, users from the experimental group have access to live dynamic and static analysis feedback, while users from the control group only have access to live static analysis feedback, which, as described in Chapter 3, can be quite limited.

Participant Assignment. In order to ensure, to the best of our abilities, that the only significant difference between the control group and experimental group is the liveness in their development environment, a set of demographic and personal experience questions have been placed in a questionnaire. This questionnaire aims at gathering data related to the experiment and is thoroughly described in Section 7.5.2.

By ensuring that the answers from both groups to these questions are similar, we can be more confident that both groups have similar abilities and, therefore, reduce the risk of having the participants' own abilities significantly influence the results of the study.

As such, during the assignment of participants to the groups, two main aspects should be considered: the number of participants in each group and their answers to the demographics and personal experience questions should be as similar as possible.

More information on participant recruitment and selection is disclosed in Section 7.6.

Experimental Procedure. In order to ensure that the performance of the participant's machine does not affect the results of the experiment, all participants should execute the tasks in the same remote machine, to which they connect using a remote desktop tool such as *TeamViewer* [2]. In order to solve any potential technical issues regarding the remote desktop environment, participants establish an audio connection with the observing researcher at the start of the experiment.

After connecting to the remote desktop environment, participants should be given access to the instructions document which includes the instructions and rules for the experiment, a tutorial for the extension features that the participants have at their disposal inside the IDE and a link to the questionnaire. Participants should be instructed to start reading and follow the instructions presented in this document. The instructions documents for both groups is available in Appendix C and the questionnaires for both groups are available in Appendix D.

[2]Links to the mentioned tools can be found in Appendix A

[3]Dockerlive can be found at `https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive`

According to the instructions, participants start by answering the *Demographic Information* and *Skills and Experience* section of the questionnaire. After this, they should be given 15 minutes to read the rest of the instructions document. They are also allowed to consult the instructions document at any time during the execution of the experiment. The next section of the instructions document contains a list of all the extension features they have at their disposal in the IDE. Since the experimental group has access to the *Dockerlive* extension, this section of the instructions document is slightly larger for the experimental group. Features are described using a short paragraph and a few images. This ensures that all participants have a basic understanding of the features provided by their IDE so that their performance isn't excessively influenced by their lack of experience using the provided extensions.

The next section of the instructions document presents the global rules and instructions for the experiment. These rules are the following:

- In each task, participants are given a Dockerfile which they must edit until the container has the desired behaviour.
- Each task ends once either (a) the participant notifies the observing researcher that they've reached the desired behaviour or (b) the participant gives up or (c) the participant spends more than 20 minutes in the same task.
- Participants may only edit the Dockerfile. No other file in the task folder needs to be edited in order to achieve the desire behaviour. However, participants are allowed to make temporary changes to the code (e.g. print a variable or comment a line), which must be reverted for the task to be considered successfully executed.
- Participants are allowed to consult the instructions document at any time.
- Participants are allowed to consult any documentation and perform any web searches they may need, at any time. However, they must do so in the remote computer where they're performing the tasks, so that it's possible for the observing researcher to measure the time spent in the web browser.
- If something isn't clear in the instructions file or in the descriptions of the tasks themselves, participants must alert the observing researcher immediately.

Participants then read the instructions for the first task, execute the first task, fill the questionnaire section destined for the first task, and repeat these steps for the two other tasks. During the task execution, the observing researcher performs time and context switch measurements, which are described in Section 7.5.1.

After finishing the last task, the participants fill the *After Tasks* section of the questionnaire, after which the experiment is over.

The remote desktop environment should be reset after every participant executes the tasks — the browser history should be cleared, to avoid leading other participants to consult previously visited documentation and the Visual Studio Code's recent file feature should be cleared as well. Docker artifacts (images, containers, volumes) are cleared not only after each participant, but also after each task.

7.4 Tasks

In this experiment, participants must perform three tasks. All tasks follow the same pattern: a Dockerfile and a few other files are provided to the participant, and the participant must edit that Dockerfile until the container exhibits the desired behaviour. The desired behaviour is described in the instructions document.

7.4.1 First Task

In the first task, participants are given a folder containing the following files:

- *Dockerfile* — The Dockerfile which must be edited in order to achieve the desired container behaviour.
- *file* — A file containing the text "some data"
- *package-lock.json* — A file containing information regarding the *NodeJS* project.
- *package.json* — A file containing information regarding the *NodeJS* project, including the script which starts the project.
- *start_file.js* — The script which is executed by the project's start script. It attempts to read the file *file* and print it's contents to the standard output. If an exception is thrown, it prints "Could not read file" to the standard output.

In order to successfully complete the task, the following container behaviour must be observed:

- Container standard output must show the text *some data*.
- Container must have a *node* process running.

In this task, instead of adding the file *start_file.js*, the Dockerfile attempts to add the file *start_flle.js*, which doesn't exist. In order to fix this problem, the participant could change the `ADD` instruction or the name of the file so that the name of the file in the Dockerfile matches the name of the file in the file system.

Furthermore, the *start_file.js* script attempts to read the file *file*, which is not being added to the container. By adding this file using the `ADD` instruction, the script becomes able to read the intended file. After implementing these fixes, the container behaves as desired.

7.4.2 Second Task

In the second task, participants are given a folder containing the following files:

- *Dockerfile* — The Dockerfile which must be edited in order to achieve the desired container behaviour.
- *downloader.py* — The python script which is executed at the container's entrypoint. This script downloads a 10MB file from the internet and attempts to write the downloaded data to a file on the root directory of the container named *10Mio.dat*.

In order to successfully complete the task, the following container behaviour must be observed:

- Container standard output must show the text *Success!*.
- Container standard output must not show: *Error downloading file* nor *Error writing file*.
- Container must download 9MB-15MB of data during its lifetime.

In this task, the Dockerfile's parent image is an image tagged `davidreis1997/df2:1`. This image contains the *10Mio.dat* file in its root directory. However, this file is configured so that the default user has no permissions to write. For this reason, the python script, which runs at the start of the container with the default user permissions, is not able to write on it.

For this task, there are at least two possible solutions. The first one consisted of adding the instruction `USER root` to the Dockerfile, which gives permissions to write on the *10Mio.dat*. Another possible fix is to add the instruction `RUN chmod +777 10Mio.dat` which gives all the possible permissions to the *10Mio.dat* file, including the permission to write. After implementing one of these fixes, the container behaves as desired.

7.4.3 Third Task

In the third task, participants are given a folder containing the following files:

- *Dockerfile* — The Dockerfile which must be edited in order to achieve the desired container behaviour.
- *index.js* — The script which is executed by the project's start script. It attempts to launch a server which attempts to bind to the IP address stored in the environment variable `IP_ADDRESS`.
- *package-lock.json* — A file containing information regarding the *NodeJS* project.
- *package.json* — A file containing information regarding the *NodeJS* project, including the script which starts the project.

In order to successfully complete the task, the following container behaviour must be observed:

- Container standard output must show the text *Listening!*.
- Container standard output must not show: *Could not bind*.
- Container must have a TCP service running on the exposed port 3000.

In this task, the Dockerfile defines the `IP_ADDRESS` environment variable using the instruction `ENV IP_ADDRESS 0.0.0.0` and launches the *index.js* script at the container's entrypoint using the command `ENTRYPOINT npm run start`. However, the `start` script in `package.json` starts by defining the environment variable `IP_ADDRESS` to 3.3.3.3, overwriting the definition of that environment variable in the *Dockerfile*.

In order to make sure that the value of the `IP_ADDRESS` defined in the *Dockerfile* is not overwritten, the `ENTRYPOINT` instruction can be changed to `ENTRYPOINT node index.js`, which directly starts the script without changing the environment variable. After implementing this fix, the container behaves as desired.

7.5 Data Collection

This section details the two sources of data in this experiment — task measurements and the participant questionnaire. Task measurement consist of precise numerical attributes, such as time, which are measured by the observing researcher during the execution of the tasks. The participant questionnaire provides data input directly by the participants through a questionnaire.

7.5.1 Task Measurements

Task measurements are made by the observing researcher during the execution of the tasks, and are measured individually for each task. Two types of data are extracted: **task times segmented by context** and **context switches**.

In order to assess the participant's time usage during the tasks, time is measured separately across different contexts. Four contexts were defined:

- Using VSCode (except when using VSCode's integrated terminal).
- Using a terminal (including VSCode's integrated terminal).
- Using a web browser.
- Consulting the instructions document.

All actions that participants can do during a task fit into one and only one of these contexts. Therefore, the sum of the time spent in these four contexts is the total time spent by a participant in a task. For example, if a participant, during a certain task, spent 1 minute using VSCode, 2 minutes using a terminal, 2 minutes using a web browser and 1 minute consulting the instructions document, then that participant took 6 minutes to execute that task.

The number of context switches is also measured separately, indicating how many times the participant switched to a certain context.

In order to measure these parameters, the Android app *Turns Timer* was used. Figure 7.1 shows an example screenshot of the app, including a description of the UI elements used for these measurements.

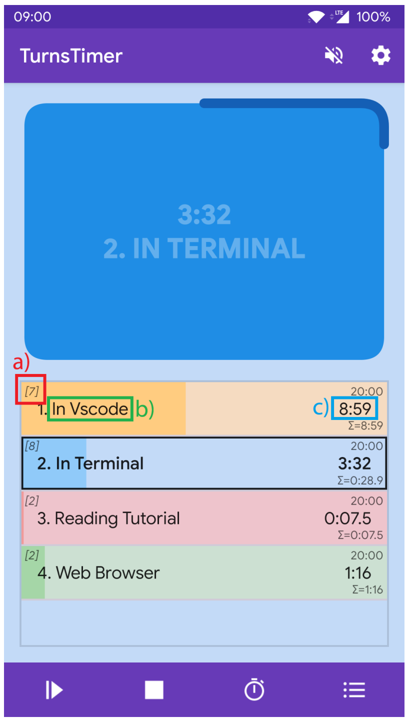


Figure 7.1: Screenshot of the app Turns Timer. a) Switches to the context; b) Name of the context; c) Time spent in the context

7.5.2 Participant Questionnaire

The participants are asked to answer a questionnaire in order to gather their opinion about the features they have at their disposal and the tasks that they have to execute, as well as to gather some demographic and personal experience information. This questionnaire is available in Appendix D.

In some questions, Likert-type rating scales are used. Likert scales are used to measure a subject's attitude — the personal level of agreement towards a certain statement [18]. In our implementation for this user study, this scale consists of 5 options: Strongly Disagree, Disagree, Neutral, Agree and Strongly Agree, which indicate the level of agreement of a respondent towards a statement.

Before the experiment. Before the start of the experiment, participants answer the ***Demographic Information*** and ***Skills and Experience*** sections of the questionnaire. These sections focus on gathering relevant demographic and professional experience information which can then be used to ensure that both the control and experimental groups have similar characteristics. This is very important, since the only significant difference between the two groups should be the experimental condition: the live feedback and features that participants have at their disposal. Demographic questions include the age, gender and highest degree of education of participants. Questions related to the skill and experience of the participants include their familiarity with technologies such as Docker, Dockerfiles, Visual Studio Code, NodeJS and Python, which are answered using a Likert scale. Participants are also asked to disclose the number of projects in which they've had contact with and edited a Dockerfile. This last question intends to provide a more objective measurement of experience in Dockerfile development when compared to the questions answered with Likert scales.

During the experiment. At the end of each task, the participants answer the section of the questionnaire respective to the completed task. Although the questions are the same for the three tasks, they must be answered separately, once for each task. The first question asks the participant how clear and simple to understand the task instructions were, which can be used to ensure that the instructions are consistently being well understood by all the participants. The second question asks which IDE features were useful during the execution of the tasks.

After the experiment. After all the tasks, participants answer the only remaining section of the questionnaire. The first and third part of this questionnaire section make it possible to understand how much participants felt that their IDE environment helped them solve the proposed tasks. The second part of this questionnaire section makes it possible to understand if they felt overwhelmed by the amount of information available in the IDE — something that, as described in Section 3.2.1, should be avoided. The fourth part of this questionnaire section consists of three open-ended questions. The first question tries to understand which features the participant felt were the most useful, as well as understand the reason behind their answer. The second and third questions try to understand any problems that the participant might have run into during the tasks, as well as possible suggestions on how to improve the level of liveness they used to solve the tasks.

7.6 Recruitment and Demographics

In the run performed, from the total of 20 participants who participated in the user study, 17 were contacted directly. The vast majority of these participants were students or very recent graduates from the integrated masters in Informatics and Computer Engineering at the Faculty of Engineering of the University of Porto. The remaining participants were also students from software-engineering related courses. During this initial contact, a few of the demographic and personal experience questions were asked. According to their responses, they were placed in one of the groups, so that the groups would be as balanced as possible. 3 other participants responded to a call for participants sent via email to 412 MSc students in software-engineering related courses from the Faculty of Engineering of the University of Porto. These 3 participants filled a separate preliminary questionnaire containing just the demographic and personal experience sections of the main questionnaire, described in Section 7.5.2, and were placed in one of the groups according to their answers.

7.7 Data Analysis

A run of this user study was performed with a sample of 20 participants — 10 participants belonged to the control group and the other 10 belonged to the experimental group.

This section compares the data retrieved from the both groups using statistical tests which analyse if two sets of samples come from the same distribution, answering the research questions presented in Section 7.2 and, ultimately, the main hypothesis presented in Section 5.2.

In this user study, the Mann-Whitney U statistical test [25] was chosen to prove the significance of the differences between the two groups, since this test does not assume that the data belongs to a normal distribution and can be robust to a broken assumption. This test was used with a 95% confidence level.

Furthermore, regarding the questions which use a Likert scale, each step of the Likert scale was mapped to a numerical value:

- **Strongly Disagree** — 1
- **Disagree** — 2
- **Neutral** — 3
- **Agree** — 4
- **Strongly Agree** — 5

This mapping allows the processing of Likert scale questions using methods which handle numerical data, such as the arithmetic mean, standard deviation or the Mann-Whitney U test.

7.7.1 Demographic Information

Regarding demographic information, the participants provided their gender, age and highest completed degree of education at the time of the experiment. Since, as expected, the gender or age of participants did not influence the results in any way, Figure 7.2 presents just the highest completed level of education of the participants.

The highest completed degree of education slightly favours the control group, where 4 participants already had completed a master's degree. However, since the groups were balanced in order to have similar levels of experience with Docker technologies, we believe that this slight imbalance should not be observable in the final results.

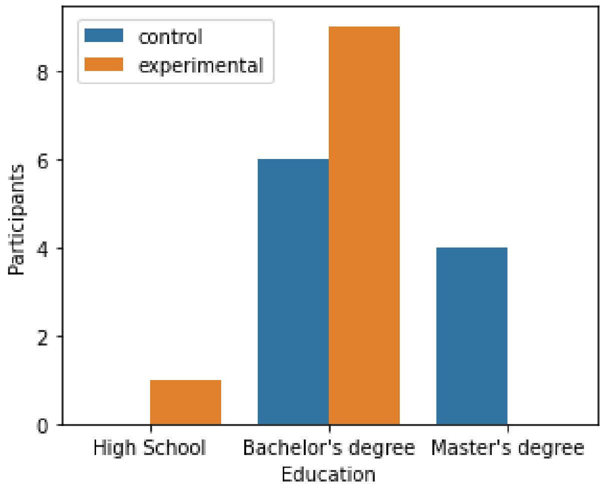


Figure 7.2: Participant's highest completed level of education, separated by group.

7.7.2 Skills and Experience

Regarding the skills and experience of the participants, the following Likert scale questions were answered:

- **Q1.1** — At this point in time I am comfortable working with *Docker technologies.*
- **Q1.2** — At this point in time I am comfortable working with *Dockerfiles.*
- **Q1.3** — At this point in time I am comfortable working with *Visual Studio Code.*
- **Q1.4** — At this point in time I am comfortable working with *Visual Studio Code to edit Dockerfiles.*
- **Q1.5** — At this point in time I am comfortable working with *linters.*
- **Q1.6** — At this point in time I am comfortable working with *syntax highlighters.*
- **Q1.7** — At this point in time I am comfortable working with *Unix operating systems.*
- **Q1.8** — At this point in time I am comfortable working with *Javascript and NodeJS.*
- **Q1.9** — At this point in time I am comfortable working with *Python.*
- **Q2.1** — I have a good enough understanding of English so that I can confidently answer this survey and understand sentences with Docker and Unix related terms.

Table 7.1 shows the mean and standard deviation of the answers provided by both groups. A two-tailed Mann-Whitney U was also performed in order to assess if there were significant differences between the two groups, with the **null hypothesis** that both groups belong to the same distribution and the **alternative hypothesis** that the groups show a statistically significant difference in results. Considering a confidence level of 95%, since none of the tests shows a ρ value where $\rho < 0.05$, we can say that there is not enough difference between the experimental and control group answers to reject the null hypothesis. In other words, the groups didn't provide significantly different answers to this set of questions.

Table 7.1: Mean, Standard Deviation and Mann-Whitney U for Likert questions performed in the *Skills and Experience* section.

Question	Control		Experimental		Two-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q1.1	**3.7**	0.78	**3.7**	1.27	45	**0.71**
Q1.2	**3.6**	0.8	**3.3**	1.87	57.5	**0.58**
Q1.3	**4.3**	0.46	**4.6**	0.49	35	**0.2**
Q1.4	**3.8**	1.08	**3.8**	1.25	49	**0.97**
Q1.5	**3.5**	1.02	**3.5**	0.5	50	**0.97**
Q1.6	**4.1**	0.94	**3.8**	0.75	62	**0.34**
Q1.7	**3.8**	0.6	**3.8**	0.75	47	**0.82**
Q1.8	**4.2**	0.87	**4**	0.89	56	**0.66**
Q1.9	**3.1**	1.13	**3.3**	1.18	44	**0.66**
Q2.1	**4.7**	0.46	**4.9**	0.3	40	**0.3**

Participants also provided the number of projects where they worked on which had Dockerfiles (**Q3.1**), where they used Dockerfiles created by others (**Q3.2**) and where they created/updated Dockerfiles (**Q3.3**). Notice that only **Q3.3** implies that the participant actually developed a Dockerfile, being, for that reason, perhaps the most important question from these three.

Table 7.2: Mean, Standard Deviation and Mann-Whitney U for project questions performed in the *Skills and Experience* section.

Question	Control		Experimental		Two-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q3.1	**4.6**	2.54	**4.6**	2.50	45.5	**0.75**
Q3.2	**2.6**	1.69	**3.2**	2.32	43.5	**0.65**
Q3.3	**2.9**	2.07	**2.2**	1.67	60	**0.47**

Table 7.2 presents the mean, standard deviation and, in similar fashion to the analysis performed for the Likert scale questions, a two-tailed Mann-Whitney U test which aims to assess if the differences between the two groups were statistically significant. Once again, considering a confidence level of 95%, since none of the tests shows a ρ value where $\rho < 0.05$, we can say that there is not enough difference between the experimental and control group answers to reject the null hypothesis. In other words, the groups also provided similar answers to this set of questions.

In sum, the answers provided to these questions show that both groups had similar previous experiences working with Dockerfiles — both in their own opinion and also in the number of projects they had worked before. These results provide a solid base for the conclusions presented in the rest of this analysis, since they diminish the possibility of one of the groups performing better on the proposed tasks due to having more experience with the technologies or systems used.

7.7.3 Task Understanding

It's important to verify if all the participants understood the task's instructions clearly and easily, since the challenges that they face during the execution of the tasks should come from the difficulties presented by the tasks themselves and not from the interpretation of the instructions. As such, three Likert scale questions were performed in order to assess the participant's understanding of the proposed tasks — one question for each of the tasks. In order to ensure that one of the groups didn't present a significantly different level of understanding than the other group, a two-tailed Mann-Whitney U was also performed. The **null hypothesis** for this test is that both groups belong to the same distribution and the **alternative hypothesis** that the groups show a statistically significant difference in results.

Table 7.3: Mean, Standard Deviation and Mann-Whitney U for Likert scale questions regarding the understanding of the three proposed tasks.

Task	Control		Experimental		Two-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Task 1	**4.4**	0.66	**4.6**	0.49	43	**0.58**
Task 2	**4.8**	0.4	**4.8**	0.4	50	**0.95**
Task 3	**5**	0	**4.8**	0.4	60	**0.17**

Table 7.3 shows the results of these calculations for each of the three tasks. Considering a confidence level of 95%, since none of the tests shows a ρ value where $\rho < 0.05$, we can say that there is not enough difference between the experimental and control group answers to reject the null hypothesis. In other words, both groups had similar success in understanding the instructions of the task.

Furthermore, looking at the full set of answers provided, it can be observed that from a total of 60 answers obtained to these three questions, only 1 answer was less than *Agree* in the Likert scale. As such, it can be comfortably stated that the participants generally had no trouble understanding the objective of the task or the task's instructions.

It can also be noticed that the mean answer, in both groups, tends to grow with each question. This is possibly due to the fact that the set of instructions for each of the three tasks had similar structure and, therefore, participants got more comfortable with the format of the instructions throughout the experiment.

7.7.4 Total Task Time

In order to help answer the research question RQ1, mentioned in Section 7.2 and validate the main hypothesis of this dissertation, mentioned in Section 5.2, we analyse and compare the time that participants from both groups took to solve the designated tasks. This section focuses on this analysis.

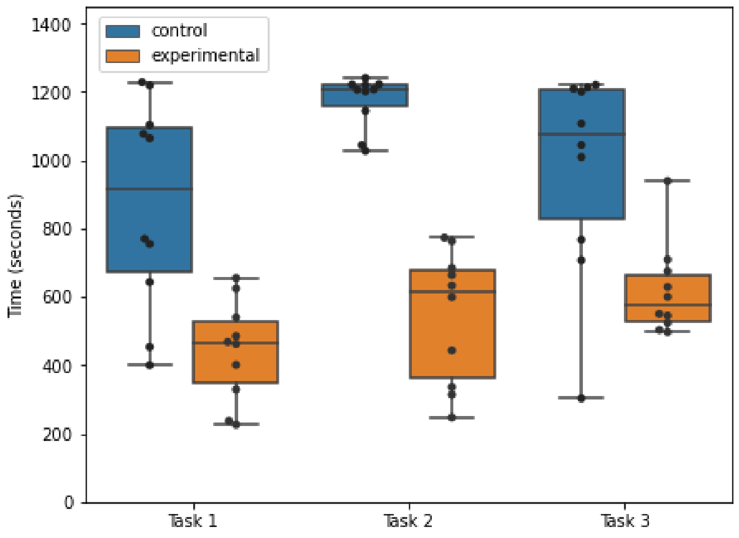


Figure 7.3: Total time used by participants in each task, separated by group.

Figure 7.3 shows a *boxplot* representation of the distribution of the total times used by participants in order to finish each of the three tasks. From this figure, it's possible to see that in all the tasks, the experimental group generally finished the three tasks significantly faster than the control group.

In order to confirm that the experimental group was significantly faster than the control group in the three tasks, three one-tailed Mann-Whitney U tests were performed. The **null hypothesis** for these tests is that the results from both groups belong to the same distribution, while the **alternative hypothesis** is that the experimental group shows significantly lower results.

Table 7.4: Mean, Standard Deviation and Mann-Whitney U for the total times used by participants in each of the tasks. Times are in the format *mm:ss*.

Task	Control		Experimental		One-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Task 1	**14:31**	04:52	**07:23**	02:19	86	**< 0.01**
Task 2	**19:33**	01:13	**09:06**	03:04	100	**< 0.01**
Task 3	**16:18**	04:45	**10:17**	02:07	87	**< 0.01**

Table 7.4 shows the results of these tests. Considering a confidence level of 95%, since in all three tests $\rho < 0.05$, it's possible to confidently state that the participants in the experimental

group were significantly faster than the participants in the control group.

Directly addressing the main hypothesis of this dissertation, as well as the research question RQ1, these results demonstrate that developers working in an environment with richer feedback and a higher level of liveness were **significantly more efficient** than developers working in an environment without those features.

Since the participants and the observing researcher have a limited amount of time to spend in this experiment, the time spent in each task has been limited to a maximum of 20 minutes. For the purpose of this analysis, cases where the participant exceeds the limit task time of 20 minutes were treated as if the participant completed the task at the time that they were interrupted (around 20 minutes). This approach was chosen since, despite yielding results which are lower from the truth (i.e. these participants would have finished in more than 20 minutes), all of these cases happened in the control group. This fact ensures that this approach doesn't negatively affect the conclusions drawn in this section, since without the 20 minutes time limit, the difference between the control and experimental group would be either equal or larger, but never smaller. An alternative approach would be to exclude these participants from the user study. However, this would result in a significantly lower number of observations in the control group, which would be negative for the user study.

7.7.5 Segmented Task Time

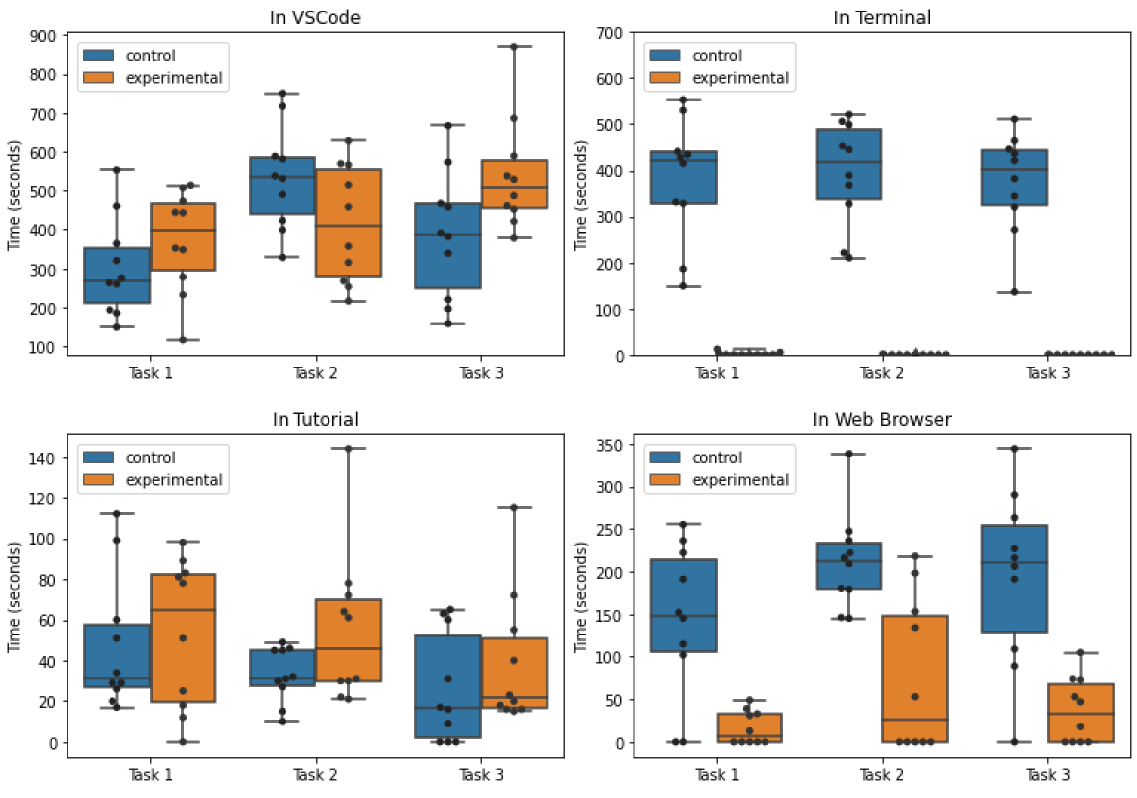


Figure 7.4: Total time used by participants in each task, separated by context and group.

As described in Section 7.5.1, the time that developers spent in each task was counted separately for 4 different contexts: **VSCode**, **Terminal**, **Web Browser** and **Tutorial**. As described in Section 7.5.1, the **Tutorial** context accounts for all the time that a participant spends consulting the instructions document. The sum of the times spent in each of these contexts equals the total task time, analysed in Section 7.7.4.

By comparing the time that the experimental group and the control group spent in each of the contexts, we can try to understand how the increased live feedback changes the way that developers spend time during the development process. This also helps answering the research question RQ1, mentioned in Section 7.2.

Figure 7.4 shows the *boxplot* representation of the time spent by the participants in the contexts **VSCode** and **In Tutorial**. From these graphs, it's possible to see that participants from both groups spent similar amounts of time in these contexts. In order to confirm that the difference observed is not significant, two-tailed Mann-Whitney U tests were performed, with the **null hypothesis** that both groups belong to the same distribution and the **alternative hypothesis** that the groups show a statistically significant difference in results.

Table 7.5: Mean, Standard Deviation and Mann-Whitney U for project questions regarding the time spent in VSCode and in the instructions document. Times are in the format *mm:ss*.

Context	Task	Control		Experimental		Two-Tailed Mann-Whitney U	
		$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
VSCode	Task 1	**05:02**	02:00	**06:11**	02:04	34	**0.24**
	Task 2	**08:55**	02:07	**06:55**	02:23	72	**0.1**
	Task 3	**06:25**	02:36	**09:01**	02:18	24	**0.054**
Instructions	Task 1	**00:48**	00:32	**00:54**	00:35	50.5	**1**
	Task 2	**00:33**	00:13	**00:55**	00:36	34.5	**0.26**
	Task 3	**00:26**	00:26	**00:39**	00:31	35	**0.27**

Table 7.5 shows the result of these tests, as well as the mean and standard deviation of the observations collected from both groups. The Mann-Whitney U test, considering a confidence level of 95%, shows that there were no significant differences between the two groups, since for all tests $\rho > 0.05$. However, with a slightly lower confidence level of 90%, the test shows that participants from the experimental group spent more time in VSCode than participants from the control group in Tasks 2 and 3. It's feasible to hypothesize that some participants from the experimental group spent slightly more time inside Visual Studio Code when compared to participants from the control group due to the fact that the experimental group had more features and feedback in VSCode. Similarly, some participants in the experimental group spent a superior amount of time reading the tutorial possibly due to the fact that they were using an environment with a set of features which they had never used before, meaning that they had to spend some time exploring and experimenting with the features that they had at their disposal. This hypothesis is further encouraged by the fact that median values for the time spent by the participants in the experimental group in the tutorial document slightly decreased from each task

to the next, becoming progressively closer to the median of the participants from the control group, leading to believe that participants were progressively getting more familiar with the environment during the experiment.

Figure 7.4 also shows a *boxplot* representation of the time spent by the participants in each of the groups in the **Terminal** and **Web Browser** contexts. These graphs show that participants from the experimental group tend to spend less time than the control group in these two contexts. In order to verify if this difference is statistically significant, a one-tailed Mann-Whitney U test was performed. The **null hypothesis** for this test is that the results from both groups belong to the same distribution, while the **alternative hypothesis** is that the experimental group shows significantly lower results.

Table 7.6: Mean, Standard Deviation and Mann-Whitney U for project questions regarding the time spent in the terminal and in the web browser. Times are in the format *mm:ss*.

Context	Task	Control		Experimental		One-Tailed Mann-Whitney U	
		$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Terminal	Task 1	**06:20**	02:06	**00:02**	00:04	100	**< 0.01**
	Task 2	**06:34**	01:47	**00:01**	00:01	100	**< 0.01**
	Task 3	**06:13**	01:45	**00:00**	00:00	100	**< 0.01**
Web Browser	Task 1	**02:22**	01:26	**00:17**	00:18	85	**< 0.01**
	Task 2	**03:32**	00:54	**01:16**	01:26	88	**< 0.01**
	Task 3	**03:14**	01:37	**00:37**	00:37	91	**< 0.01**

Table 7.6 shows the results of these tests for the **Terminal** and **Web Browser** contexts. Considering these results and a confidence level of 95%, since $\rho < 0.05$ in all of the Mann-Whitney U tests, it's possible to confirm that the experimental group spent significantly less time than the control group in the **Terminal** and **Web Browser** contexts in every tasks. This discrepancy may be due to the fact that the live feedback and the automatic building and instantiation of the container in the experimental environment drastically reduced the need to use the terminal in order to solve the tasks. Along the same line, if the participants from the experimental group didn't need to use command line tools, then they also had no need to consult documentation for those tools, resulting in participants from the experimental group also spending less time in the web browser.

In summary, and helping to answer the research question RQ1, these results show evidence that developers with access to live dynamic feedback in their environment can tend to spend less time in the terminal and have less need to consult resources using a web browser. From the data displayed in this section, we can also see that the differences regarding the total task time between the two groups, demonstrated in Section 7.7.4, largely come from the experimental group spending less time in the terminal and in a web browser. Finally, we can also see that the increased liveness and reduced time spent in those two contexts doesn't significantly affect the time spent in other contexts.

7.7.6 Context Switches

This section attempts to help answering the research question RQ1, defined in Section 7.2, by comparing the number of context switches that participants perform during the execution of the tasks. In order to understand if one of the groups tends to change contexts more times than the other in the same time interval, we calculate the number of context switches per minute by dividing the number of context switches performed by each participant by the minutes that each participant took to finish a task.

Figure 7.5 presents a *boxplot* graph comparing the context switches per minute for both groups. From the observation of this graph, it's possible to see that there is a tendency for the experimental group to perform less context switches per minute spent in all tasks.

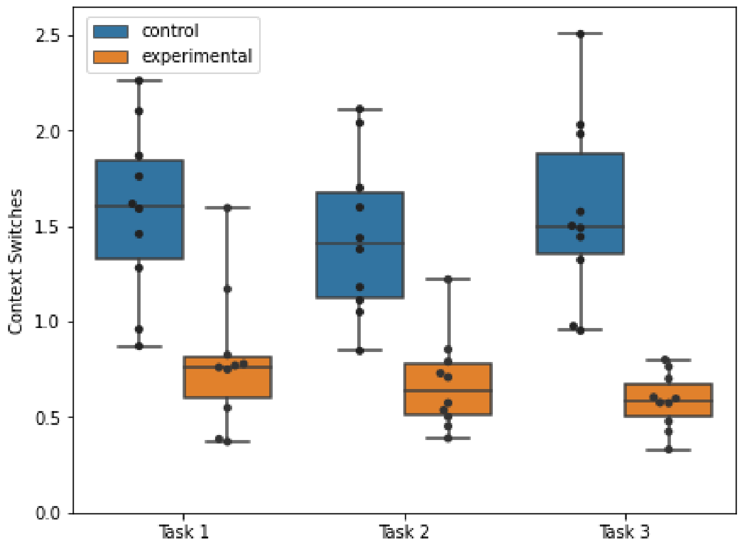


Figure 7.5: Context switches per minute in each task, separated by group.

In order to verify if this decrease is significant, a one-tailed Mann-Whitney U test was performed. The **null hypothesis** for this test is that the results from both groups belong to the same distribution, while the **alternative hypothesis** is that the experimental group shows significantly lower results.

Table 7.7: Mean, Standard Deviation and Mann-Whitney U for the number of context switches per minute by participants in each of the tasks.

Task	Control		Experimental		One-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Task 1	**1.58**	0.43	**0.79**	0.35	93	**< 0.01**
Task 2	**1.44**	0.40	**0.67**	0.23	95	**< 0.01**
Task 3	**1.58**	0.46	**0.58**	0.14	100	**< 0.01**

Table 7.7 shows the results of these tests. Considering these results and a confidence level of 95%, since $\rho < 0.05$ in all of the Mann-Whitney U tests, it's possible to confirm that the experimental group performed significantly less context switches per minute in each of the tasks, when compared to the control group. Helping understand the way that developers spend time with liveness in their environment, this data reveals that one of the consequences of having increased live feedback while developing Dockerfiles is that developers tend to switch between contexts less times.

Figure 7.6 shows the total number of context switches performed to each context (i.e. for the context **VSCode**, shows the number of times that the participants switched to the **VSCode** context in the three tasks). It's noticeable from this graph that the users from the control group switched between the **VSCode** and **Terminal** contexts many more times than the experimental group. Looking at these results, we consider that the constant need to travel between the **VSCode** and **Terminal** can be disruptive and generally negative for the development experience.

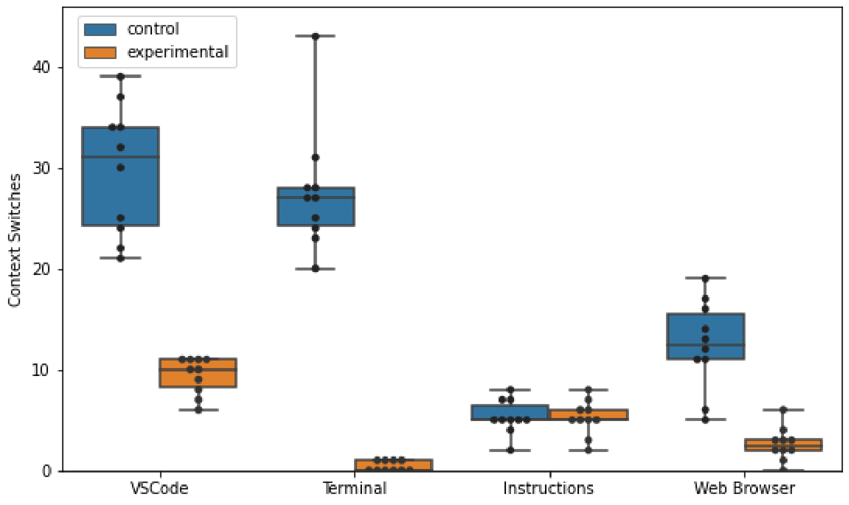


Figure 7.6: Total context switches to each context, separated by group.

Furthermore, from the results presented in this section and the total time that participants take to finish the tasks, discussed in Section 7.7.4, we can see that participants from the experimental group generally take less time to finish the tasks and perform less context switches per minute. We believe that the reduced need to switch from one context to another may be one of the factors that contributes to the decrease in the time required to finish a task.

7.7.7 Feature Usage

Participants from both groups were asked to indicate which features they used during the execution of each task. We analyse this data in order to discover which features were most used during the experiment. This can help validate the feature choices that were made during the design of the approach, described in Section 6.1, and help direct future development efforts. Participants were able to choose from a list of the features that directly matched the features described in the instructions document or input any text using a text field. They could also choose the option 'None' if they felt that no feature in particular had been helpful in that task.

Figure 7.7 presents the feature usage reported by the control group. The control group, which had substantially less features at their disposal, frequently felt that there wasn't a single feature that had been helpful in the execution of the task. In fact, from the total of 30 tasks that were executed by the 10 participants in this group, in 11 of those tasks the participant reported that none of the available features was helpful. Furthermore, some participants mentioned the terminal and the web search as helpful features. These features do not provide live feedback and the terminal in particular requires manual work from the developer which would not be as needed in a live environment.

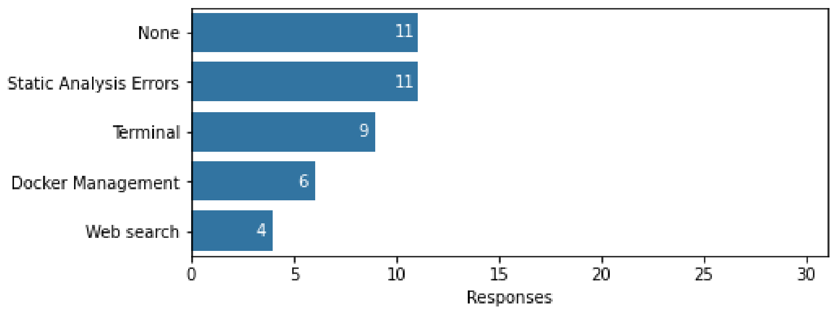


Figure 7.7: Feature usage reported by the control group in all tasks.

Figure 7.8 presents the feature usage reported by the experimental group. Users identified two features in particular as being used in almost every task: *Container log output* (30/30 tasks) and *Image Build and Container Runtime Errors* (28/30 tasks). This is probably due to the fact that building an image and a container and inspecting the container's standard output are steps that may need to be performed very often while debugging a Dockerfile. As such, continuously raising a container and continuously displaying the container's standard output in the IDE without requiring any effort from the developer can become very useful features, as displayed by the graph. Other features can be a bit more situational and, therefore, might have been useful in only a few of the tasks. Still, most of the live features described in the instructions document were reported to be useful in at least one third of the tasks executed. It can also be noted that even with the live dynamic analysis features, static analysis features were still perceived as useful, being used in 14 of the 30 total tasks, reinforcing the idea presented in Section 6.1 that live dynamic analysis features are not necessarily a replacement for live static analysis features, but rather a complement. Finally, it can be noticed that the features "Open shell in container" and "Docker management" weren't used by any participant. For the first feature, this can be due to the fact that participants from the experimental group generally didn't feel the need to access the terminal, as shown in Section 7.7.5. The lack of usage of the second feature can be due to the fact that since Dockerlive automatically builds and instantiates a container as the Dockerfile is being edited, participants from the experimental group felt no need to perform actions related to the management of Docker artifacts.

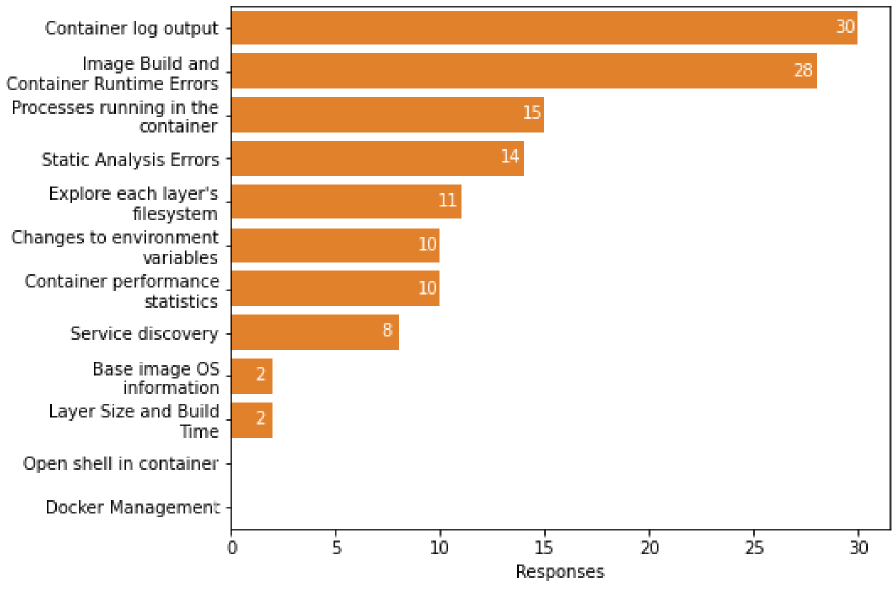


Figure 7.8: Feature usage reported by the experimental group in all tasks.

7.7.8 Dockerfile Development Activity Improvement

In order to understand how the presence of live feedback helps in each of these activities, particularly in the most time-consuming ones, participants answered the following Likert scale questions, which directly address the development activities identified in the survey described in Chapter 4:

- **Q1** — During the execution of the tasks, the feedback provided in the IDE helped me *finding out what parent image is the most suitable.*
- **Q2** — During the execution of the tasks, the feedback provided in the IDE helped me *finding out what are the dependencies of the system that must be added to the docker image.*
- **Q3** — During the execution of the tasks, the feedback provided in the IDE helped me *finding out what are the Dockerfile commands that I need.*
- **Q4** — During the execution of the tasks, the feedback provided in the IDE helped me *trying to understand if the resulting container is working as intended (e.g., running commands and tests on the container).*
- **Q5** — During the execution of the tasks, the feedback provided in the IDE helped me *trying to understand why the resulting container is not working as intended.*
- **Q6** — During the execution of the tasks, the feedback provided in the IDE helped me *finding out which commands are responsible for the container misbehaviour.*
- **Q7** — During the execution of the tasks, the feedback provided in the IDE helped me *rebuilding the image and re-running the container to confirm that it is working as intended.*

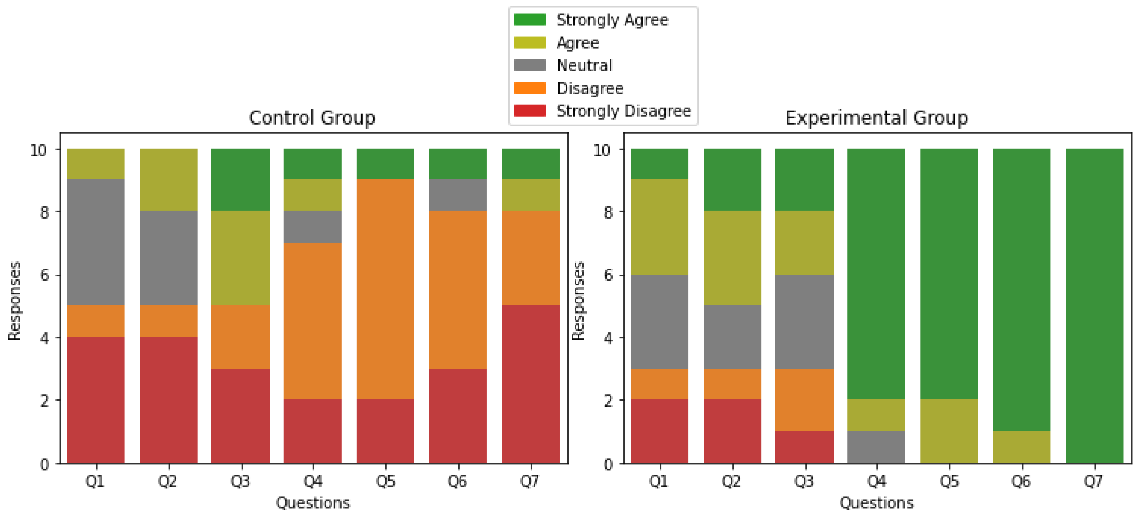


Figure 7.9: Perceived helpfulness of the development environment in each Dockerfile development activity.

Figure 7.9, which presents the answers to the questions Q1-Q7, shows that participants from the experimental group tended to respond more positively to these questions, particularly in questions Q4-Q7. In order to verify if the experimental group provided significantly more positive answers, a one-tailed Mann-Whitney U was performed with the **null hypothesis** that the results from both groups belong to the same distribution and the **alternative hypothesis** that the experimental group shows significantly higher results.

Table 7.8: Mean, Standard Deviation and Mann-Whitney U for Likert scale questions regarding each of the Dockerfile development activities.

Question	Control		Experimental		One-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q1	**2.2**	1.08	**3**	1.26	32	**0.08**
Q2	**2.3**	1.19	**3.2**	1.4	31.5	**0.08**
Q3	**2.9**	1.58	**3.2**	1.25	44.5	**0.35**
Q4	**2.4**	1.2	**4.7**	0.64	8	**< 0.01**
Q5	**2.1**	1.04	**4.8**	0.4	6	**< 0.01**
Q6	**2.1**	1.14	**4.9**	0.3	5.5	**< 0.01**
Q7	**2**	1.34	**5**	0	5	**< 0.01**

According to the results presented in Table 7.8, considering a confidence level of 95%, there isn't a significant difference in the responses to questions Q1, Q2 and Q3, since for those 3 tests $\rho > 0.05$. In other words, participants did not believe that (Q1) the live dynamic feedback significantly helped finding out what parent image is most suitable, (Q2) finding out what are the dependencies of the system that must be added to the docker image or (Q3) finding out the right Dockerfile commands. In Q1, this is can be due to the fact that the participants were asked to edit an existing Dockerfile, which already had a valid base image, and therefore never had to go through the task of choosing a parent image. In Q2, this can be due to the fact that there isn't a feature in Dockerlive dedicated to the management of dependencies, although some of the features available, such as automatically building an image and underlining any Dockerfile instructions that fail, can aid developers in this activity to a certain degree. In Q3, this can be due to the fact that the static analysis extension provides some access to Dockerfile documentation, and since both the experimental group and the control group have access to this feature, there isn't a significant difference between both groups.

However, in the tests for questions Q4, Q5, Q6 and Q7, $\rho < 0.05$. This shows that participants in the experimental group perceived their environment significantly more helpful than the control group in the activities of (Q4) trying to understand if the resulting container is working as intended, (Q5) trying to understand why the resulting container is not working as intended, (Q6) finding out which commands are responsible for the container misbehaviour and (Q7) rebuilding the image and re-running the container to confirm that it is working as intended. This can be due to the fact that the live environment's features, which are available only to the experimental group, directly help the developer in these development activities. It should also be noticed that in the experimental group the mean answer for these questions was very high (between *Agree* (4) and *Strongly Agree* (5)) and the standard deviation was relatively low ($\sigma < 1$), indicating that these participants consistently rated very highly the help provided by their environment in these development activities.

In Chapter 4, the Dockerfile development process was analysed and, by inquiring the opinion of experienced developers using a survey, the tasks where developers feel like they spend the most time were identified. According to the answers to that survey, presented in Section 4.4, the development activities targeted with questions Q4, Q5, Q6 and Q7 were some of the activities that were generally identified as most time consuming. Therefore, the results presented in this section in combination with the results of the Dockerfile development survey indicate that live dynamic analysis feedback can significantly help developers during some of the most time-consuming activities of Dockerfile development. This also helps us answer the research question RQ2, defined in Section 7.2.

7.7.9 Usefulness and Usability

In order to assess the perceived usefulness and usability, helping to answer the research question RQ3 defined in Section 7.2, at the end of experiment each participant responded to the following questions, answered with a Likert scale:

- **Q1.1** — It was easy working with the remote desktop environment.
- **Q2.1** — Having feedback displayed in the IDE (instead of, for example, in an external tool) helped me solve the tasks more quickly.

- **Q2.2** — I found it easy to get all the information I needed to solve the tasks without leaving the IDE.
- **Q2.3** — The feedback provided inside the IDE made it easier to solve the designated tasks.
- **Q3.1** — I felt overwhelmed by the quantity of information displayed inside VSCode.
- **Q3.2** — I felt overwhelmed by the way the information was displayed inside VSCode.

Question Q1.1 isn't directly related to the usefulness and usability of the participant's environment. However, if participants have trouble using the remote environment, it's possible for their performance and their perspective on the usability of the development environment to be negatively affected. For that reason, it's important to verify that the remote environment didn't have a negative impact and was easy to work with, both in the experimental and in the control group. As such, a two-tailed Mann-Whitney U test was performed. The **null hypothesis** for this test is that both groups belong to the same distribution and the **alternative hypothesis** that the groups show a statistically significant difference in results.

Table 7.9: Mean, Standard Deviation and Mann-Whitney U for Likert scale questions regarding the remote environment used during the experiment.

Question	Control		Experimental		Two-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q1.1	**4.4**	0.49	**4.6**	0.66	38	**0.32**

Table 7.9 presents the results obtained with the answers to these questions. Considering a confidence level of 95%, since $\rho > 0.05$, these results show evidence that there isn't a significant difference in the results to Q1.1 from both groups. On average both groups answered between a *Agree* and a *Strongly Agree* with a small standard deviation, indicating that most participants generally had no trouble using the remote environment. This minimizes the chance of the remote environment influencing the rest of the results presented in this chapter.

Questions Q2.1, Q2.2 and Q2.3 aim to test if the experimental group felt more than the control group that the features that they had at their disposal were helpful and useful in the task execution. As such, a one-tailed Mann-Whitney U was performed. The **null hypothesis** for this test is that the results from both groups belong to the same distribution, while the **alternative hypothesis** is that the experimental group shows significantly higher results.

Table 7.10: Mean, Standard Deviation and Mann-Whitney U for Likert scale questions regarding the usefulness of the features provided.

Question	Control		Experimental		One-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q2.1	**3.1**	1.45	**4.9**	0.3	12.5	**< 0.01**
Q2.2	**1.7**	1.19	**4.2**	0.75	8	**< 0.01**
Q2.3	**2.9**	1.37	**4.9**	0.3	3	**< 0.01**

Considering a confidence level of 95%, since $\rho < 0.05$ in all the tests presented in Table 7.10, we can conclude that participants from the experimental group displayed a statistically significant tendency to find their environment more useful and helpful than participants from the control group. These results also help us answer the research question RQ3, showing evidence that users perceive environments with live dynamic analysis features as more helpful and useful than environments with less liveness.

The purpose of the questions Q3.1 and Q3.2 is to verify if the participants' environment, particularly the experimental environment since it has much more information available to the developer. Furthermore, these questions make it possible to verify if the participants perceive one of the environments as more overwhelming than the other. In order to verify if there is a significant difference in the perceived overwhelmingness between both groups of participants, a two-tailed Mann-Whitney U test has been performed. The **null hypothesis** for this test is that both groups belong to the same distribution and the **alternative hypothesis** that the groups show a statistically significant difference in results.

Table 7.11: Mean, Standard Deviation and Mann-Whitney U for Likert scale questions regarding the overwhelmingness of the environment used for the tasks.

Question	Control		Experimental		Two-Tailed Mann-Whitney U	
	$\bar{x}$	σ	$\bar{x}$	σ	U	ρ
Q3.1	**1.7**	0.78	**1.6**	0.66	52.5	**0.87**
Q3.2	**1.8**	0.75	**1.6**	0.66	57	**0.59**

Table 7.11 presents the results obtained with the answers to these questions. Once again, considering a confidence level of 95%, since $\rho > 0.05$ in all the tests we can conclude that both groups presented similar opinions in regards to the overwhelmingness of their environment. In both the control group and experimental group the mean response was low and no answers above "Neutral" in the Likert scale were given.

As described in the analysis of the state-of-the-art approaches to liveness and feedback in IDEs, presented in Section 3.2, one of the pitfalls that live development environments can fall into is to display too much information in a way that is not friendly for the user, making the user feel overwhelmed. The results presented in this section show some evidence that Dockerlive successfully avoids these problems.

7.7.10 Long-text Feedback

Participants were also asked to answer a three free text questions. This section presents a summary of the answers provided to these questions.

Useful features. The first question asked participants what features were most useful in their environment, and why. In the control group, since their environment did not have as many features as the experimental group did, the answers were generally shorter. Four participants mentioned the **quick access to Dockerfile documentation via static analysis**, while other three participants mentioned the **Docker management** feature where they could manually interact with Docker artifacts using a menu in VSCode. The three remaining participants did not name any feature. In the experimental group, 7 participants mentioned the **automatic building and instantiation of the container** as one of the most useful features. Two of these users mentioned that this feature was valuable to them due to the *"instant feedback [that] allows for quicker identification of errors"*. Another participant stated: *"I could see myself using [Container log output and automatic image build] several times throughout my day to day tasks"*. Other features frequently mentioned were the **file system explorer**, the **container performance graphs**, the **service discovery**, **environment variables change detection** and **visualization of processes running in container**. The general reasoning behind theses answers is that these features provide quick ways to inspect the container in a constantly updating way. In general, these results were consistent with the results of the feature usage questions, presented in Section 7.7.7.

Problems with level of feedback. The second question asked participants to identify problems they had working with the level of feedback that they were provided. Participants from the control group mentioned *"Lack of debugging information"*, *"Leaving IDE workflow too often"*, *"Lack of feedback in terms of knowing the origin of the error"*, and *"Took some time finding out what was causing the problem"*. Participants from the experimental group did not mention any of these problems. The problems reported by participants from the control group are very much in line with the problems identified in the Dockerfile development survey presented in Chapter 4, further sustaining our hypothesis that the current Dockerfile development workflow has problems that could be tackled with live dynamic feedback in the developer's environment. In the experimental group, only two participants reported a problem. Both problems were related to their own inexperience and the initial learning curve to Dockerlive. This was expected, since they had no previous experience with Dockerlive.

Improvement suggestions. The third and final question asked participants how they'd improve the level of feedback they had to work with. In the control group, 3 participants expressed that they would like to have features which would give them more feedback on the Dockerfile under development. 3 other participants mentioned that they'd like to have features which would give them a quicker path to perform Docker-related actions. 4 other participants didn't identify any way to improve the level of feedback, one of which even stated that *"The IDE can't think for you,* *the help it provides is already what you can expect as a software developer"*. In general, these responses show that liveness may not be something that developers immediately identify as a potential improvement to their Dockerfile development environment, even though the findings reported in this user study clearly show that it is. In the experimental group, 1 participant mentioned that the image build errors could be more precise and point to the specific part of the instruction that is failing instead of the entire instruction. Another participant mentioned that showing suggestions for common mistakes would be an improvement. 7 other participants expressed that they had no improvements to suggest.

7.7.11 Experience and Total Task Time

This section aims at understanding if the participant's experience with Docker technologies has any impact in the time that participants take to finish the tasks.

Considering the questions asked regarding the participant's experience, analysed in Section 7.7.2, it would be reasonable to expect a relationship between the experience of each participant and the time that they take to execute the tasks within each participant group. In particular, participants who reported having more experience could be expected to be faster than other participants from the same group who reported having less experience.

However, this is not the case. We were unable to find a clear relationship between a participant's responses to the personal experience questions and the participant's performance during the tasks. As an example, Figure 7.10 shows that a clear relationship between the total time spent in Task 3 and the Dockerfiles created/updated by each participant doesn't seem to exist. Similar results were found for the other tasks.

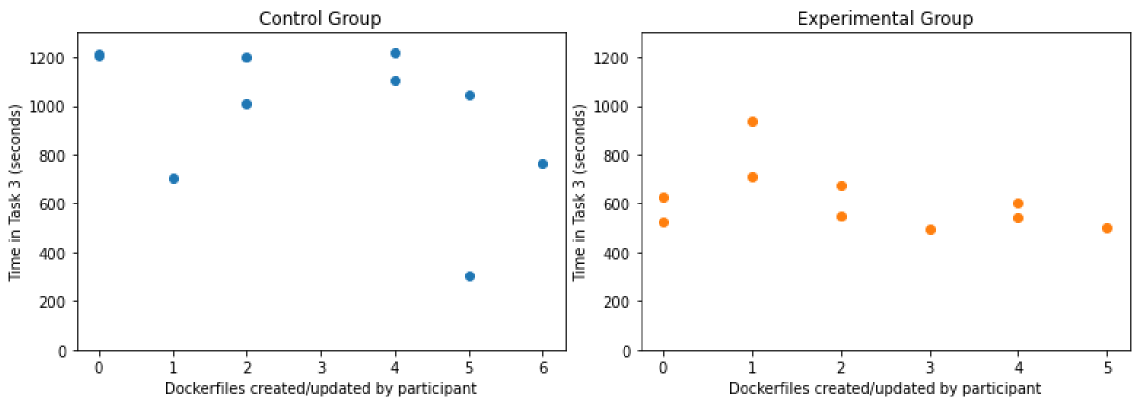


Figure 7.10: Total time spent in Task 3 and number of Dockerfiles edited.

This can be due to the fact that since all participants have very similar backgrounds: almost all the participants are students — or very recent graduates — from the integrated masters in Informatics and Computer Engineering at the Faculty of Engineering of the University of Porto. Therefore, since all participants went through similar learning experiences prior to the study, the number of Dockerfiles created/updated by each participant may not have a very big impact in the participant's performance.

Nonetheless, it was still important to include these questions in order to ensure in the best way possible that the groups have balanced levels of experience when it comes to the technologies used during the user study. At a larger scale, with an increased number of participants, this would become even less of a concern, since the larger the groups, the more they would tend to be balanced by nature.

7.8 Threats to validity

The user study described in this chapter aims to validate the main hypothesis, presented in Section 5.2, and help answer the research questions, presented in Section 7.2. This section discusses the characteristics of this user study which can pose a threat to the validity of the conclusions drawn in this chapter.

Participant's background. All the participant's from this user study have very similar professional backgrounds. Almost all the participants (19/20) are students or very recent graduates from a MSc in Informatics and Computer Engineering at the Faculty of Engineering of the University of Porto. As such, it may not be completely safe to generalize the conclusions drawn in this user study to a wider audience, namely an audience with strong professional experience in programming or Docker technologies. However, the methodology used in this user study should be viable and produce valuable results if executed with more experienced participants. As such, it would be interesting, as future work, to perform a run of this user study with more experienced participants and even to compare the results with the results presented in this chapter.

Sample size. The sample size for this user study could be considered relatively small. A larger sample size would have allowed to perform a completely random grouping of participants, since the higher the number of participants, the more naturally balanced randomly generated groups can become. However, given the very significant disparity of results from the control and experimental group, evidenced in Section 7.7, we would not expect a significant difference in the conclusions drawn from this chapter compared to the conclusions drawn a user study performed with a larger set of participants.

Control group environment. The control group's environment was designed to be as close to the "typical" Dockerfile development environment as possible. Therefore the control groups environment is the combination of two tools: The Visual Studio Code IDE, which was the most popular development environment amongst DevOps developers in 2019 [4], and the *Docker for Visual Studio Code (Preview)* [5] extension, which was the most commonly used extension during

[4]Stack Overlow Developer Survey 2019 available at `https://insights.stackoverflow.com/survey/2019`

[5]Links to the mentioned tools can be found in Appendix A

Dockerfile development according to the survey described in Chapter 4. However, there is a concern that other Dockerfile development tools, such as the ones presented in Chapter 3, might benefit the participants in ways that this environment didn't. Nevertheless, a baseline environment had to be chosen and it'd be impossible to account for the favourite development tool of each participant, so we chose the set of tools that, to the best of our knowledge, is most commonly used for Dockerfile development nowadays.

Task complexity. Since each participant only had a limited amount of time to spend in this user study, the tasks proposed couldn't be very complex, as it would take too long for the participants to solve them. As such, instead of obtaining broken Dockerfiles from public open-source projects and creating the tasks around fixing them, we manually created small Dockerfiles with mistakes which we believe to be feasible to happen in a Dockerfile and, at the same time, are small and simple enough that they can be understood and fixed in a short amount of time. It would be interesting, as future work, to perform a run of this user study with Dockerfiles extracted from real projects. As stated before, this would imply recruiting participants willing to perform the tasks during an extended amount of time, who may not be easy to achieve.

Task generalization. Still related to the participants' time constraints, the tasks consisted of fixing existing Dockerfiles and never creating a Dockerfile from scratch. It could be argued that since the tasks only focus on fixing existing Dockerfiles, the impact of liveness hasn't been measured in the full Docker development process. However, we believe that the impact of liveness verified in this user study would translate in similar ways to development tasks where developers have to create a Dockerfile from scratch. It would be interesting, as future work, to perform a run of this user study where participants are asked to fully create a Dockerfile from scratch with the goal of replicating a certain container behaviour. Performing user studies with real-world software projects would also help mitigate this threat.

Remote environment. As detailed in Section 7.3, participants were asked to perform the tasks in a remote environment. There is a risk that the interactions with the development environment could be hindered by the fact that participants are using a remote environment. This could affect the usability of the environment when compared to the most common real-world usage, where a developer uses the IDE on its own computer. In order to ensure, to the best of our abilities, that this factor has as little impact as possible, participants were asked about how easy it was to work with the remote environment. The answers to this question, detailed in Section 7.7.9, show that, according to the participants, the remote environment was easy to work with, giving confidence that the impact of the remote environment in the results obtained was reduced.

Peer pressure. During the experiment, the observing researcher continuously monitored the progress of the participants through the experiment. Since the increased pressure and stress from being under observation can have a significant impact on the performance of developers [3], there's a chance that the performance of the participants was slightly different from their performance if they were doing the tasks without the added pressure of being monitored. However, it should also be noted that since both groups performed the experiment under the same conditions, both groups can equally be affected by this issue.

Participant bias. In controlled experiments, such as the one described in this chapter, where participants are questioned about a novel system, participants may artificially favour the system they believe to be designed by the researcher [21]. In order to mitigate this issue, participants were not informed about their group until after the experiment was over. Nonetheless, it's still possible that some participants realized throughout the experiment that they were using a novel system. However, we believe this risk to be small since, as stated, participants were not informed about their group. In addition, since the control group also had access to a VSCode extension for Docker development, it's possible that some participants from the control group also thought to be using a novel system. As such, we believe that this threat had low potential for impact and equally affected both groups and, therefore, didn't significantly skew the results obtained. Furthermore, we also believe that measurements performed by the researching observer, such as the time spent in each task, are not affected by this threat.

7.9 Main Findings

In summary, this user study provided the following main findings:

- Participants with a live development environment were significantly more efficient at solving the proposed tasks than participants without a live development environment.
- Participants with a live development environment had a more continuous development flow since they did not switch contexts as often as participants without a live development environment.
- Participants with a live development environment, in most cases, did not use a terminal at all and spent less time searching documentation and help in a web browser.
- The live features available to participants with a live development environment were generally highly regarded and perceived as useful and helpful.
- Participants with a live development environment did not feel overwhelmed by the increased live feedback available to them.
- Participants with a live development environment perceived the features available to them as helpful in most Dockerfile development activities, particularly in the ones that were identified in the Docker development survey presented in Chapter 4 to be the most time-consuming.

These findings support the main hypothesis of this dissertation, defined in Section 5.2 — the presence of live dynamic feedback can have a significant and positive impact in the efficiency of the Dockerfile development process.

These findings also help us answer the research questions defined in Section 7.2. RQ1 questions the effect of live dynamic feedback on the way that developers spend time during Dockerfile development. These findings show evidence that in an environment with live dynamic feedback developers tend to be more efficient, tend to spend less time in the terminal and in the web browser and tend to switch contexts less often.

RQ2 inquires about the effect of live dynamic feedback in the Dockerfile development activities identified in Chapter 4. These findings show evidence that developers regard the live dynamic feedback as particularly helpful in the activities that were found to be the some of the most time-consuming by the survey presented in Chapter 4. This reinforces our belief that the approach to generate live dynamic feedback described and implemented in Chapter 6 tackles the most critical activities in the Dockerfile development process.

Finally, RQ3 questions the way that developers perceive an environment with live dynamic feedback in regards to its helpfulness and overwhelmingness. The findings presented in this section show evidence that developers were comfortable using the live dynamic environment and were not overwhelmed by the information displayed in the IDE.

Chapter 8

Conclusion

This chapter presents a summary of the work and contributions developed in this dissertation. As such, Section 8.1 starts by presenting an overview of the problem that this dissertation attempts to tackle, the main hypothesis, the validation methodology and the main conclusions drawn. Furthermore, Section 8.2 presents the main contributions present in this dissertation while Section 8.3 presents some potential expansion points which can be used to expand this dissertation as future work.

8.1 Overview

In recent years, Docker has become one of the most popular containerization technologies, allowing developers to deliver their products in a **secure**, **portable** and **efficient** way. This allows DevOps teams to develop their infrastructure in an agile way and using version control techniques [32]. However, the current development process of a Dockerfile can still be improved.

As discussed in Section 3.1, there is currently very little live feedback available to a developer working with Dockerfiles, as only some of the static analysis tools provide a level of liveness above 3. To the best of our knowledge, there are no tools which provide live dynamic feedback in the Dockerfile development environment. On the other hand, Section 3.2 analyses multiple environments where the presence of live feedback, when correctly implemented, can improve the developer's workflow, promoting a better understanding of the system under development and reducing the time required for a developer to fix problems. Live programming environments, in general, can improve the developer's workflow by introducing immediate feedback, facilitating the exploration process that often takes place during development [10].

The results of the survey presented in Chapter 4 show further evidence that the Dockerfile development process could be improved. The respondents generally agreed that a considerable amount of time is spent in most Dockerfile development activities and only a small percentage uses plugins or tools to help during development.

This dissertation's main hypothesis, described in Section 5.2, is that the Dockerfile development process can become more efficient with the introduction of live dynamic analysis feedback in the developer's IDE. In this context, an increase in efficiency is defined as a decrease in the time required to perform a Dockerfile development task (e.g. fixing a bug in a Dockerfile).

In order to validate this hypothesis, an approach which brings live dynamic feedback to the Dockerfile development environment has been designed, as described in Chapter 6. This approach has been implemented as **Dockerlive** [1], an extension for Visual Studio Code.

A controlled experiment with users, described in Chapter 7, has been performed with the goal of observing the impact of the presence of live dynamic feedback in the Dockerfile development environment. One of the findings from this user study is that developers can take significantly less time to perform a Dockerfile development task when using a development environment with live dynamic feedback.

These results show evidence that the presence of live dynamic feedback in the Dockerfile development environment can **significantly increase** the efficiency in its development process, therefore validating the main hypothesis presented in this dissertation.

8.2 Contributions

In order to validate the main hypothesis of this dissertation, described in Section 5.2, this dissertation makes four main contributions. This section focuses on presenting and describing each of those contributions, as well as how they were useful to achieve the dissertation's main goal of validating the main hypothesis.

Preliminary survey. A preliminary survey, answered by students and professionals, focused on understanding the current Dockerfile development process. This survey identifies the most time-consuming Dockerfile development activities as well as the main approaches that developers use to fix issues in Dockerfiles. With this information, a new approach, described in Section 6.1, was designed to specifically tackle the development activities that the participants of this survey regard as particularly time-consuming. Similarly, when analysing the behaviour of the participants of the controlled experiment with users described in Chapter 7, we can specifically analyse the impact of the tool in the steps identified as most time-consuming. The methodology used in this survey and the results obtained are thoroughly documented in Chapter 4.

Approach to increase liveness in Dockerfile development. In order to analyse the impact of live dynamic feedback in the developer's efficiency and validate the main hypothesis, an approach was designed with the aim of providing live dynamic analysis feedback to a developer working on Dockerfiles. This approach allows the developer to have a workflow with shorter and more

[1] Dockerlive's VSCode Marketplace page is available at `https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive`

frequent feedback loops which, according to the hypothesis, could improve the efficiency of the developer. Section 6.1 provides an in-depth description of this analysis.

Reference architecture and implementation. A reference architecture for the previously mentioned approach has been documented and implemented. This reference architecture has the objective of supporting and guiding software developers who wish to implement the approach defined in Section 6.1. This ensures that the aforementioned approach can be efficiently and independently implemented in other contexts. This reference architecture was implemented as **Dockerlive** [2], an extension for Visual Studio Code. This implementation was used during the controlled experiment with users presented in Chapter 7. An in-depth description of this architecture and Dockerlive's implementation details can be found in Chapter 6 provides.

User study. In order to validate the main hypothesis, mentioned in Section 5.2, we have compared the performance of developers working with and without live dynamic feedback. For that purpose, a controlled experiment with users was performed. In this user study, participants were asked to perform three development tasks where a Dockerfile must be edited. Participants were split into 2 groups. One of the groups performed the development tasks using an environment with live dynamic feedback, while the other group only had live static feedback. Every participant was timed, making it possible to measure the impact of the increased live feedback on the efficiency of developers. Further questions were asked to the participants, in order to evaluate the usability of their environment and obtain other individual remarks that could point towards possible flaws in the environment. The methodology and the results obtained in this user study are thoroughly documented in Chapter 7.

8.3 Future Work

Although the main objective of this dissertation has been achieved, there's still work that can be done in order to expand the implementation of liveness and better understand its impact. This section focuses on exploring those possibilities.

Live feedback. Through the approach presented in Section 6.1, live dynamic feedback with the 4th level of liveness has been implemented. We believe that it could be valuable to increase the level of liveness to the 5th or 6th level by introducing live predictive features which generate multiple code changes and suggest those changes to the developer. A participant from the user study described in Chapter 7 suggested the introduction of the 5th level of liveness, further sustaining this belief. Implementing a liveness level above 4 would require at least an extra step when compared to our approach, where the predictive features are executed. It could also be

[2] Dockerlive's VSCode Marketplace page is available at https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive

valuable to increase the amount of live dynamic feedback that is being generated, without increasing the level of liveness, by identifying and probing new interest points.

Other platforms. Although most of the contributions in this dissertation are directed towards Docker technologies, we believe that our approach could also translate well into other infrastructure-as-code platforms, such as Vagrant [3]. A survey targeting the communities of these platforms could be a good starting point to understand which platforms are the most promising candidates to enable our approach.

Improvements on current implementation. During the controlled experiment with users described in Chapter 7, the participants were asked to give suggestions about what could be improved in Dockerlive. The following changes were suggested:

- Automatically detect common mistakes and code smells in the Dockerfile and suggest potential fixes. [5th level of liveness]
- When an error is detected in the `ADD` instruction, underline the specific arguments of the instruction which caused the error to occur. This can also be applied to other instructions such as the `RUN` instruction.

The time that it takes for the live feedback to be generated and displayed in the IDE can affect the experience of liveness [29]. In that sense, reducing the image build time could significantly reduce the time that it takes for feedback to be generated by Dockerlive, potentially improving the liveness experience even further. To this effect, some of the approaches described in Section 3.1.6, which focus on reducing the Docker image build times, could be integrated with Dockerlive in order to enable a shorter feedback loops.

Improve other Dockerfile development activities. According to the results presented in Section 7.7.8, the Dockerfile development activities regarding the debugging phase of Dockerfile development, such as detecting and identifying container misbehaviour or rebuilding a Docker image, were significantly helped by the presence of the live dynamic analysis feedback implemented. Work could be done in order to bring significant benefits to other activities of Dockerfile development, such as selecting an image or managing dependencies.

Further user studies. As mentioned in Section 7.8, it would be valuable to carry out an experiment similar to the one described in Chapter 7 with a few differences. Firstly, in regards to the participants, having a larger sample size where participants have different backgrounds and higher levels of experience would give more confidence about the generalizability of the results to the population of Dockerfile developers at large. In addition, it would be valuable to design tasks based on Dockerfiles extracted from real open-source projects and tasks which consist of

[3]Links to the mentioned tools can be found in Appendix A

creating a Dockerfile from scratch. These tasks would likely be much more time-consuming to execute than the ones described in Section 7.4, but they could enable a broader measurement of the impact of liveness in the Dockerfile development process.

It could also be useful to perform a case study in a real project in the industry where the usage of Dockerlive would be monitored during a considerable amount of time, such as a few months. This would give some insight into any long-term benefits of having live dynamic feedback during the development of Dockerfiles which may not be observable in short experiments, such as the one performed, where users usually do not have the chance to get fully familiar with their environment.

In order to get a deeper understanding of the current state of Dockerfile development, a subject that is approached in Chapter 5, it could be useful to expand on the survey described in Chapter 4. Doing so would allow us to better understand the strategies and approaches that developers tend to use during Dockerfile development, which in turn could help tweak and improve the approach and implementation of liveness described in Chapter 6.

As explored in Section 2.2, there can be multiple motivations for liveness, such as accessibility, comprehension or productivity [30]. This user study focused mainly on analysing the impact of live dynamic feedback in the productivity of the developers. However, it could also be useful to explore the impact in other motivations. This could involve, for example, assessing the level of comprehension of developers while working with and without live dynamic feedback.

References

[1] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *ACM International Conference Proceeding Series*, pages 1–6, New York, USA, 4 2019. Association for Computing Machinery.

[2] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: Introducing infrastructure-as-code. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 497–498. Institute of Electrical and Electronics Engineers Inc., 6 2017.

[3] Mahnaz Behroozi, Chris Parnin, and Titus Barik. Hiring is Broken: What Do Developers Say about Technical Interviews? In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, volume 2019-Octob, pages 15–23. IEEE Computer Society, 10 2019.

[4] Ilias Bergström and Alan F. Blackwell. The practices of programming. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, volume 2016-Novem, pages 190–198. IEEE Computer Society, 11 2016.

[5] David Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 9 2014.

[6] Jens Böttcher and Andreas Steffens. Current State of Testing Infrastructure as Code. *Full-scale Software Engineering / The Art of Software Testing 2017*, pages 13–18, 2017.

[7] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings - International Conference on Software Engineering*, volume 1 of *ICSE '10*, pages 455–464, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Minh Thanh Chung, Nguyen Quang-Hung, Manh Thin Nguyen, and Nam Thoai. Using Docker in high performance computing applications. In *2016 IEEE 6th International Conference on Communications and Electronics, IEEE ICCE 2016*, pages 52–57. Institute of Electrical and Electronics Engineers Inc., 9 2016.

[9] Jurgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *IEEE International Working Conference on Mining Software Repositories*, pages 323–333. IEEE Computer Society, 6 2017.

[10] Davide Della Casa and Guy John. LiveCodeLab 2.0 and its language LiveCodeLang. In *FARM 2014 - Proceedings of the 2014 ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 1–8. Association for Computing Machinery, 2014.

[11] Rajdeep Dua, Vaibhav Kohli, Sriram Patil, and Swapnil Patil. Performance analysis of Union and CoW File Systems with Docker. In *International Conference on Computing, Analytics and Security Trends, CAST 2016*, pages 550–555. Institute of Electrical and Electronics Engineers Inc., 4 2017.

[12] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support PaaS. In *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pages 610–614. Institute of Electrical and Electronics Engineers Inc., 9 2014.

[13] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and DevOps. In *Proceedings - 3rd International Workshop on Release Engineering, RELENG 2015*, page 3. Institute of Electrical and Electronics Engineers Inc., 2015.

[14] 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 2016*, pages 181–195, Santa Clara, CA, 2 2019. USENIX Association.

[15] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container. In *IEEE Symposium on Mass Storage Systems and Technologies*, volume 2019-May, pages 28–37. IEEE, 5 2019.

[16] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of Docker as Edge computing platform. In *ICOS 2015 - 2015 IEEE Conference on Open Systems*, pages 130–135. Institute of Electrical and Electronics Engineers Inc., 1 2016.

[17] Ivar Jacobson, Ian Spence, and Pan-Wei Ng. Agile and SEMAT: Perfect Partners. *Commun. ACM*, 56(11):53–59, 11 2013.

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

[19] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure DevOps. In *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, pages 202–211. Institute of Electrical and Electronics Engineers Inc., 6 2016.

[20] Hyeonsu Kang and Philip J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *UIST 2017 - Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 737–745. Association for Computing Machinery, Inc, 10 2017.

[21] Andrew J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2013.

[22] Jan Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. How live coding affects developers' coding behavior. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 5–8. IEEE Computer Society, 2014.

[23] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: Insights from the practice. In *Proceedings - International Conference on Software Engineering*, pages 1090–1101. IEEE Computer Society, 5 2018.

[24] Yan Li, Bo An, Junming Ma, and Donggang Cao. Comparison between chunk-based and layer-based container image storage approaches: An empirical study. In *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*, pages 197–202. Institute of Electrical and Electronics Engineers Inc., 5 2019.

[25] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[26] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), 3 2014.

[27] Naoki Mizusawa, Yuya Seki, Jian Tao, and Saneyasu Yamaguchi. A Study on I/O Performance in Highly Consolidated Container-Based Virtualized Environment on OverlayFS with Optimized Synchronization. In *Proceedings of the 2020 14th International Conference on Ubiquitous Information Management and Communication, IMCOM 2020*. Institute of Electrical and Electronics Engineers Inc., 1 2020.

[28] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[29] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. How live are live programming systems? Benchmarking the response times of live programming environments. In *ACM International Conference Proceeding Series*, volume 18-July-20, pages 1–8. Association for Computing Machinery, 7 2016.

[30] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming*, 3(1), 7 2018.

[31] Jay Shah and Dushyant Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019*, pages 184–189. Institute of Electrical and Electronics Engineers Inc., 3 2019.

[32] Jay Shah, Dushyant Dubaria, and John Widhalm. A Survey of DevOps tools for Networking. In *2018 9th IEEE Annual Ubiquitous Computing, Electronics and Mobile Communication Conference, UEMCON 2018*, pages 185–188. IEEE, 11 2018.

[33] 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*, PLoP '15, USA, 2015. The Hillside Group.

[34] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a Pattern Language for Engineering Software for the Cloud. In *25th Conference on Pattern Languages of Programs*, PLoP '18, USA, 2018. The Hillside Group.

[35] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering Software for the Cloud. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, SugarLoafPLoP '16, pages 1–8, USA, 2018. The Hillside Group.

[36] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127–139, 1990.

[37] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings*, pages 31–34. IEEE Computer Society, 2013.

[38] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. CNTR: Lightweight OS containers. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, pages 199–212, Boston, MA, 7 2020. USENIX Association.

[39] Jiwei Xu, Yuewen Wu, Zhigang Lu, and Tao Wang. Dockerfile TF smell detection based on dynamic and static analysis methods. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 185–190. IEEE Computer Society, 7 2019.

[40] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. WeScheme: The browser is your programming environment. In *ITiCSE'11 - Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science*, pages 163–167. Association for Computing Machinery, Inc, 2011.

[41] Xiong Zhang and Philip J. Guo. DS.js: Turn any webpage into an example-centric live programming environment for learning data science. In *UIST 2017 - Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 691–702. Association for Computing Machinery, Inc, 10 2017.

[42] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. An Insight into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 138–143. IEEE Computer Society, 6 2018.

Appendix A

Tool Sources

This appendix contains the sources for the tools categorized in Section 3.1.

Tool	Source
AppOptics	https://www.appoptics.com/
cAdvisor	https://github.com/google/cadvisor
Captain	https://getcaptain.co/
Chef InSpec	https://www.inspec.io/
ctop	https://github.com/bcicen/ctop
dive	https://github.com/wagoodman/dive
Docker-Alertd	https://github.com/deltaskelta/docker-alertd
Docker.el	https://github.com/Silex/docker.el
Docker Explorer	https://marketplace.visualstudio.com/items?itemName=formulahendry.docker-explorer
Docker for Visual Studio Code (Preview)	https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker
docker_monitoring_logging_alerting	https://github.com/uschtwill/docker_monitoring_logging_alerting
Dockeron	https://github.com/dockeron/dockeron
Docker Runner	https://marketplace.visualstudio.com/items?itemName=Zim.vsc-docker
DockerSpec	https://github.com/zuazo/dockerspec
Docker Workspace	https://marketplace.visualstudio.com/items?itemName=tiibun.vscode-docker-ws
Docker [Plugin for Intellij IDEA IDE]	https://plugins.jetbrains.com/plugin/7724-docker
Dockstation	https://github.com/DockStation/dockstation https://dockstation.io/
Dozzle	https://github.com/amir20/dozzle
GoInside	https://github.com/iamsoorena/goinside
goss	https://github.com/aelsabbahy/goss

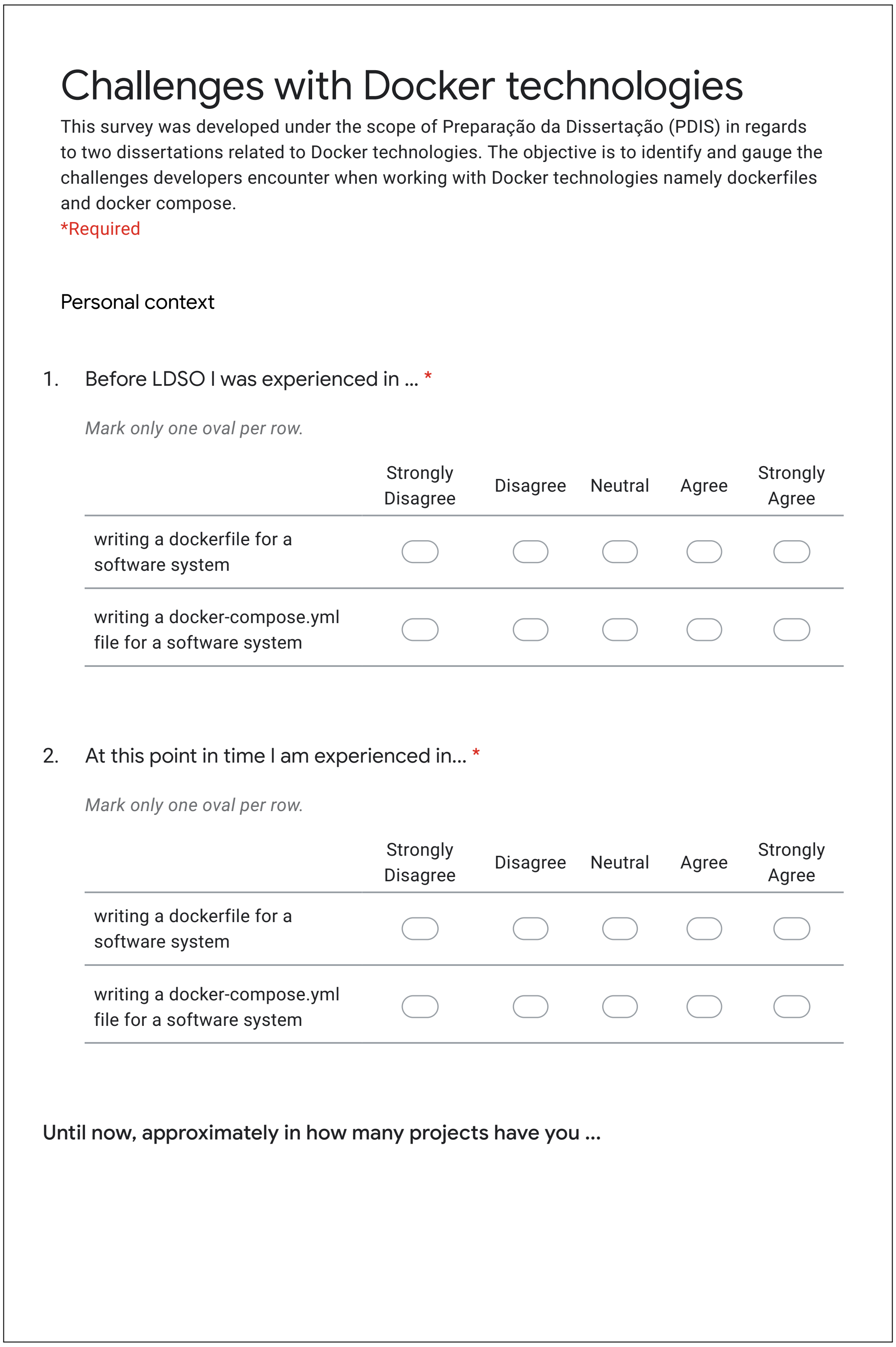
Tool	Source
Hadolint	https://github.com/hadolint/hadolint
Hadolint [Plugin for VSCode]	https://marketplace.visualstudio.com/items?itemName=exiasr.hadolint
Haven	https://github.com/codeabovelab/haven-platform
IntellijIDEA	https://www.jetbrains.com/idea/
lazydocker	https://github.com/jesseduffield/lazydocker
Microsoft Azure	https://azure.microsoft.com/
monit-docker	https://github.com/decryptus/monit-docker
Nmap	https://nmap.org/
Portainer	https://github.com/portainer/portainer https://www.portainer.io/
RSpec	https://rspec.info/
Seagull	https://github.com/tobegit3hub/seagull
ServerSpec	https://serverspec.org
Sysdig	https://sysdig.com/
TeamViewer	https://www.teamviewer.com/
Vagrant	https://www.vagrantup.com/
Visual Studio Code	https://code.visualstudio.com/
Visual Studio Container Tools Extensions (Preview)	https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vs-containers-tools-extensions
Weave Scope	https://github.com/weaveworks/scope
Wharfee	https://github.com/j-bennet/wharfee

Appendix B

Survey

This appendix presents the survey discussed in Chapter 4. Section B.1 contains the survey distributed in the first run, while Section B.2 contains the survey distributed in the second run.

B.1 First Run



Challenges with Docker technologies

This survey was developed under the scope of Preparação da Dissertação (PDIS) in regards to two dissertations related to Docker technologies. The objective is to identify and gauge the challenges developers encounter when working with Docker technologies namely dockerfiles and docker compose.

*Required

Personal context

1. Before LDSO I was experienced in ... *

Mark only one oval per row.

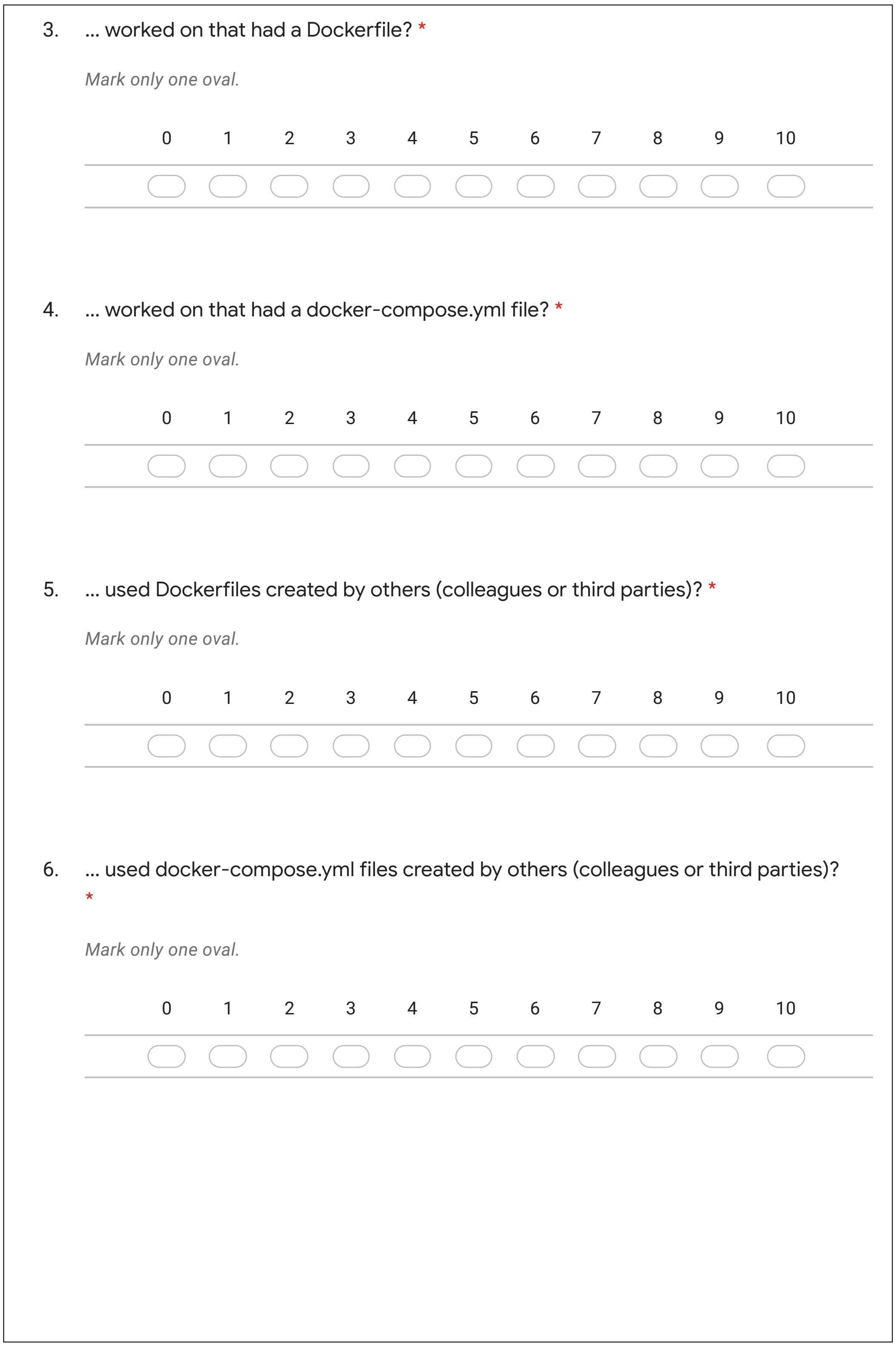
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
writing a dockerfile for a software system	◯	◯	◯	◯	◯
writing a docker-compose.yml file for a software system	◯	◯	◯	◯	◯

2. At this point in time I am experienced in... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
writing a dockerfile for a software system	◯	◯	◯	◯	◯
writing a docker-compose.yml file for a software system	◯	◯	◯	◯	◯

Until now, approximately in how many projects have you ...



3. … worked on that had a Dockerfile? *

Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

4. … worked on that had a docker-compose.yml file? *

Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

5. … used Dockerfiles created by others (colleagues or third parties)? *

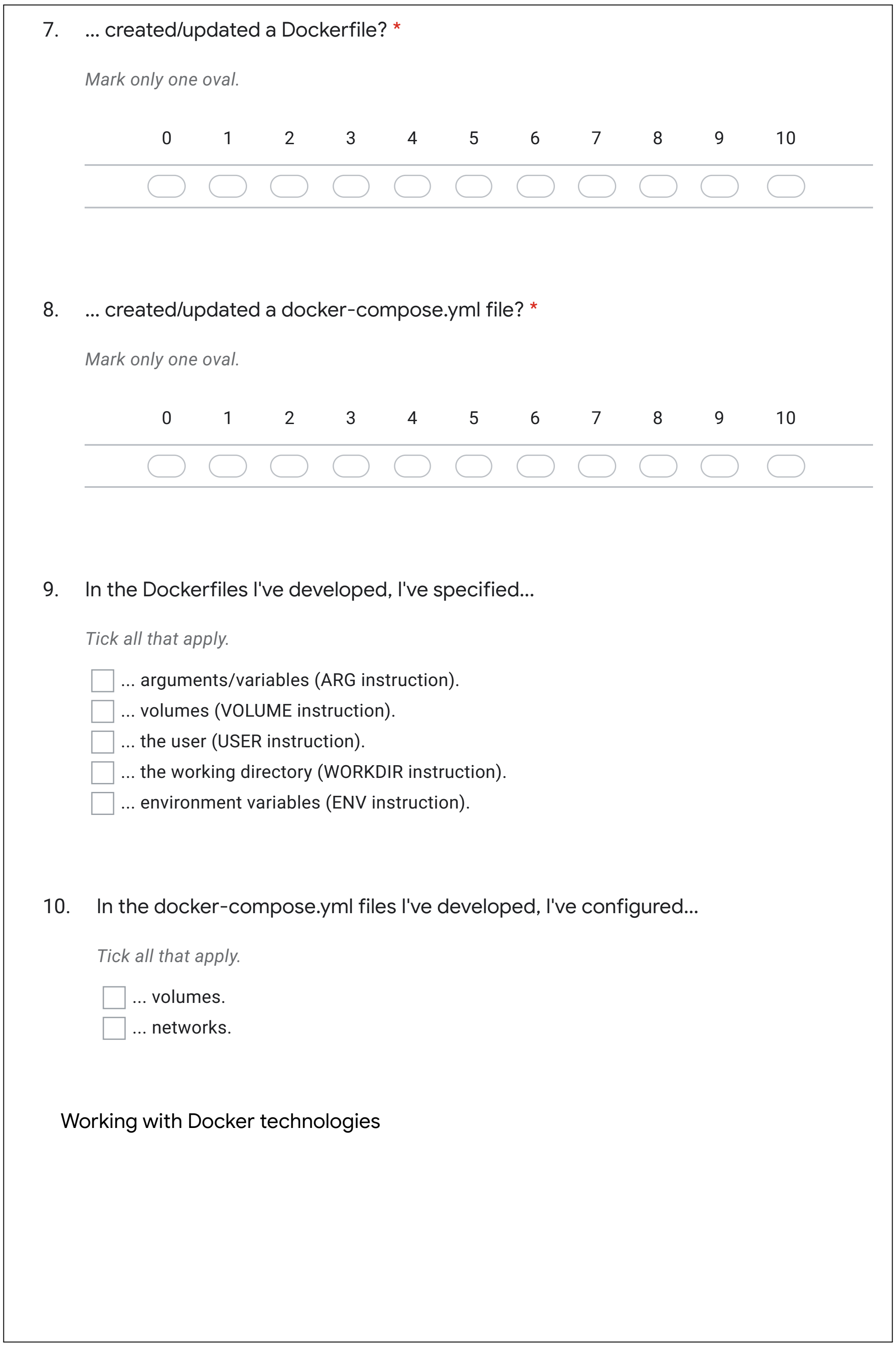
Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

6. … used docker-compose.yml files created by others (colleagues or third parties)? *

Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯



7. ... created/updated a Dockerfile? *

Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

8. ... created/updated a docker-compose.yml file? *

Mark only one oval.

0	1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

9. In the Dockerfiles I've developed, I've specified...

Tick all that apply.

- ☐ ... arguments/variables (ARG instruction).
- ☐ ... volumes (VOLUME instruction).
- ☐ ... the user (USER instruction).
- ☐ ... the working directory (WORKDIR instruction).
- ☐ ... environment variables (ENV instruction).

10. In the docker-compose.yml files I've developed, I've configured...

Tick all that apply.

- ☐ ... volumes.
- ☐ ... networks.

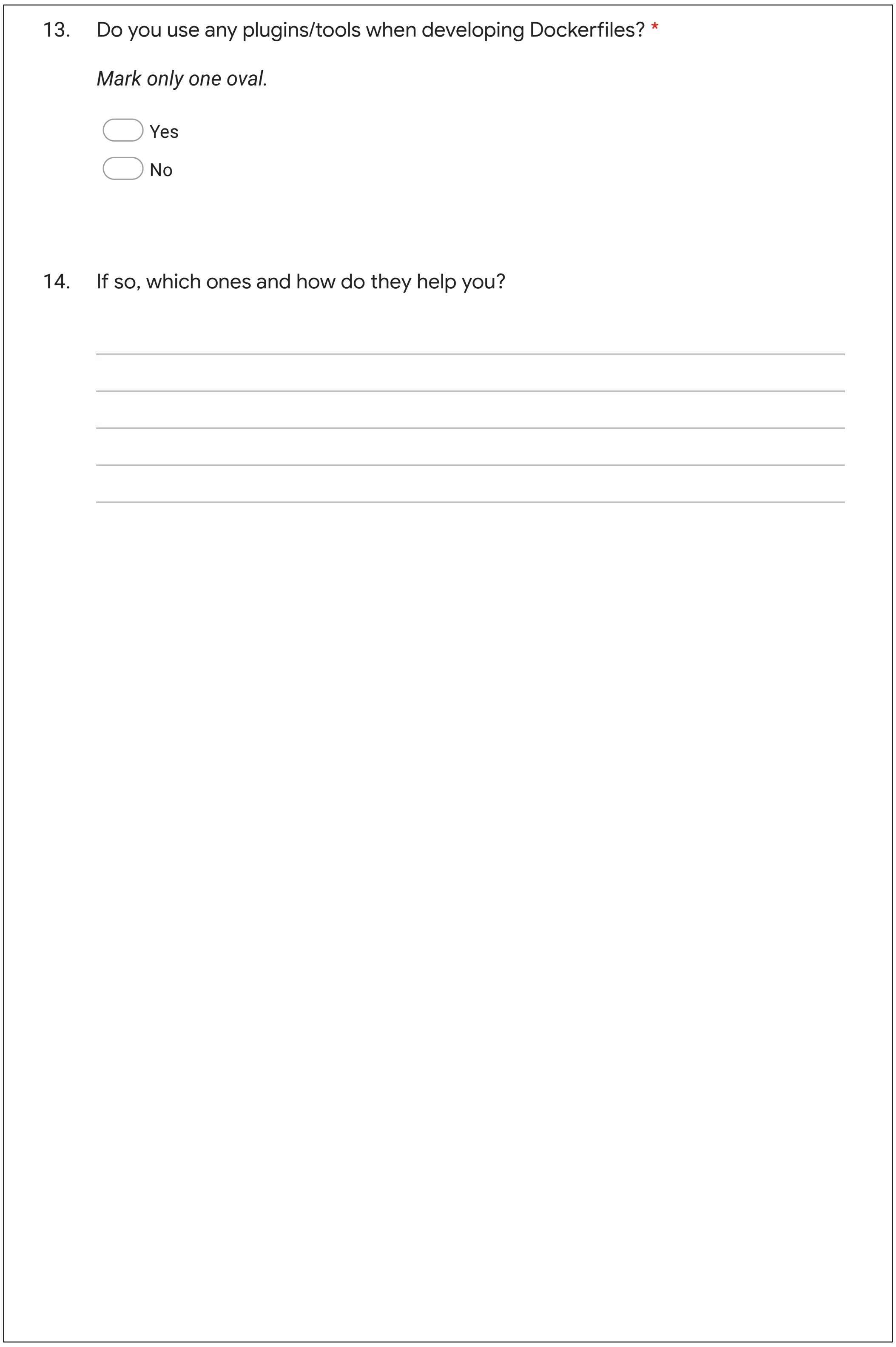
Working with Docker technologies

11. When I write a Dockerfile, I spend a lot of time... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
finding out what parent image is the most suitable.	⬭	⬭	⬭	⬭	⬭
finding out what are the dependencies of the system that must be added to the docker image.	⬭	⬭	⬭	⬭	⬭
finding out what are the Dockerfile commands that I need.	⬭	⬭	⬭	⬭	⬭
trying to understand if the resulting container is working as intended (e.g., running commands and tests on the container).	⬭	⬭	⬭	⬭	⬭
trying to understand why the resulting container is not working as intended.	⬭	⬭	⬭	⬭	⬭
finding out which commands are responsible for the container misbehaviour.	⬭	⬭	⬭	⬭	⬭
rebuilding the image and re-running the container to confirm that it is working as intended.	⬭	⬭	⬭	⬭	⬭

12. What steps or strategies do you usually follow in order to diagnose and fix bugs in the creation of Dockerfiles?



13. Do you use any plugins/tools when developing Dockerfiles? *

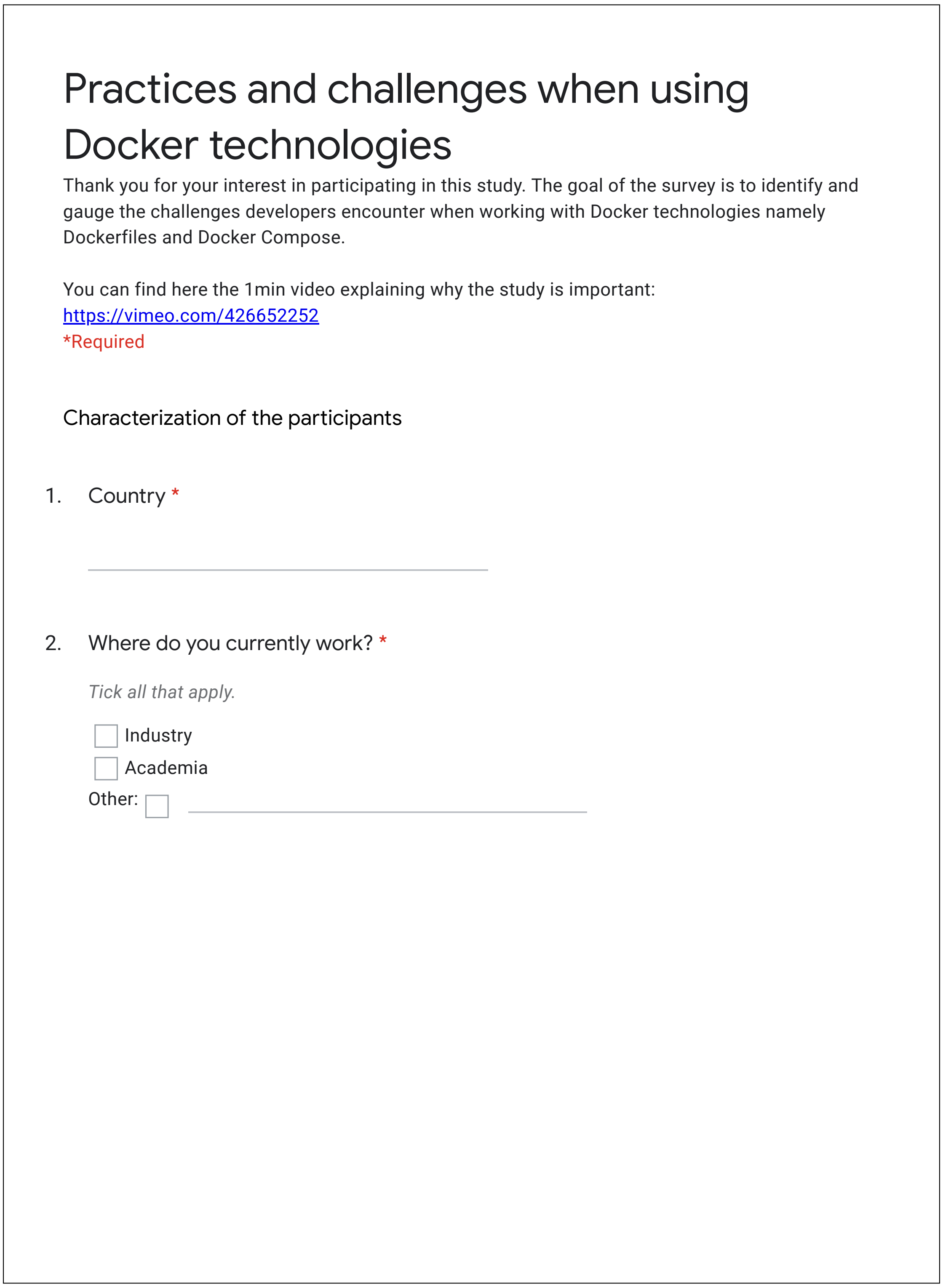
Mark only one oval.

◯ Yes

◯ No

14. If so, which ones and how do they help you?

B.2 Second Run



Practices and challenges when using Docker technologies

Thank you for your interest in participating in this study. The goal of the survey is to identify and gauge the challenges developers encounter when working with Docker technologies namely Dockerfiles and Docker Compose.

You can find here the 1min video explaining why the study is important: https://vimeo.com/426652252

*Required

Characterization of the participants

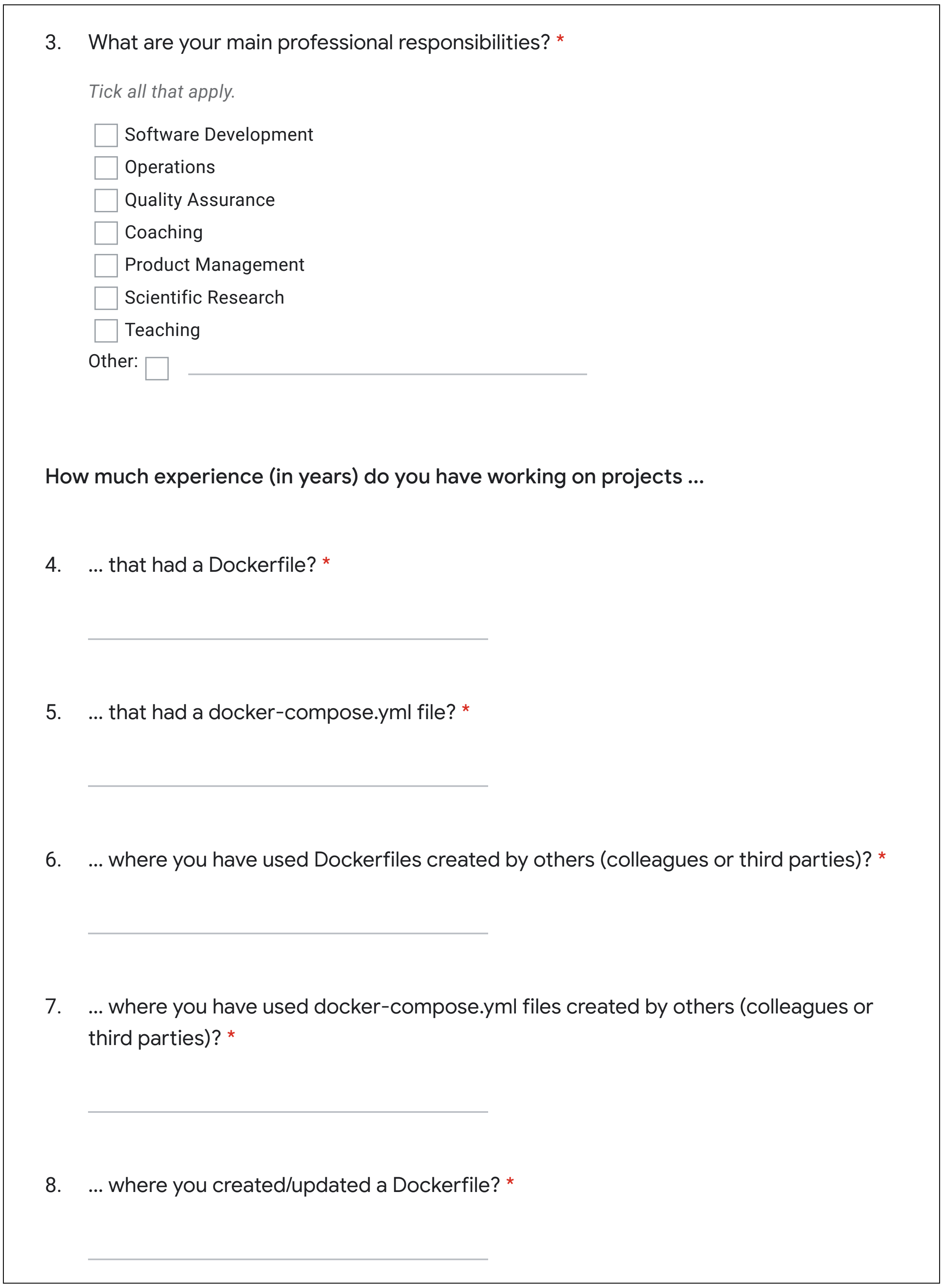
1. Country *

2. Where do you currently work? *

Tick all that apply.

- ☐ Industry
- ☐ Academia

Other: ☐



3. What are your main professional responsibilities? *

   Tick all that apply.

   ☐ Software Development
   ☐ Operations
   ☐ Quality Assurance
   ☐ Coaching
   ☐ Product Management
   ☐ Scientific Research
   ☐ Teaching
   Other: ☐ ____________________

How much experience (in years) do you have working on projects ...

4. ... that had a Dockerfile? *

   ____

5. ... that had a docker-compose.yml file? *

   ____

6. ... where you have used Dockerfiles created by others (colleagues or third parties)? *

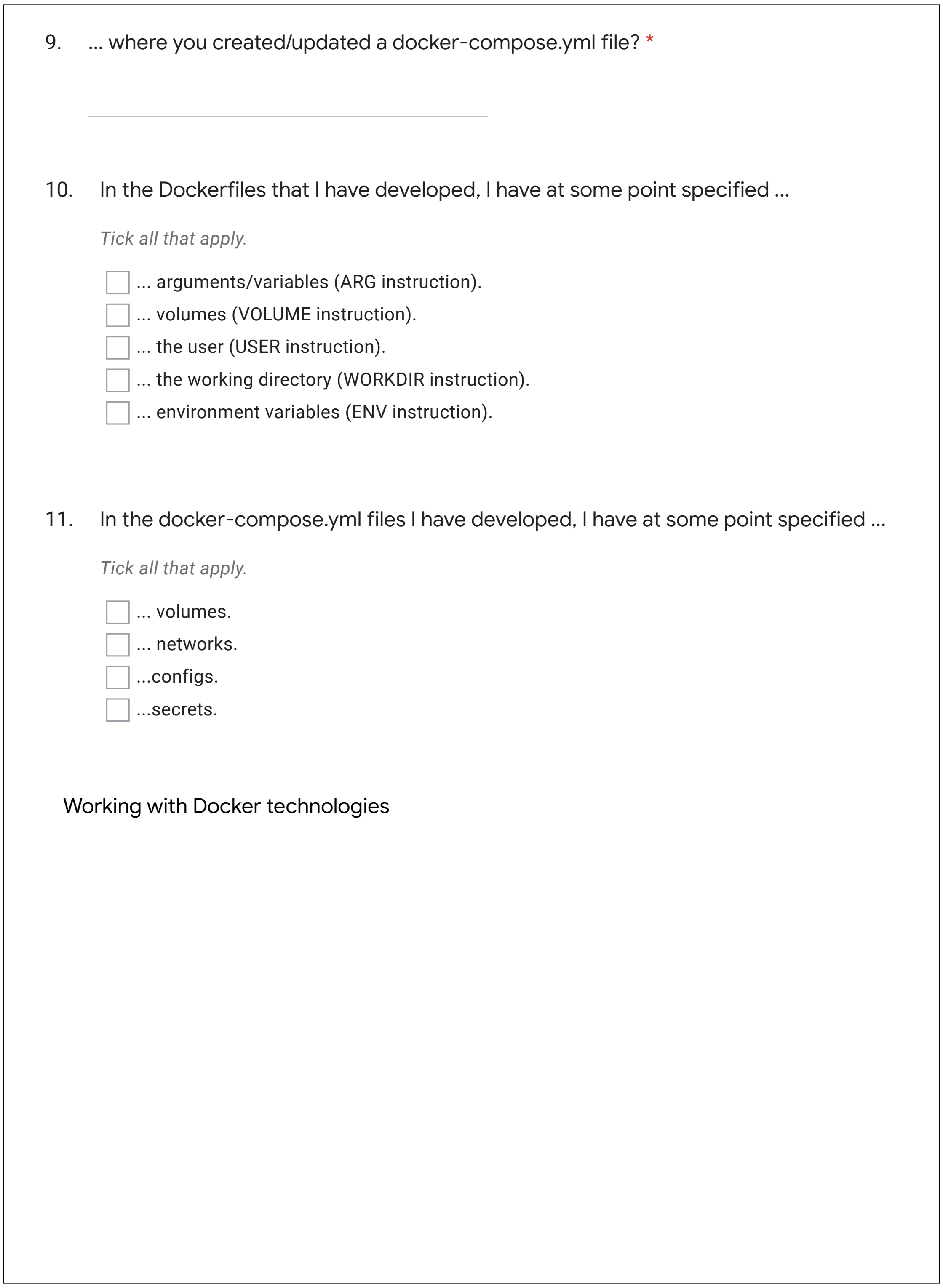
   ____

7. ... where you have used docker-compose.yml files created by others (colleagues or third parties)? *

   ____

8. ... where you created/updated a Dockerfile? *

   ____



9. ... where you created/updated a docker-compose.yml file? *

____

10. In the Dockerfiles that I have developed, I have at some point specified ...

Tick all that apply.

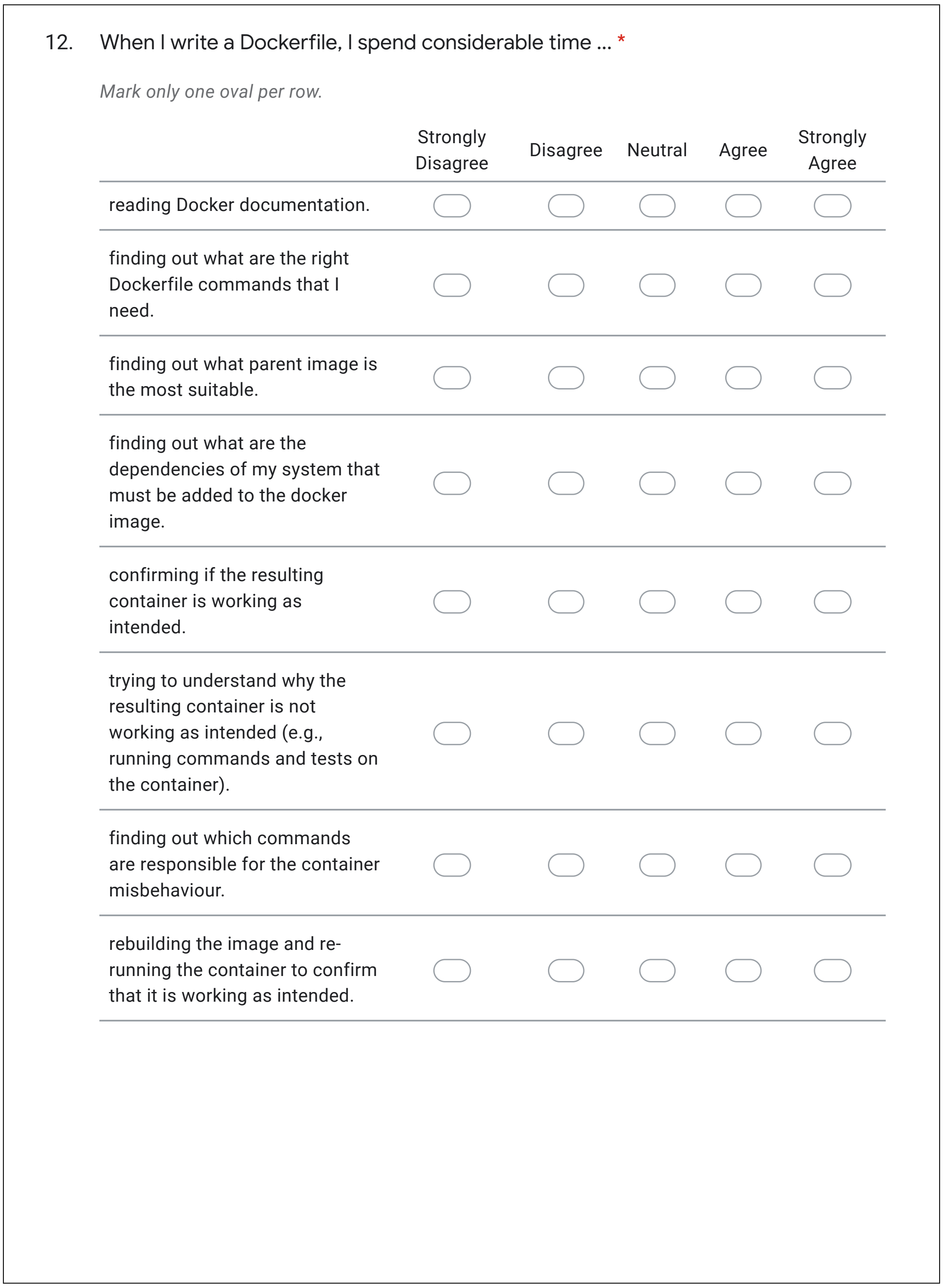
- ☐ ... arguments/variables (ARG instruction).
- ☐ ... volumes (VOLUME instruction).
- ☐ ... the user (USER instruction).
- ☐ ... the working directory (WORKDIR instruction).
- ☐ ... environment variables (ENV instruction).

11. In the docker-compose.yml files I have developed, I have at some point specified ...

Tick all that apply.

- ☐ ... volumes.
- ☐ ... networks.
- ☐ ...configs.
- ☐ ...secrets.

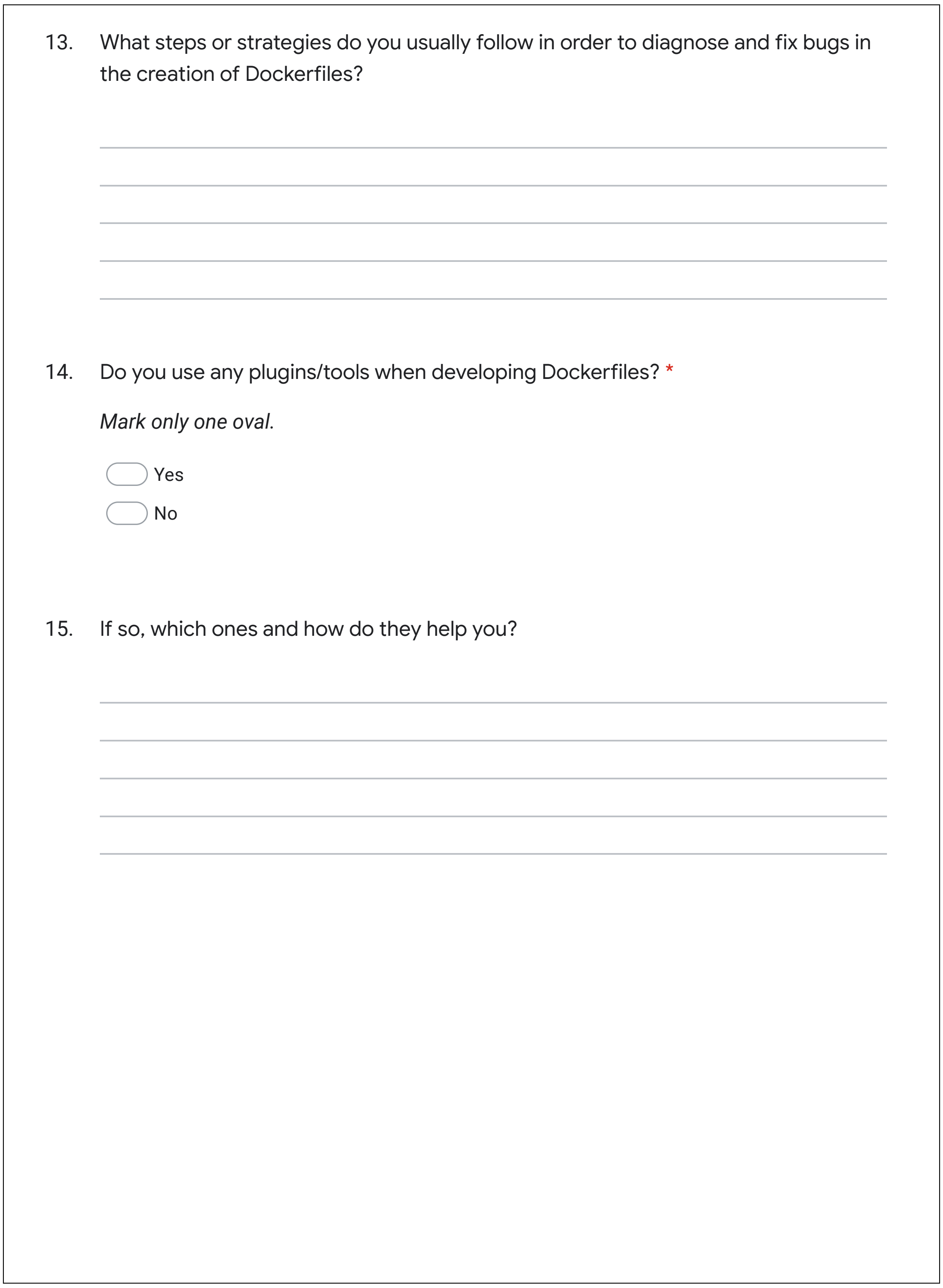
Working with Docker technologies



12. When I write a Dockerfile, I spend considerable time ... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
reading Docker documentation.	○	○	○	○	○
finding out what are the right Dockerfile commands that I need.	○	○	○	○	○
finding out what parent image is the most suitable.	○	○	○	○	○
finding out what are the dependencies of my system that must be added to the docker image.	○	○	○	○	○
confirming if the resulting container is working as intended.	○	○	○	○	○
trying to understand why the resulting container is not working as intended (e.g., running commands and tests on the container).	○	○	○	○	○
finding out which commands are responsible for the container misbehaviour.	○	○	○	○	○
rebuilding the image and re-running the container to confirm that it is working as intended.	○	○	○	○	○



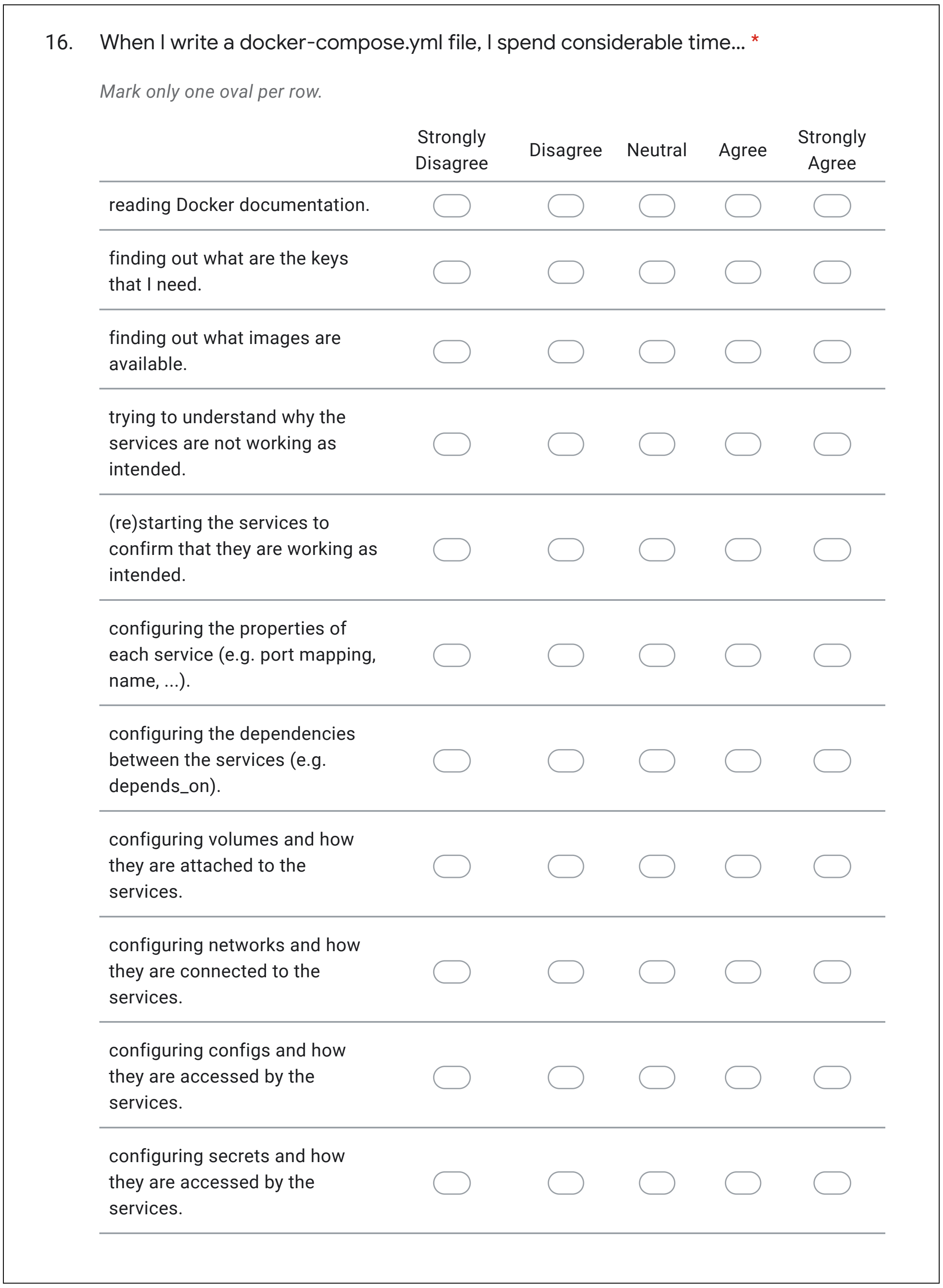
13. What steps or strategies do you usually follow in order to diagnose and fix bugs in the creation of Dockerfiles?

14. Do you use any plugins/tools when developing Dockerfiles? *

Mark only one oval.

- Yes
- No

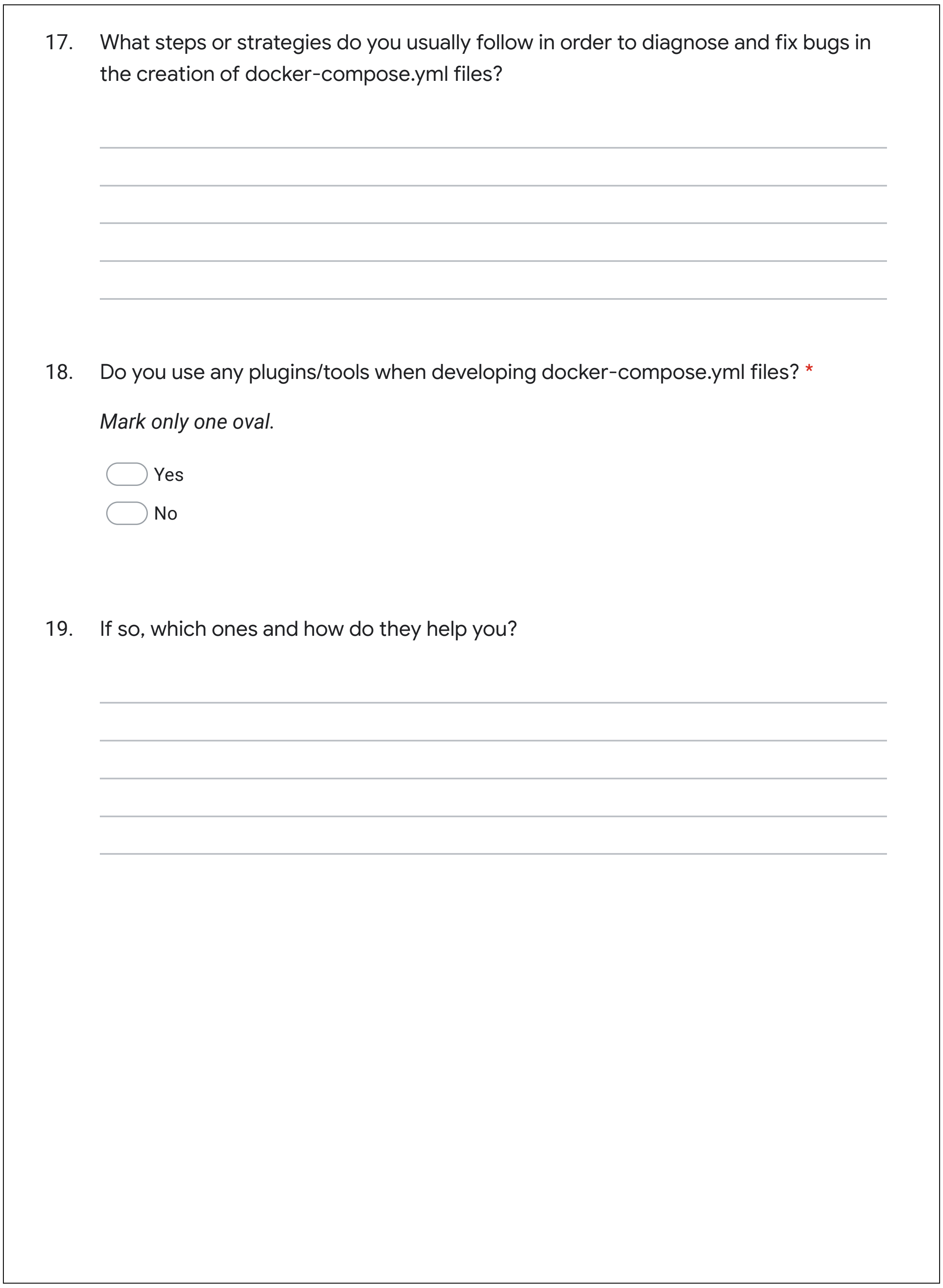
15. If so, which ones and how do they help you?



16. When I write a docker-compose.yml file, I spend considerable time... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
reading Docker documentation.	○	○	○	○	○
finding out what are the keys that I need.	○	○	○	○	○
finding out what images are available.	○	○	○	○	○
trying to understand why the services are not working as intended.	○	○	○	○	○
(re)starting the services to confirm that they are working as intended.	○	○	○	○	○
configuring the properties of each service (e.g. port mapping, name, ...).	○	○	○	○	○
configuring the dependencies between the services (e.g. depends_on).	○	○	○	○	○
configuring volumes and how they are attached to the services.	○	○	○	○	○
configuring networks and how they are connected to the services.	○	○	○	○	○
configuring configs and how they are accessed by the services.	○	○	○	○	○
configuring secrets and how they are accessed by the services.	○	○	○	○	○



17. What steps or strategies do you usually follow in order to diagnose and fix bugs in the creation of docker-compose.yml files?

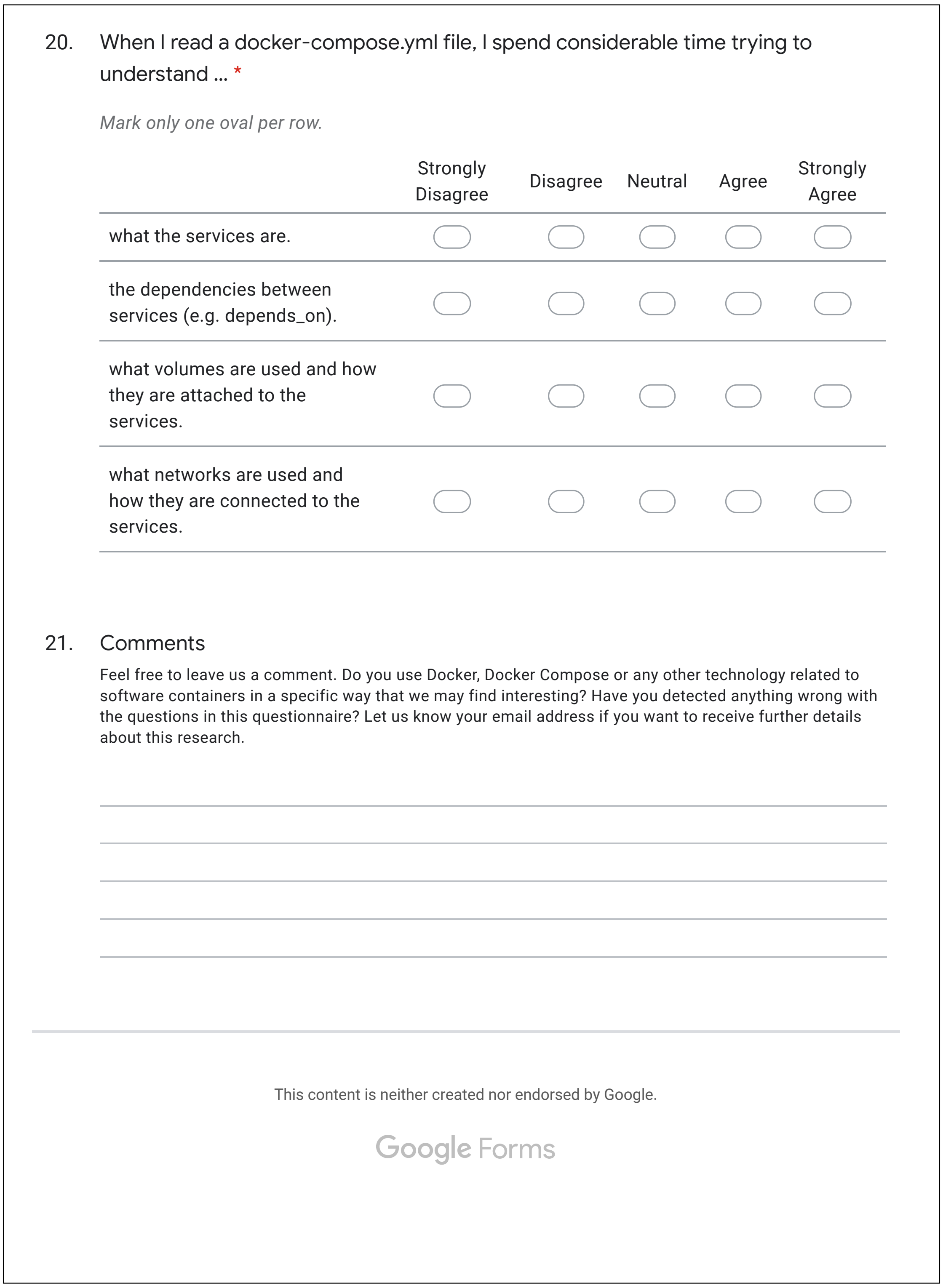
18. Do you use any plugins/tools when developing docker-compose.yml files? *

Mark only one oval.

◯ Yes

◯ No

19. If so, which ones and how do they help you?



20. When I read a docker-compose.yml file, I spend considerable time trying to understand ... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
what the services are.	◯	◯	◯	◯	◯
the dependencies between services (e.g. depends_on).	◯	◯	◯	◯	◯
what volumes are used and how they are attached to the services.	◯	◯	◯	◯	◯
what networks are used and how they are connected to the services.	◯	◯	◯	◯	◯

21. Comments

Feel free to leave us a comment. Do you use Docker, Docker Compose or any other technology related to software containers in a specific way that we may find interesting? Have you detected anything wrong with the questions in this questionnaire? Let us know your email address if you want to receive further details about this research.

Appendix C

User Study Instructions

This appendix presents the instructions document described in Chapter 7. This instructions document was provided to the participants at the start of the experiment. Section C.1 contains the instructions document given to the control group, while Section C.2 contains the instructions document given to the experimental group.

C.1 Control Group

General instructions

Before proceeding, please fill the “Demographic Information” and “Skills and Experience” sections of the questionnaire linked below. Once you reach the “DF1” section, read the rest of this document.

https://forms.gle/ZzZ1V2Gx9FuDhy2DA

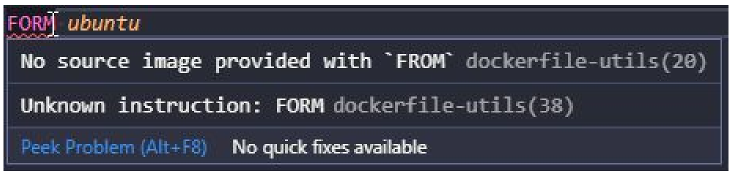
In this experiment you will perform 3 tasks. In each task you must edit a Dockerfile using Visual Studio Code. To help you in these tasks, you’ll have access to a set of extensions which will provide feedback while you edit the Dockerfiles. The combination of Visual Studio Code with these extensions will be referred to as the “environment” in this document and in the questionnaire. The features available in these extensions are described in the next sections of this document.

You have 15 minutes to read and understand the features described in this document. You are allowed to consult this document at any point during the task execution.

Available Features

1- Static Analysis Errors

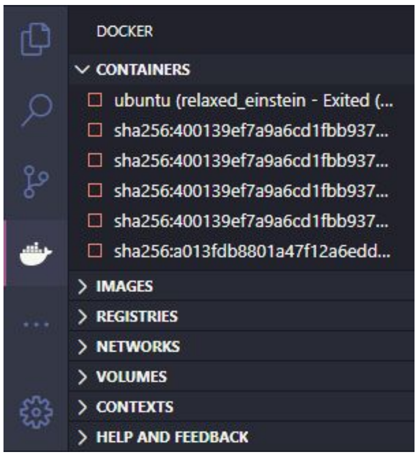
This environment analyses the Dockerfile’s syntax, underlining any syntax errors that it finds.



```
FORM ubuntu
No source image provided with `FROM` dockerfile-utils(20)
Unknown instruction: FORM dockerfile-utils(38)
```

2- Docker Management

This environment provides a sidebar menu which allows the programmer to perform quick actions related to Docker containers, images, registries, networks, volumes and contexts (e.g. start container, stop container, view container logs, delete image).



```
ubuntu (relaxed_einstein - Exited (...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:a013fdb8801a47f12a6edd...
```

Tasks

Global Rules

- In each task, you'll be given a Dockerfile which you must edit until the container has the desired behaviour.
- Each task ends once you notify the experiment observer that you have reached the desired behaviour.
- **You may only edit the Dockerfile**. No other files (such as .py or .js) need to be edited in order to achieve the desired behaviour. However, you are allowed to make temporary changes to the code (e.g. print a variable or comment a line) if you think it may help you diagnose the issue. Note that if you make a temporary change to the code you must restore the code to its original state for the task to be validated.
- You have a maximum of 20 minutes to complete each task.
- Feel free to consult this document at any time.
- All the information that you need to solve the tasks is present in this document. However, feel free to consult any documentation and perform any web searches you may need, at any time, **in the remote computer where you're performing the tasks**.
- If something isn't clear in these instructions or in the descriptions of the tasks themselves please alert the experiment observer immediately.

Instructions

- In the C:/Users/DockerliveTest/Desktop folder, you'll find 3 folders - "DF1", "DF2", "DF3". Each of these folders contains a task.
- Do these 3 tasks in order and fill the respective section of the questionnaire after each task.
- Instructions for each of the tasks are available in the remaining sections of this document.

- At the end of each task please run the following command on a powershell window: *docker-cleanup-script*
- You can now read the instructions and start the task “DF1”.

DF1

1. Open Visual Studio Code using the shortcut in the desktop.
2. In Visual Studio Code open the “DF1” folder using the menu “File” -> “Open Folder…”
3. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must show: "some data"
    - 'node' process must be running in container
4. Alert the experiment observer once you've met the criteria in point 3.
5. Execute the following command on a powershell window: *docker-cleanup-script*
6. Fill the “DF1” section of the questionnaire.
7. Move to the next task.

DF2

1. In Visual Studio Code open the “DF2” folder using the menu “File” -> “Open Folder…”
2. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must NOT show: "Error downloading file" nor "Error writing file"
    - Container stdout must show: “Success!”
    - Container must download 9MB-15MB of data
3. Alert the experiment observer once you've met the criteria in point 2.
4. Execute the following command on a powershell window: *docker-cleanup-script*
5. Fill the “DF2” section of the questionnaire.
6. Move to the next task.

DF3

1. In Visual Studio Code open the “DF3” folder using the menu “File” -> “Open Folder…”
2. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must show: "Listening!"
    - Container stdout must not show: "Could not bind"
    - Container must have a TCP service running on the exposed port 3000
3. Alert the experiment observer once you've met the criteria in point 2.
4. Execute the following command on a powershell window: *docker-cleanup-script*
5. Fill the remaining sections of the questionnaire.

C.2 Experimental Group

General instructions

Before proceeding, please fill the “Demographic Information” and “Skills and Experience” sections of the questionnaire linked below. Once you reach the “DF1” section, read the rest of this document.

https://forms.gle/fXuQu3euNfGgic769

In this experiment you will perform 3 tasks. In each task you must edit a Dockerfile using Visual Studio Code. To help you in these tasks, you’ll have access to a set of extensions which will provide feedback while you edit the Dockerfiles. The combination of Visual Studio Code with these extensions will be referred to as the “environment” in this document and in the questionnaire. The features available in these extensions are described in the next sections of this document.

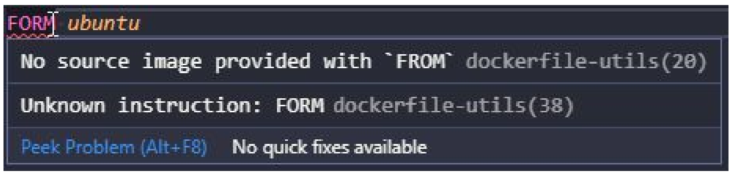
You have 15 minutes to read and understand the features described in this document. You are allowed to consult this document at any point during the task execution.

Available Features

This environment includes a VSCode extension which automatically compiles an image from the Dockerfile while it is being edited, raises a container with that image and executes some verifications. The result of those tests are shown to the developer in VSCode as explained in the following paragraphs.

1- Static Analysis Errors

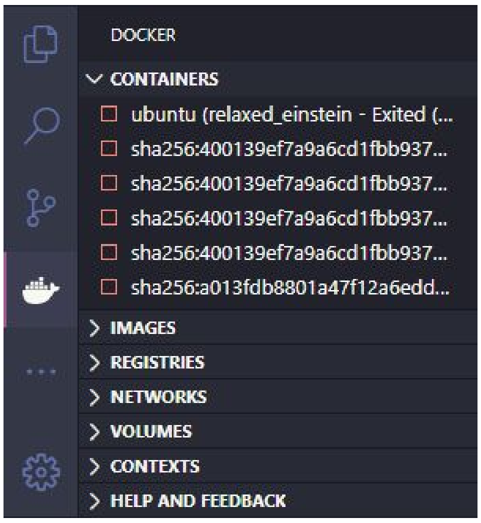
This environment analyses the Dockerfile’s syntax, underlining any syntax errors that it finds.



```
FORM ubuntu
No source image provided with `FROM` dockerfile-utils(20)
Unknown instruction: FORM dockerfile-utils(38)
```

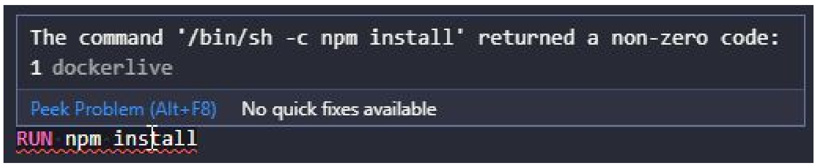
2- Docker Management

This environment provides a sidebar menu which allows the programmer to perform quick actions related to Docker containers, images, registries, networks, volumes and contexts (e.g. start container, stop container, view container logs, delete image).


DOCKER
CONTAINERS
ubuntu (relaxed_einstein - Exited (...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:400139ef7a9a6cd1fbb937...
sha256:a013fdb8801a47f12a6edd...
IMAGES
REGISTRIES
NETWORKS
VOLUMES
CONTEXTS
HELP AND FEEDBACK

3- Image Build and Container Runtime Errors

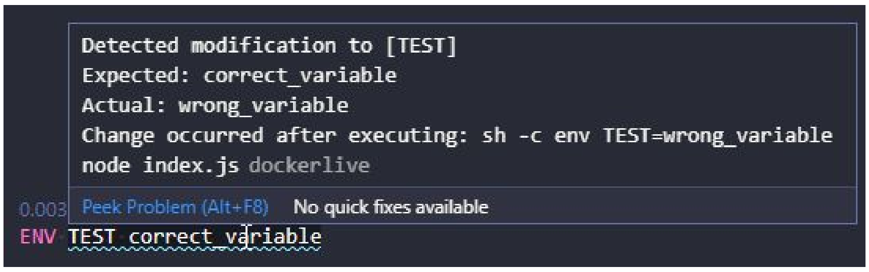
Instructions which cause errors while building the image or instantiating the image in a container are underlined in **red**. Hovering the underlined instruction shows more details about the error.



```
The command '/bin/sh -c npm install' returned a non-zero code:
1 dockerlive
Peek Problem (Alt+F8)    No quick fixes available
RUN npm install
```

4- Changes to environment variables

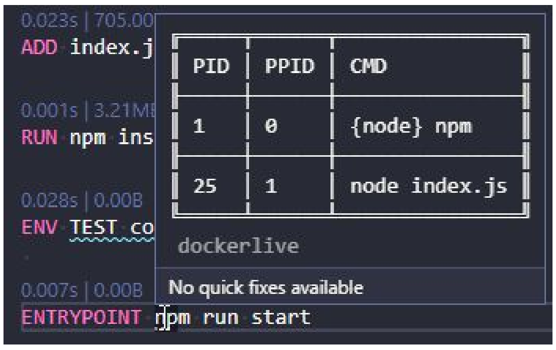
ENV instructions are underlined in **blue** when the value that they set is changed during the container's execution. Hovering the underlined instruction shows more details, including the process which changed the environment variable.



```
Detected modification to [TEST]
Expected: correct_variable
Actual: wrong_variable
Change occurred after executing: sh -c env TEST=wrong_variable
node index.js dockerlive
0.003 Peek Problem (Alt+F8)    No quick fixes available
ENV TEST correct_variable
```

5- Processes running in the container

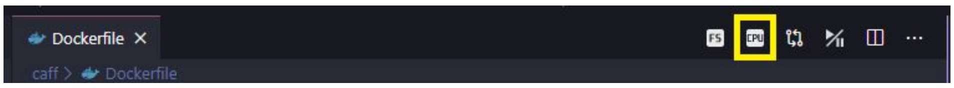
By hovering over the ENTRYPOINT/CMD instruction you can see the processes running in the container.



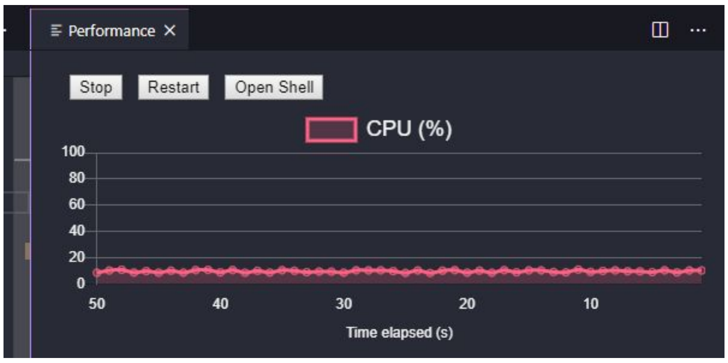
```
0.023s | 705.00
ADD index.j
0.001s | 3.21MB
RUN npm ins
0.028s | 0.00B
ENV TEST co
0.007s | 0.00B
ENTRYPOINT npm run start
```

```
┌─────┬──────┬──────────────────┐
│ PID │ PPID │ CMD              │
├─────┼──────┼──────────────────┤
│ 1   │ 0    │ {node} npm       │
├─────┼──────┼──────────────────┤
│ 25  │ 1    │ node index.js    │
└─────┴──────┴──────────────────┘
dockerlive
No quick fixes available
```

6- Container performance statistics



You can visualize performance statistics of the container by clicking the “CPU” button, located in the upper right corner of the editor tab which contains the Dockerfile. By clicking this button, a new tab with performance graphs will be visible. If the graphs are updated every second, then the container is running. If the graphs are stopped, then the container is stopped. When a new container starts (by editing the Dockerfile or pressing the restart button) data is erased from the graphs.

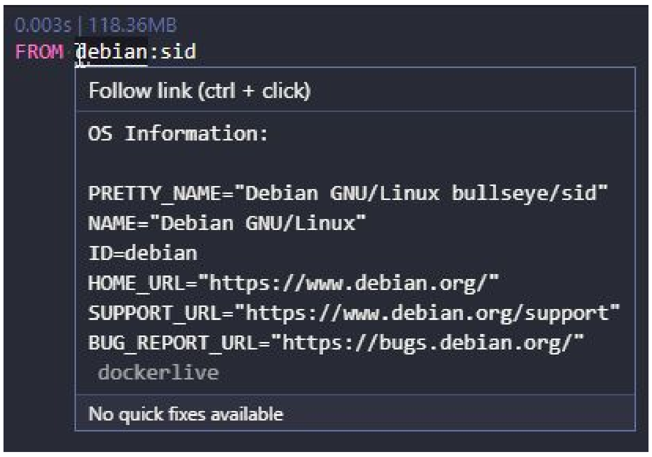


On this page there are also 3 buttons available:
- Stop - Stops the running container

- Restart - Restarts/Starts the container
- Open Shell - Open an interactive shell inside the container

7- Base image OS information

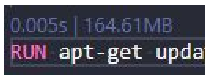
By hovering the image name on the FROM instruction, you can see some information about the OS that the container is running (including the OS Family, Version and Kernel version).



```
0.003s | 118.36MB
FROM debian:sid
    Follow link (ctrl + click)
    OS Information:

    PRETTY_NAME="Debian GNU/Linux bullseye/sid"
    NAME="Debian GNU/Linux"
    ID=debian
    HOME_URL="https://www.debian.org/"
    SUPPORT_URL="https://www.debian.org/support"
    BUG_REPORT_URL="https://bugs.debian.org/"
     dockerlive
    No quick fixes available
```

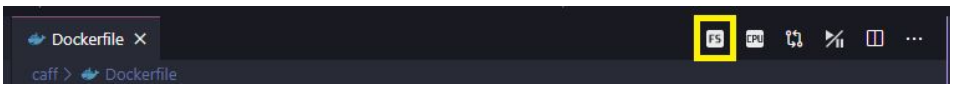
8- Layer Size and Build Time

Above every instruction you can see a small text indicating the time it took to build that layer and the size that the layer takes.



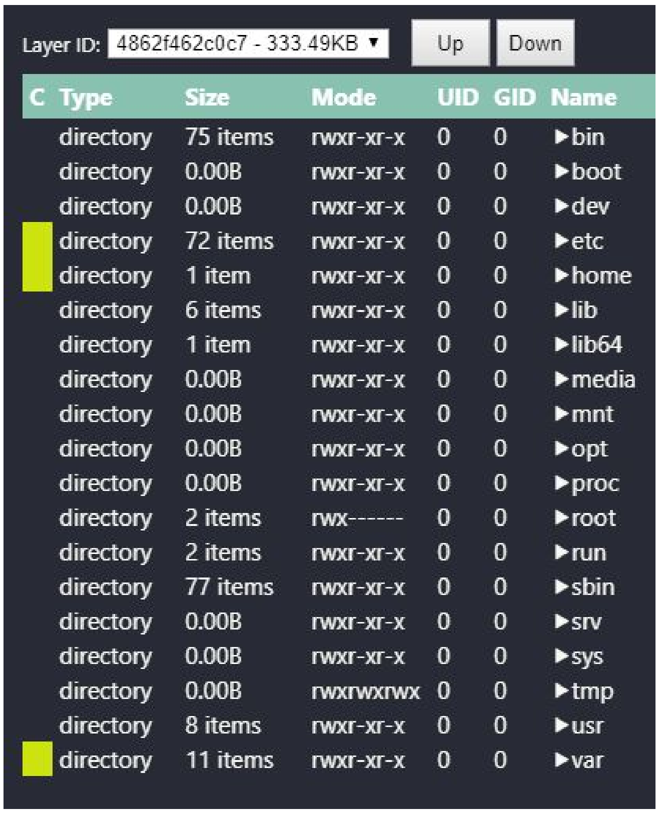
```
0.005s | 164.61MB
RUN apt-get upda
```

9- Explore each layer's filesystem



You can explore each layer's filesystem by clicking in the "FS" button, located in the upper right corner of the editor tab which contains the Dockerfile. By clicking this button a new tab will allow the developer to navigate through the aggregated filesystem of each of the layers.

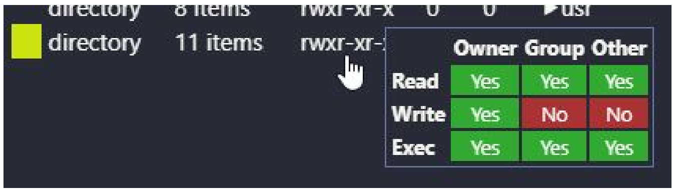
On the top of the table there is a dropdown which allows you to select the layer that is currently being displayed. Files which were created/edited/removed in the selected layer are marked with a yellow square.


Layer ID: 4862f462c0c7 - 333.49KB ▾ Up Down

C	Type	Size	Mode	UID	GID	Name
	directory	75 items	rwxr-xr-x	0	0	▶bin
	directory	0.00B	rwxr-xr-x	0	0	▶boot
	directory	0.00B	rwxr-xr-x	0	0	▶dev
🟨	directory	72 items	rwxr-xr-x	0	0	▶etc
🟨	directory	1 item	rwxr-xr-x	0	0	▶home
	directory	6 items	rwxr-xr-x	0	0	▶lib
	directory	1 item	rwxr-xr-x	0	0	▶lib64
	directory	0.00B	rwxr-xr-x	0	0	▶media
	directory	0.00B	rwxr-xr-x	0	0	▶mnt
	directory	0.00B	rwxr-xr-x	0	0	▶opt
	directory	0.00B	rwxr-xr-x	0	0	▶proc
	directory	2 items	rwx------	0	0	▶root
	directory	2 items	rwxr-xr-x	0	0	▶run
	directory	77 items	rwxr-xr-x	0	0	▶sbin
	directory	0.00B	rwxr-xr-x	0	0	▶srv
	directory	0.00B	rwxr-xr-x	0	0	▶sys
	directory	0.00B	rwxrwxrwx	0	0	▶tmp
	directory	8 items	rwxr-xr-x	0	0	▶usr
🟨	directory	11 items	rwxr-xr-x	0	0	▶var

You can expand/collapse folders by clicking on their name.

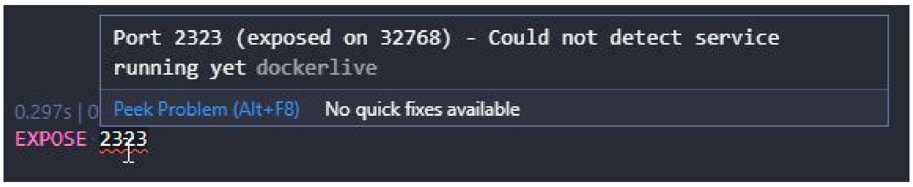
By hovering the permissions, you can see a small window which will help you interpret the permissions for each of the entities. These are in the UNIX permissions format.


	directory	8 items	rwxr-xr-x	0	0	▶usr
🟨	directory	11 items	rwxr-xr-			

	Owner	Group	Other
Read	Yes	Yes	Yes
Write	Yes	No	No
Exec	Yes	Yes	Yes

10- Service discovery

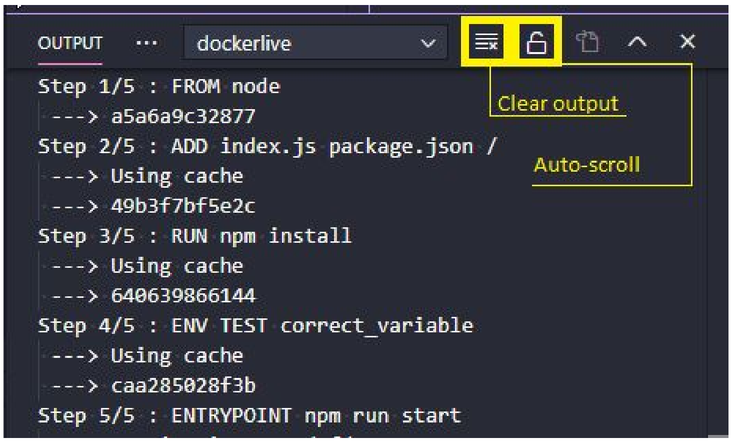
This environment will automatically try to detect any services running on ports that are exposed with the EXPOSE instruction. If no service is detected, an error underline will be displayed. By hovering over the port number on the instruction, you can see the name and protocol of the detected service.



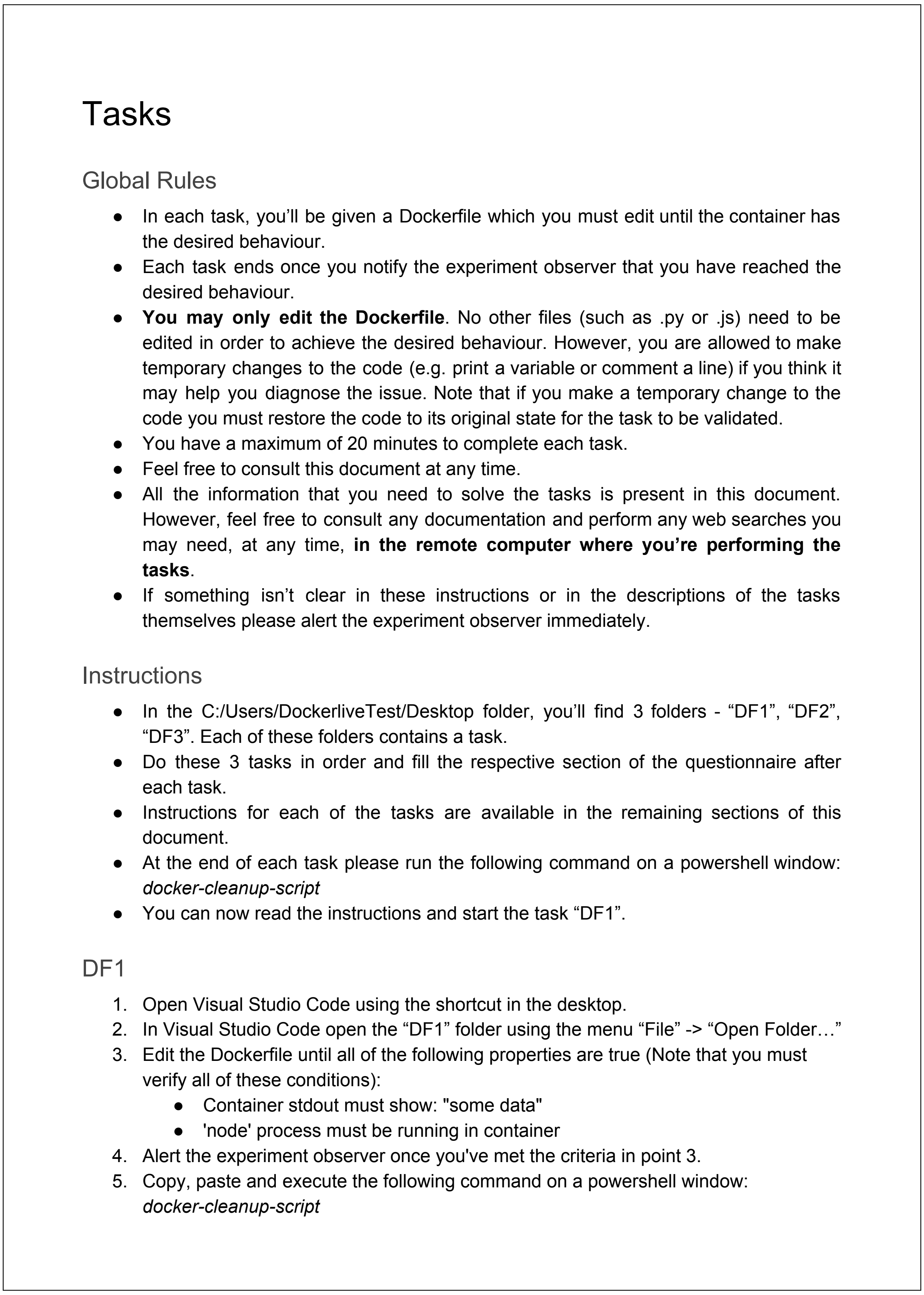
```
Port 2323 (exposed on 32768) - Could not detect service
running yet dockerlive
Peek Problem (Alt+F8)    No quick fixes available
0.297s | 0
EXPOSE 2323
```

11- Container log output

The output of the docker build and the docker container is displayed in the Output pane in VSCode. This pane opens automatically when a Dockerfile is opened and may also be opened in the VSCode top bar (View -> Output). It's advised to enable auto-scrolling (open padlock icon, as displayed below).



```
Step 1/5 : FROM node
 ---> a5a6a9c32877
Step 2/5 : ADD index.js package.json /
 ---> Using cache
 ---> 49b3f7bf5e2c
Step 3/5 : RUN npm install
 ---> Using cache
 ---> 640639866144
Step 4/5 : ENV TEST correct_variable
 ---> Using cache
 ---> caa285028f3b
Step 5/5 : ENTRYPOINT npm run start
```



Tasks

Global Rules

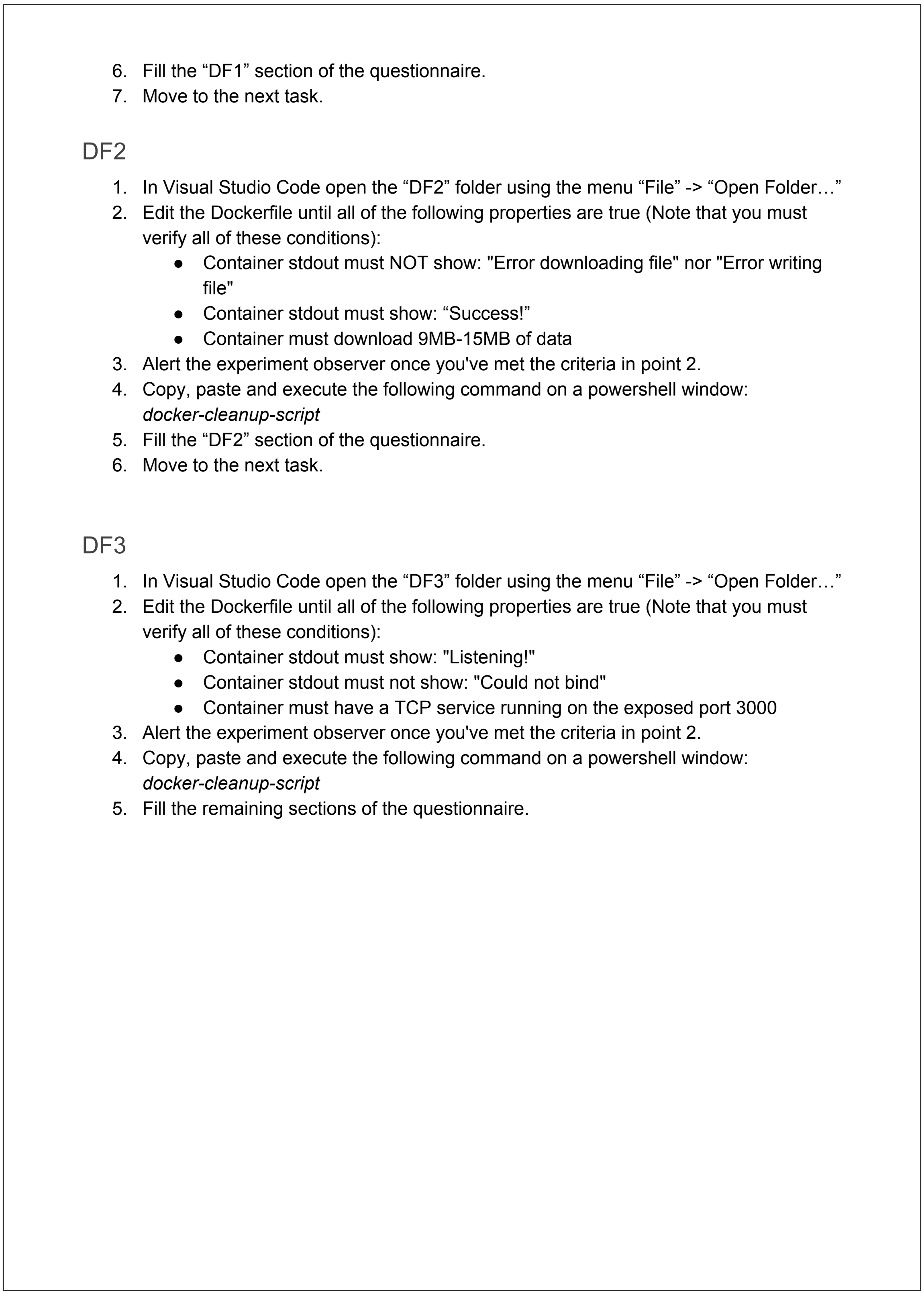
- In each task, you'll be given a Dockerfile which you must edit until the container has the desired behaviour.
- Each task ends once you notify the experiment observer that you have reached the desired behaviour.
- **You may only edit the Dockerfile**. No other files (such as .py or .js) need to be edited in order to achieve the desired behaviour. However, you are allowed to make temporary changes to the code (e.g. print a variable or comment a line) if you think it may help you diagnose the issue. Note that if you make a temporary change to the code you must restore the code to its original state for the task to be validated.
- You have a maximum of 20 minutes to complete each task.
- Feel free to consult this document at any time.
- All the information that you need to solve the tasks is present in this document. However, feel free to consult any documentation and perform any web searches you may need, at any time, **in the remote computer where you're performing the tasks**.
- If something isn't clear in these instructions or in the descriptions of the tasks themselves please alert the experiment observer immediately.

Instructions

- In the C:/Users/DockerliveTest/Desktop folder, you'll find 3 folders - "DF1", "DF2", "DF3". Each of these folders contains a task.
- Do these 3 tasks in order and fill the respective section of the questionnaire after each task.
- Instructions for each of the tasks are available in the remaining sections of this document.
- At the end of each task please run the following command on a powershell window: *docker-cleanup-script*
- You can now read the instructions and start the task "DF1".

DF1

1. Open Visual Studio Code using the shortcut in the desktop.
2. In Visual Studio Code open the "DF1" folder using the menu "File" -> "Open Folder…"
3. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must show: "some data"
    - 'node' process must be running in container
4. Alert the experiment observer once you've met the criteria in point 3.
5. Copy, paste and execute the following command on a powershell window: *docker-cleanup-script*



6. Fill the “DF1” section of the questionnaire.
7. Move to the next task.

DF2

1. In Visual Studio Code open the “DF2” folder using the menu “File” -> “Open Folder…”
2. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must NOT show: "Error downloading file" nor "Error writing file"
    - Container stdout must show: “Success!”
    - Container must download 9MB-15MB of data
3. Alert the experiment observer once you've met the criteria in point 2.
4. Copy, paste and execute the following command on a powershell window: *docker-cleanup-script*
5. Fill the “DF2” section of the questionnaire.
6. Move to the next task.

DF3

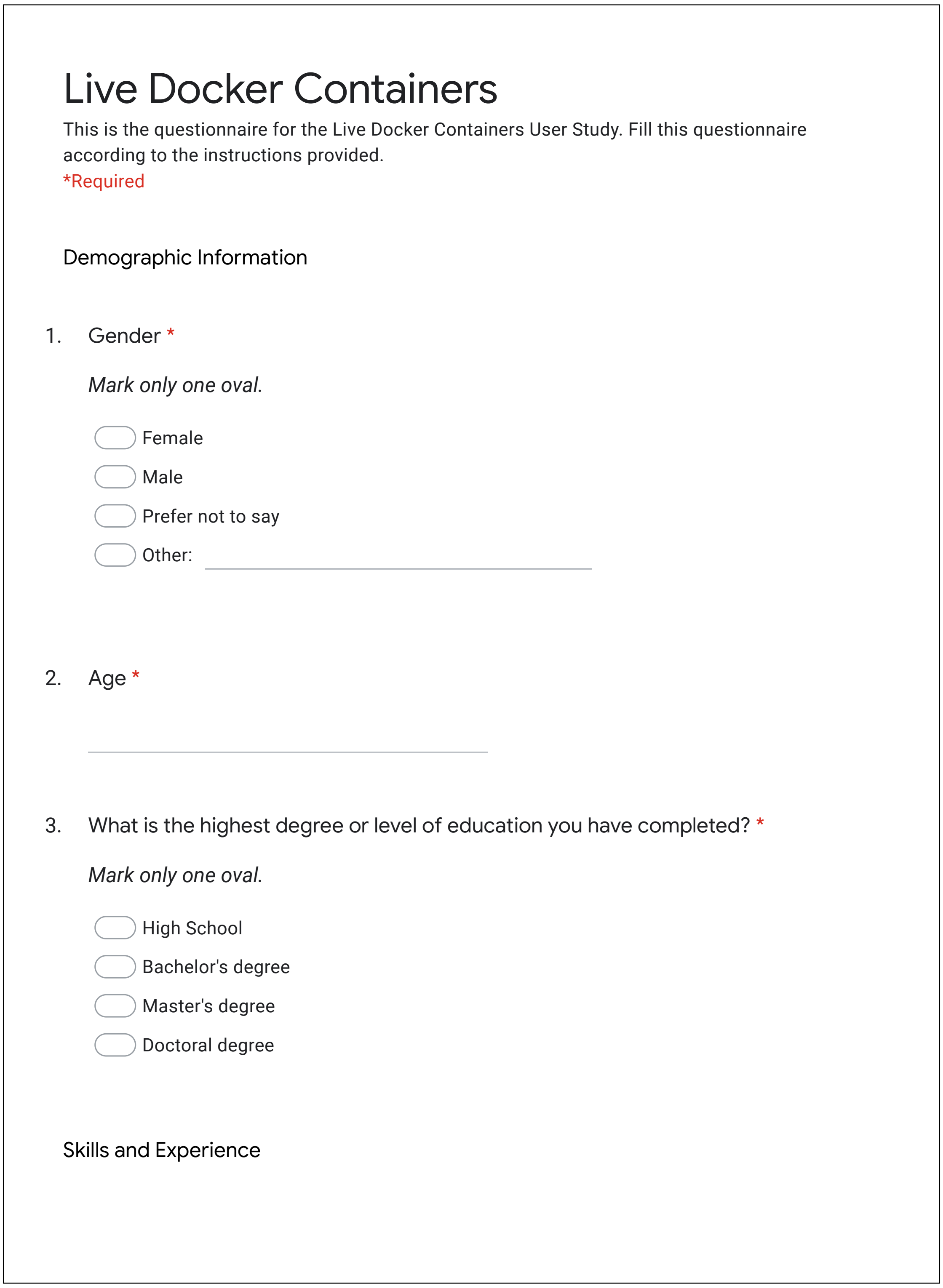
1. In Visual Studio Code open the “DF3” folder using the menu “File” -> “Open Folder…”
2. Edit the Dockerfile until all of the following properties are true (Note that you must verify all of these conditions):
    - Container stdout must show: "Listening!"
    - Container stdout must not show: "Could not bind"
    - Container must have a TCP service running on the exposed port 3000
3. Alert the experiment observer once you've met the criteria in point 2.
4. Copy, paste and execute the following command on a powershell window: *docker-cleanup-script*
5. Fill the remaining sections of the questionnaire.

Appendix D

User Study Questionnaire

This appendix presents the questionnaire described in Chapter 7. This questionnaire was answered throughout the user study by all participants. Section D.1 contains the questionnaire provided to the control group, while Section D.2 contains the questionnaire provided to the experimental group.

D.1 Control Group



Live Docker Containers

This is the questionnaire for the Live Docker Containers User Study. Fill this questionnaire according to the instructions provided.

*Required

Demographic Information

1. Gender *

   Mark only one oval.

   - Female
   - Male
   - Prefer not to say
   - Other: ____________________

2. Age *

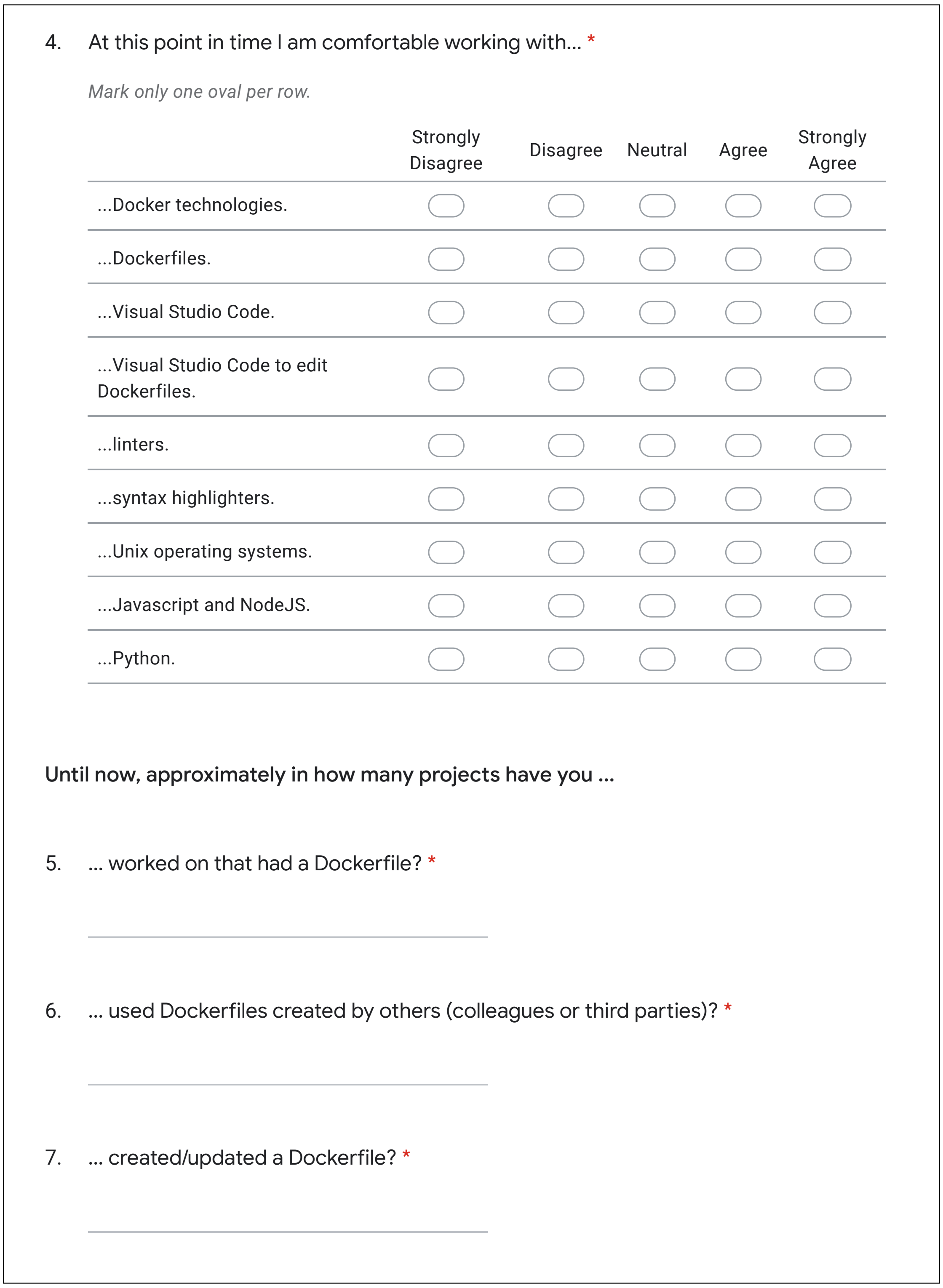
   ____

3. What is the highest degree or level of education you have completed? *

   Mark only one oval.

   - High School
   - Bachelor's degree
   - Master's degree
   - Doctoral degree

Skills and Experience



4. At this point in time I am comfortable working with... *

Mark only one oval per row.

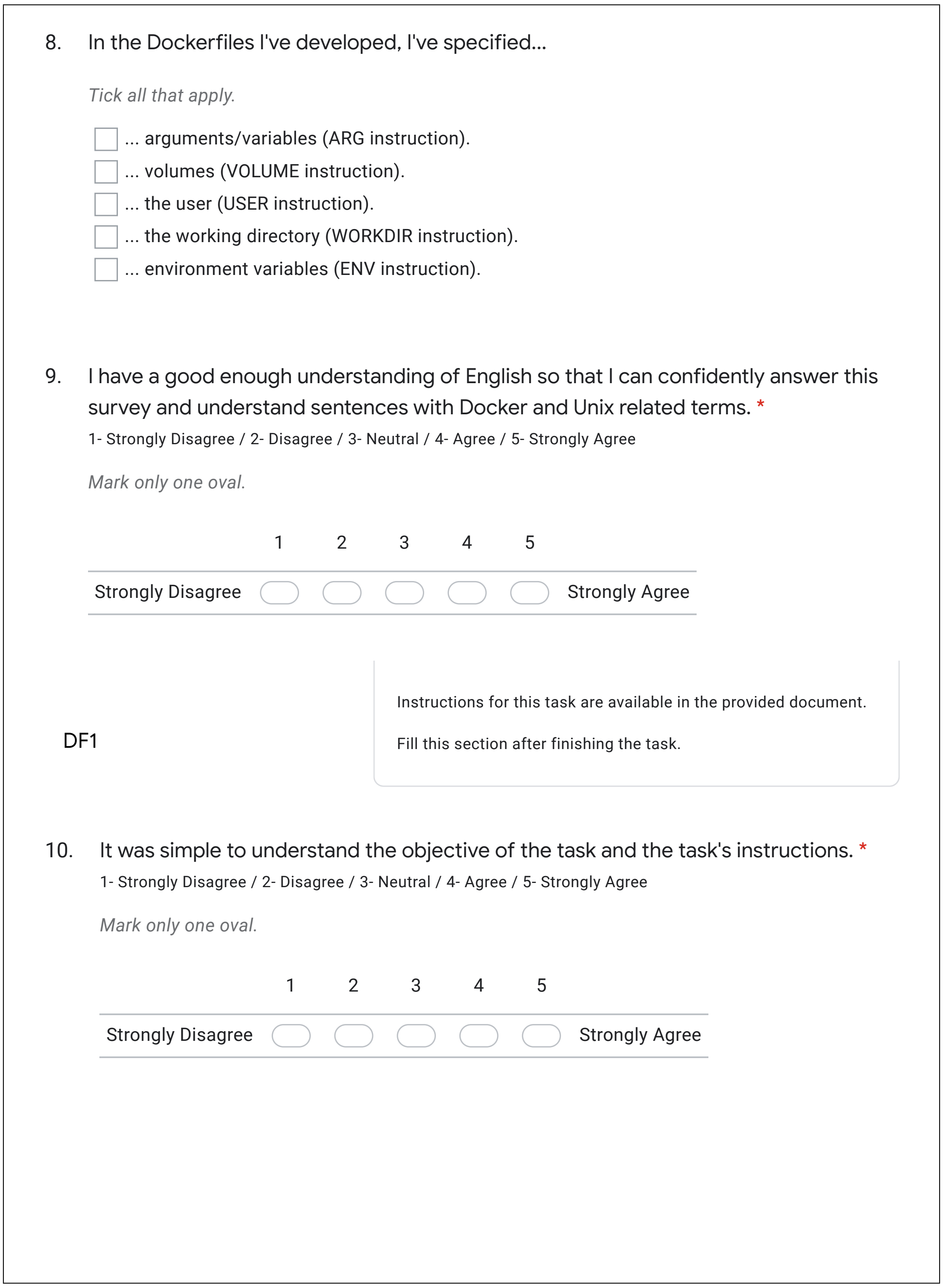
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...Docker technologies.	◯	◯	◯	◯	◯
...Dockerfiles.	◯	◯	◯	◯	◯
...Visual Studio Code.	◯	◯	◯	◯	◯
...Visual Studio Code to edit Dockerfiles.	◯	◯	◯	◯	◯
...linters.	◯	◯	◯	◯	◯
...syntax highlighters.	◯	◯	◯	◯	◯
...Unix operating systems.	◯	◯	◯	◯	◯
...Javascript and NodeJS.	◯	◯	◯	◯	◯
...Python.	◯	◯	◯	◯	◯

Until now, approximately in how many projects have you ...

5. ... worked on that had a Dockerfile? *

6. ... used Dockerfiles created by others (colleagues or third parties)? *

7. ... created/updated a Dockerfile? *



8. In the Dockerfiles I've developed, I've specified...

Tick all that apply.

- ☐ ... arguments/variables (ARG instruction).
- ☐ ... volumes (VOLUME instruction).
- ☐ ... the user (USER instruction).
- ☐ ... the working directory (WORKDIR instruction).
- ☐ ... environment variables (ENV instruction).

9. I have a good enough understanding of English so that I can confidently answer this survey and understand sentences with Docker and Unix related terms. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

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

DF1

Instructions for this task are available in the provided document.

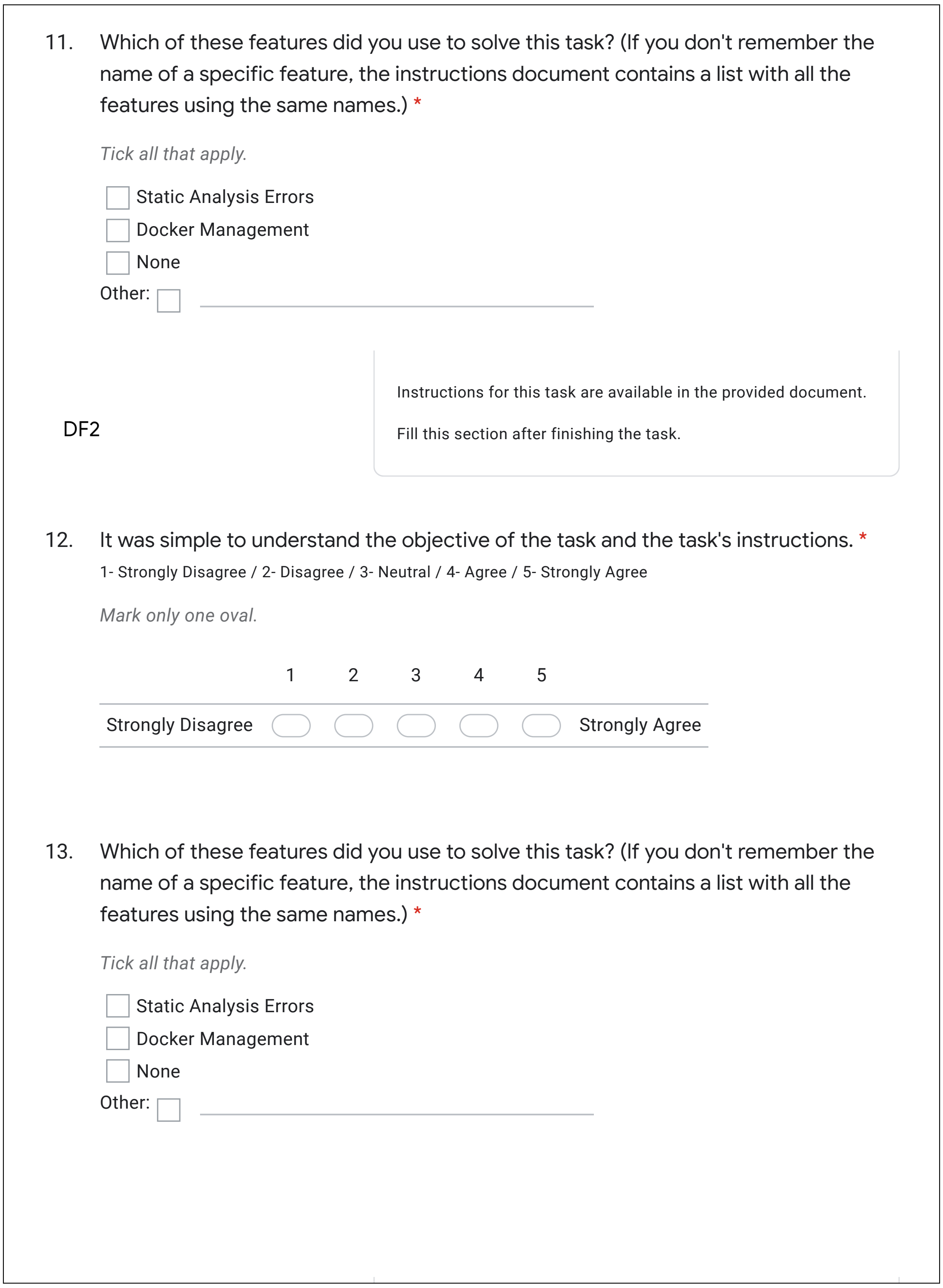
Fill this section after finishing the task.

10. It was simple to understand the objective of the task and the task's instructions. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

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



11. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

Tick all that apply.

☐ Static Analysis Errors
☐ Docker Management
☐ None
Other: ☐ ____________________

DF2

Instructions for this task are available in the provided document.

Fill this section after finishing the task.

12. It was simple to understand the objective of the task and the task's instructions. *
1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

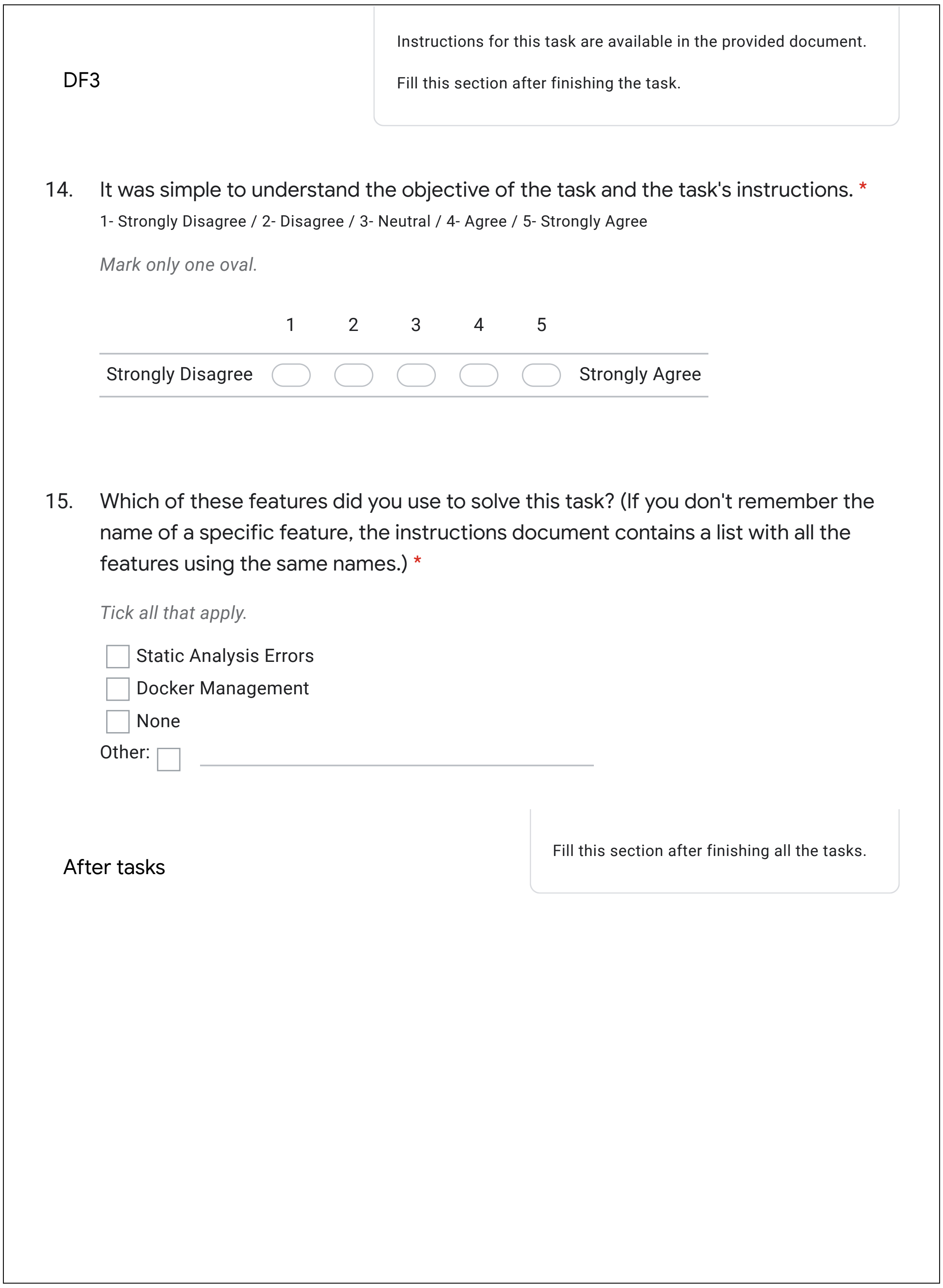
Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

13. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

Tick all that apply.

☐ Static Analysis Errors
☐ Docker Management
☐ None
Other: ☐ ____________________



DF3

Instructions for this task are available in the provided document.

Fill this section after finishing the task.

14. It was simple to understand the objective of the task and the task's instructions. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

15. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

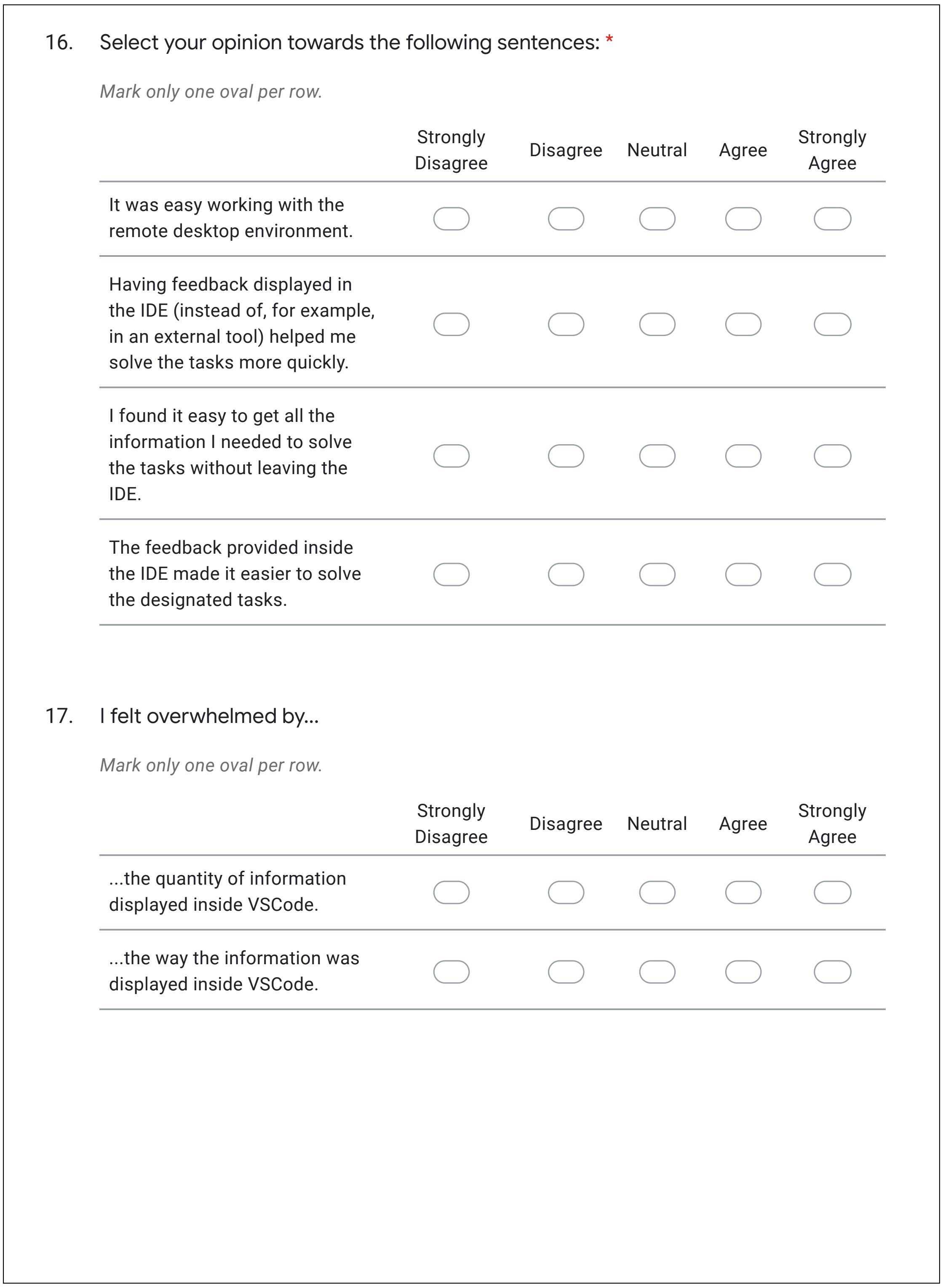
Tick all that apply.

- ☐ Static Analysis Errors
- ☐ Docker Management
- ☐ None

Other: ☐ ____________________

After tasks

Fill this section after finishing all the tasks.



16. Select your opinion towards the following sentences: *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
It was easy working with the remote desktop environment.	◯	◯	◯	◯	◯
Having feedback displayed in the IDE (instead of, for example, in an external tool) helped me solve the tasks more quickly.	◯	◯	◯	◯	◯
I found it easy to get all the information I needed to solve the tasks without leaving the IDE.	◯	◯	◯	◯	◯
The feedback provided inside the IDE made it easier to solve the designated tasks.	◯	◯	◯	◯	◯

17. I felt overwhelmed by...

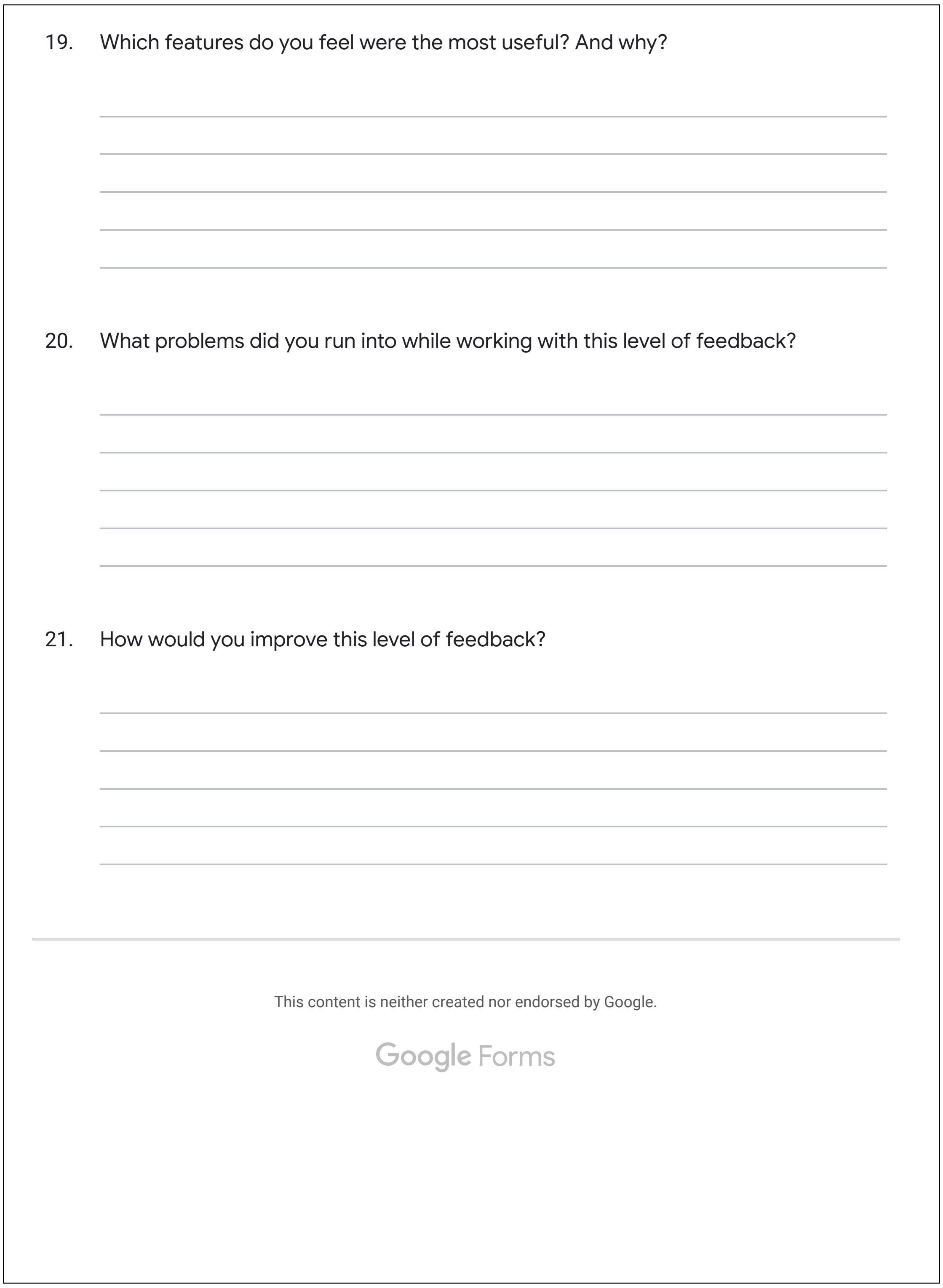
Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...the quantity of information displayed inside VSCode.	◯	◯	◯	◯	◯
...the way the information was displayed inside VSCode.	◯	◯	◯	◯	◯

18. During the execution of the tasks, the feedback provided in the IDE helped me... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...finding out what parent image is the most suitable.	◯	◯	◯	◯	◯
...finding out what are the dependencies of the system that must be added to the docker image.	◯	◯	◯	◯	◯
...finding out what are the Dockerfile commands that I need.	◯	◯	◯	◯	◯
...trying to understand if the resulting container is working as intended (e.g., running commands and tests on the container).	◯	◯	◯	◯	◯
...trying to understand why the resulting container is not working as intended.	◯	◯	◯	◯	◯
...finding out which commands are responsible for the container misbehaviour.	◯	◯	◯	◯	◯
...rebuilding the image and re-running the container to confirm that it is working as intended.	◯	◯	◯	◯	◯

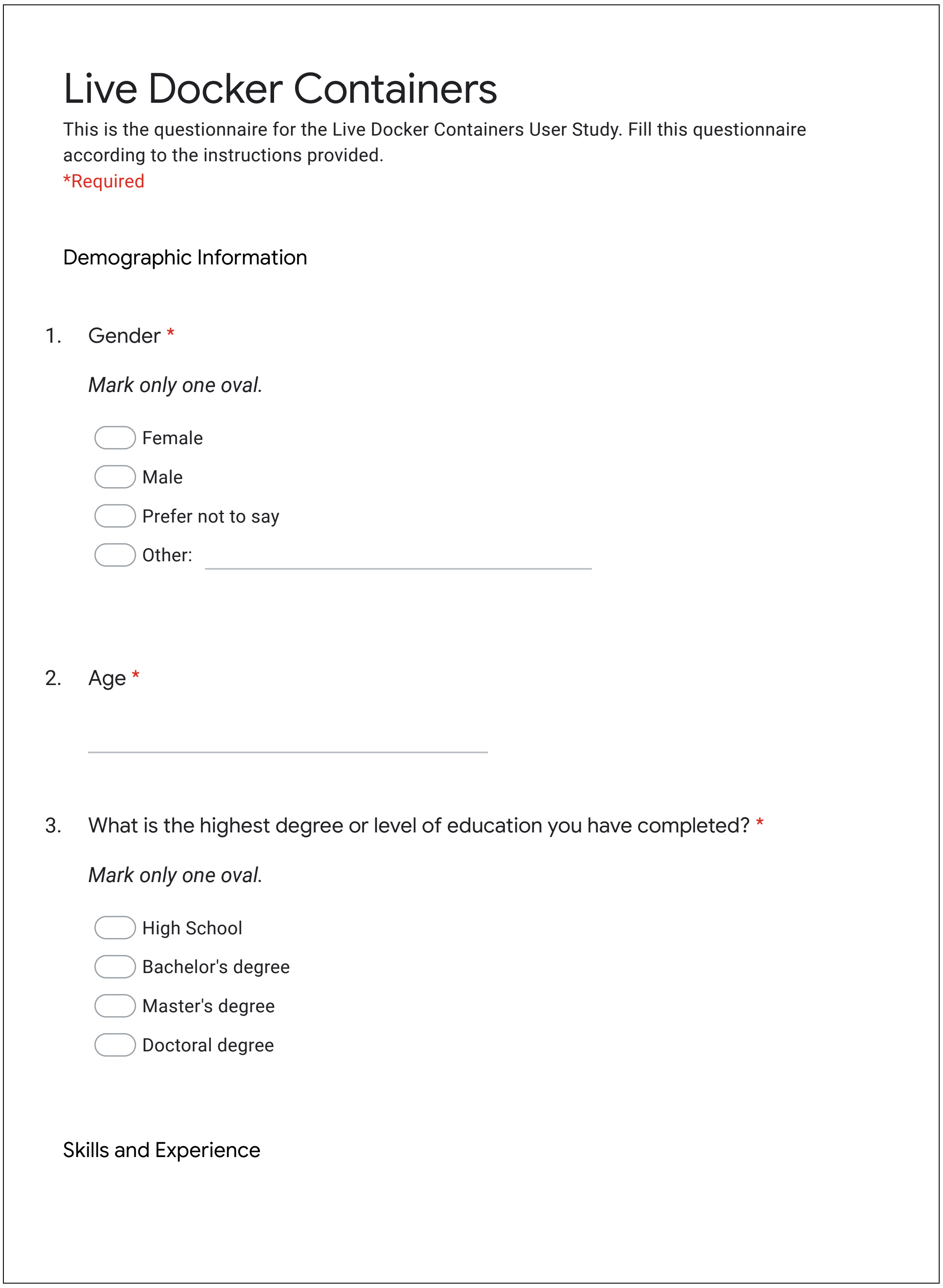


19. Which features do you feel were the most useful? And why?

20. What problems did you run into while working with this level of feedback?

21. How would you improve this level of feedback?

D.2 Experimental Group



Live Docker Containers

This is the questionnaire for the Live Docker Containers User Study. Fill this questionnaire according to the instructions provided.

*Required

Demographic Information

1. Gender *

Mark only one oval.

- ◯ Female
- ◯ Male
- ◯ Prefer not to say
- ◯ Other: ____________________

2. Age *

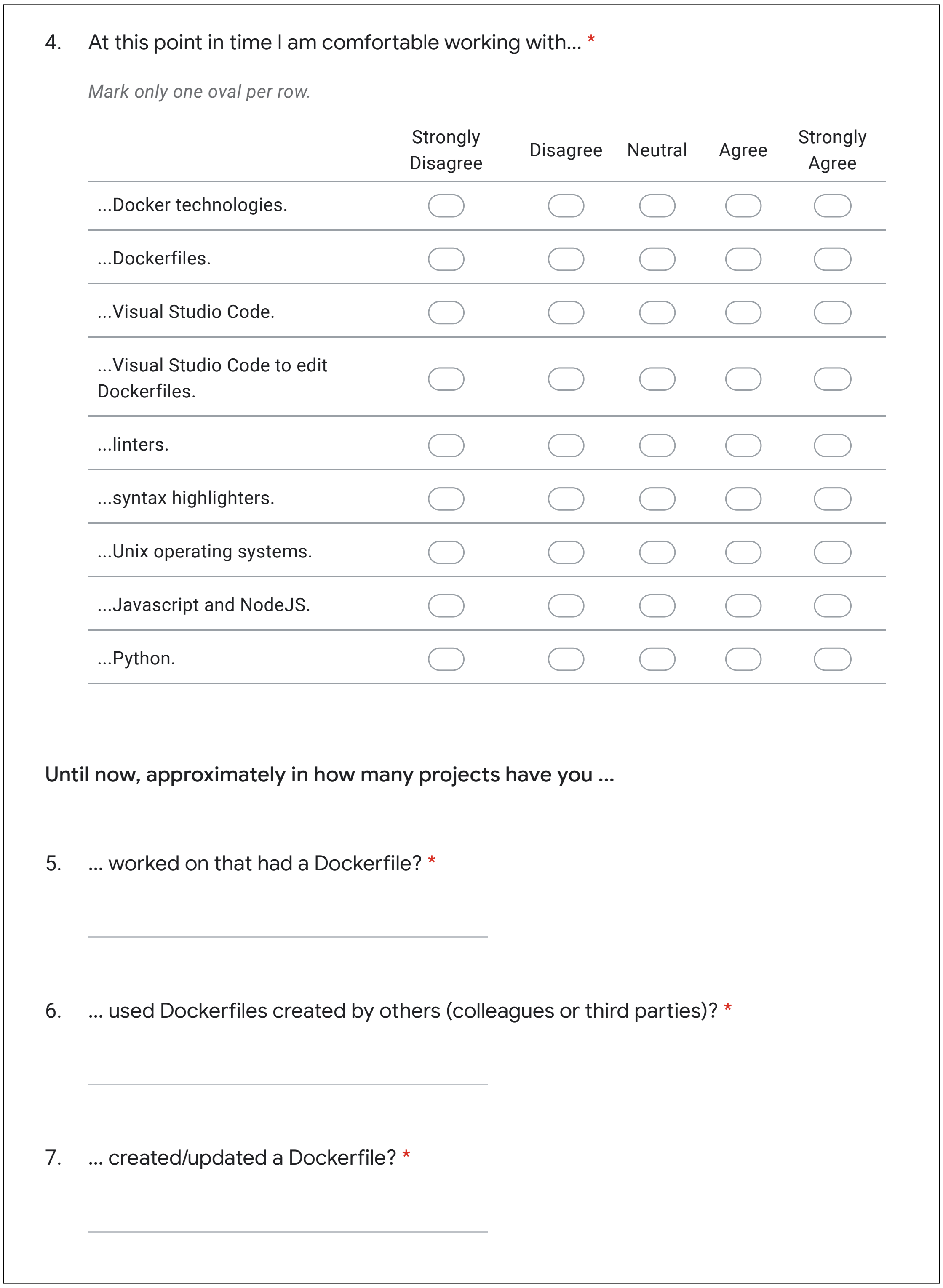
____

3. What is the highest degree or level of education you have completed? *

Mark only one oval.

- ◯ High School
- ◯ Bachelor's degree
- ◯ Master's degree
- ◯ Doctoral degree

Skills and Experience



4. At this point in time I am comfortable working with... *

Mark only one oval per row.

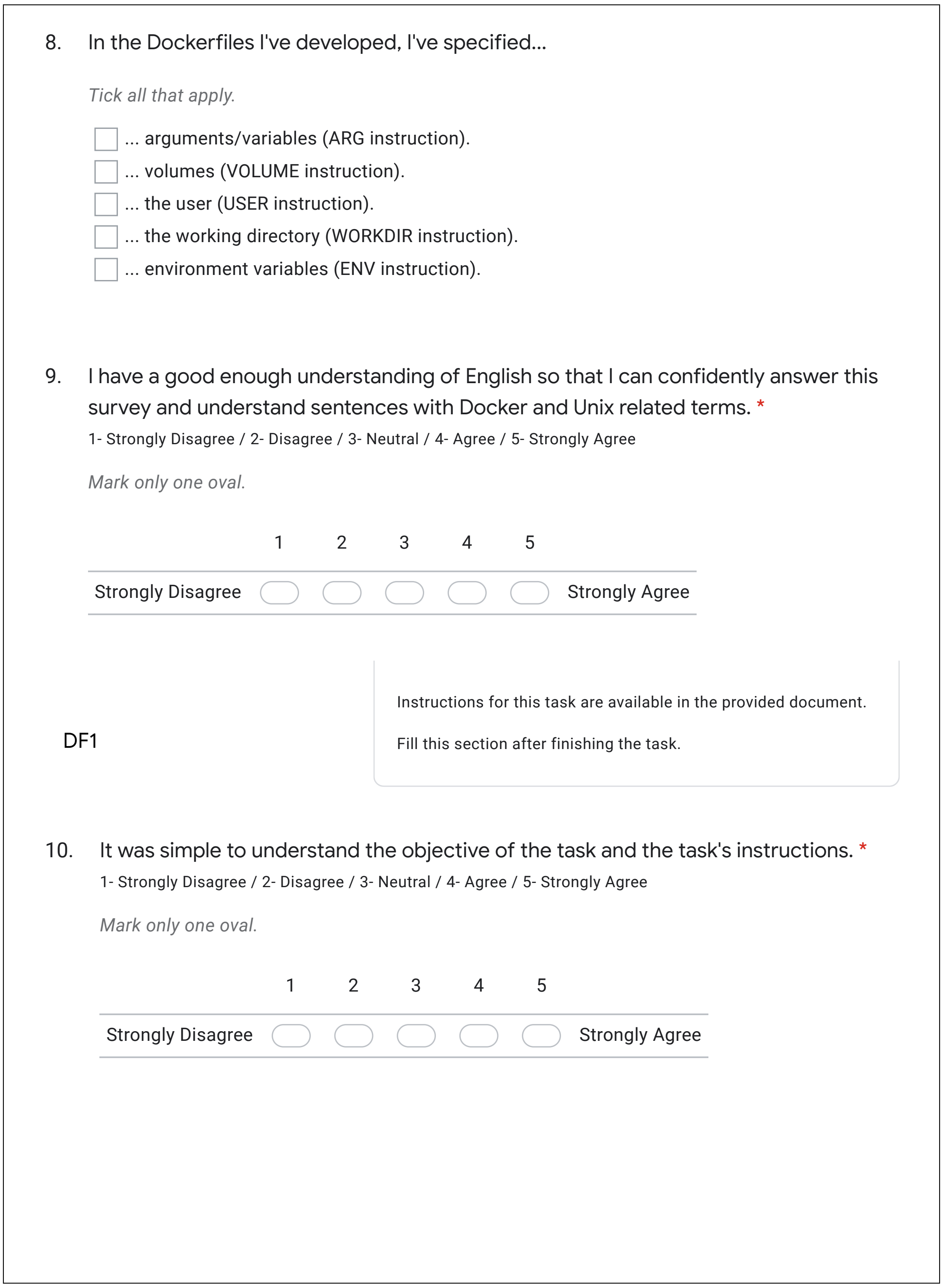
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...Docker technologies.	◯	◯	◯	◯	◯
...Dockerfiles.	◯	◯	◯	◯	◯
...Visual Studio Code.	◯	◯	◯	◯	◯
...Visual Studio Code to edit Dockerfiles.	◯	◯	◯	◯	◯
...linters.	◯	◯	◯	◯	◯
...syntax highlighters.	◯	◯	◯	◯	◯
...Unix operating systems.	◯	◯	◯	◯	◯
...Javascript and NodeJS.	◯	◯	◯	◯	◯
...Python.	◯	◯	◯	◯	◯

Until now, approximately in how many projects have you ...

5. ... worked on that had a Dockerfile? *

6. ... used Dockerfiles created by others (colleagues or third parties)? *

7. ... created/updated a Dockerfile? *



8. In the Dockerfiles I've developed, I've specified...

Tick all that apply.

- ☐ ... arguments/variables (ARG instruction).
- ☐ ... volumes (VOLUME instruction).
- ☐ ... the user (USER instruction).
- ☐ ... the working directory (WORKDIR instruction).
- ☐ ... environment variables (ENV instruction).

9. I have a good enough understanding of English so that I can confidently answer this survey and understand sentences with Docker and Unix related terms. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

DF1

Instructions for this task are available in the provided document.

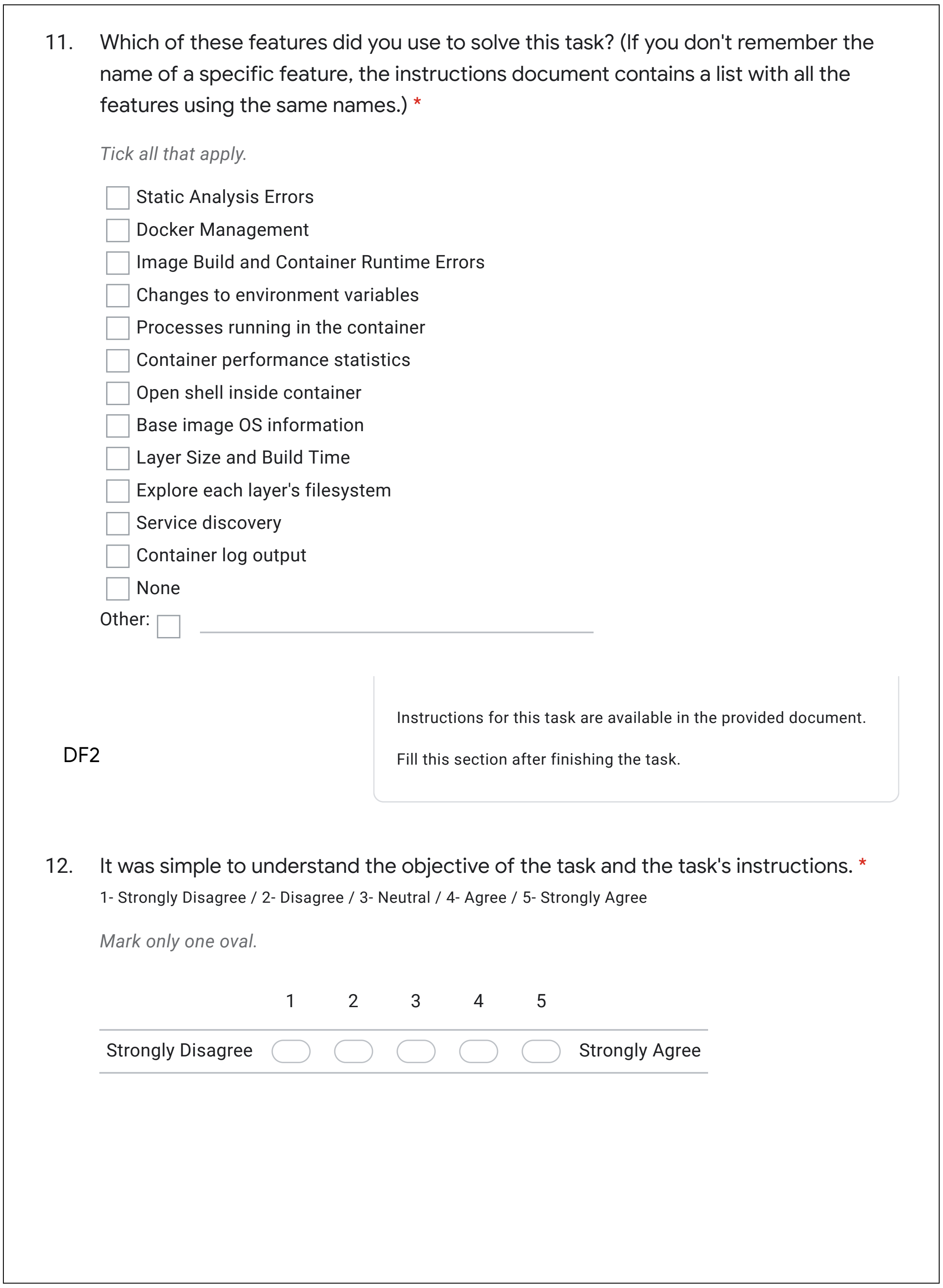
Fill this section after finishing the task.

10. It was simple to understand the objective of the task and the task's instructions. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree



11. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

Tick all that apply.

- ☐ Static Analysis Errors
- ☐ Docker Management
- ☐ Image Build and Container Runtime Errors
- ☐ Changes to environment variables
- ☐ Processes running in the container
- ☐ Container performance statistics
- ☐ Open shell inside container
- ☐ Base image OS information
- ☐ Layer Size and Build Time
- ☐ Explore each layer's filesystem
- ☐ Service discovery
- ☐ Container log output
- ☐ None

Other: ☐ ______________________

DF2

Instructions for this task are available in the provided document.

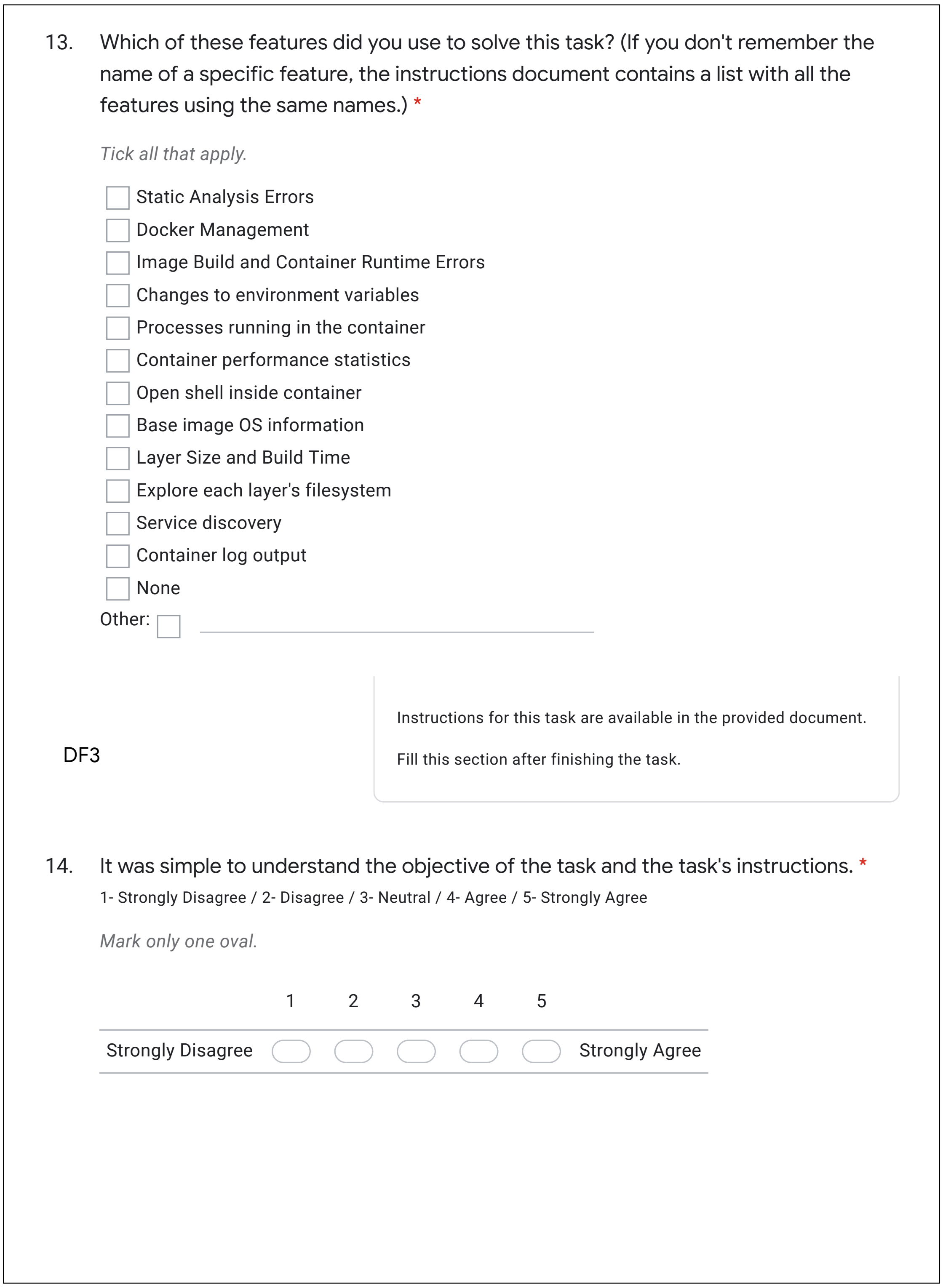
Fill this section after finishing the task.

12. It was simple to understand the objective of the task and the task's instructions. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree



13. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

Tick all that apply.

- ☐ Static Analysis Errors
- ☐ Docker Management
- ☐ Image Build and Container Runtime Errors
- ☐ Changes to environment variables
- ☐ Processes running in the container
- ☐ Container performance statistics
- ☐ Open shell inside container
- ☐ Base image OS information
- ☐ Layer Size and Build Time
- ☐ Explore each layer's filesystem
- ☐ Service discovery
- ☐ Container log output
- ☐ None

Other: ☐ ______________________

DF3

Instructions for this task are available in the provided document.

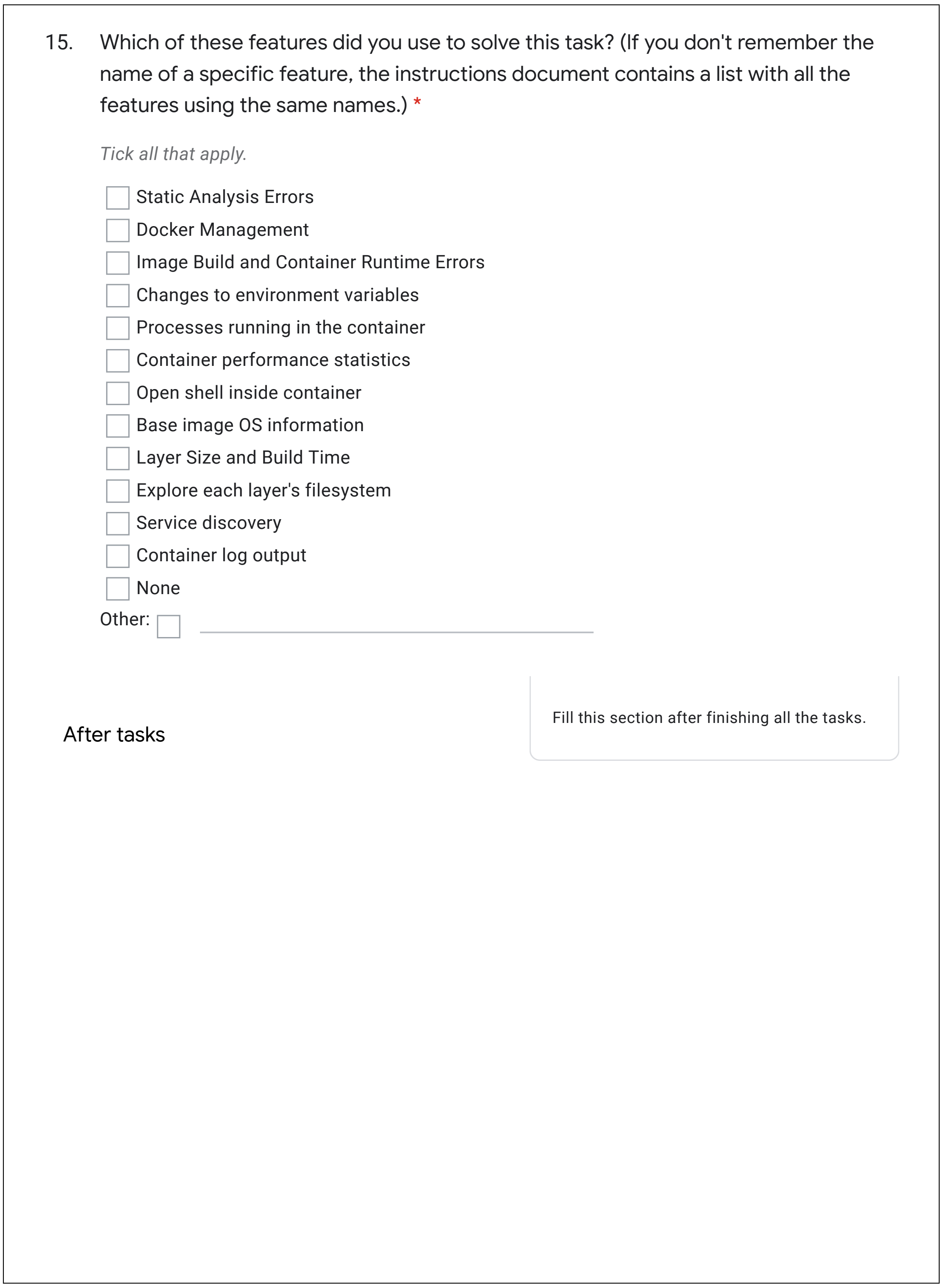
Fill this section after finishing the task.

14. It was simple to understand the objective of the task and the task's instructions. *

1- Strongly Disagree / 2- Disagree / 3- Neutral / 4- Agree / 5- Strongly Agree

Mark only one oval.

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



15. Which of these features did you use to solve this task? (If you don't remember the name of a specific feature, the instructions document contains a list with all the features using the same names.) *

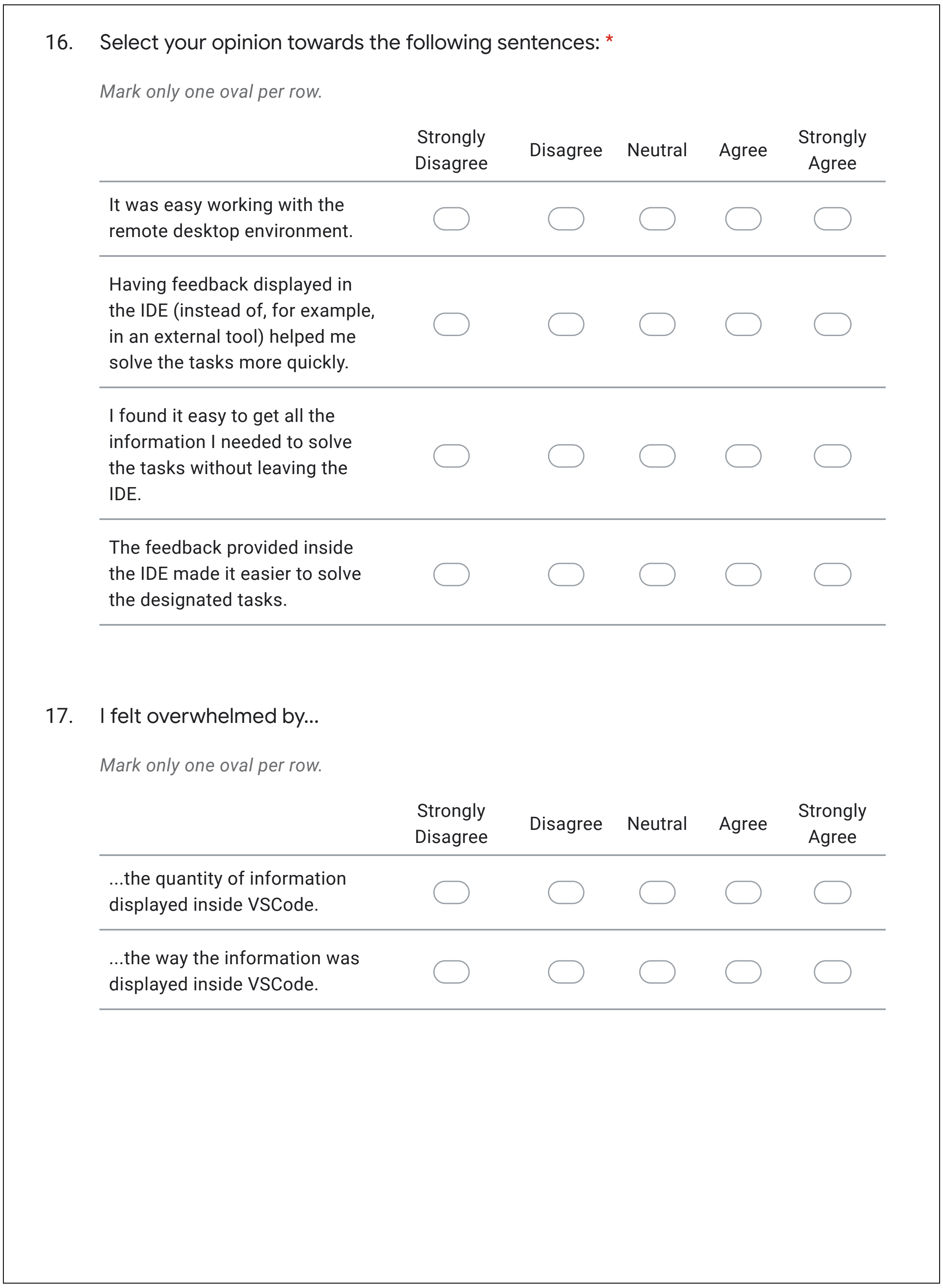
Tick all that apply.

- ☐ Static Analysis Errors
- ☐ Docker Management
- ☐ Image Build and Container Runtime Errors
- ☐ Changes to environment variables
- ☐ Processes running in the container
- ☐ Container performance statistics
- ☐ Open shell inside container
- ☐ Base image OS information
- ☐ Layer Size and Build Time
- ☐ Explore each layer's filesystem
- ☐ Service discovery
- ☐ Container log output
- ☐ None

Other: ☐ ____________________

After tasks

Fill this section after finishing all the tasks.



16. Select your opinion towards the following sentences: *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
It was easy working with the remote desktop environment.	◯	◯	◯	◯	◯
Having feedback displayed in the IDE (instead of, for example, in an external tool) helped me solve the tasks more quickly.	◯	◯	◯	◯	◯
I found it easy to get all the information I needed to solve the tasks without leaving the IDE.	◯	◯	◯	◯	◯
The feedback provided inside the IDE made it easier to solve the designated tasks.	◯	◯	◯	◯	◯

17. I felt overwhelmed by...

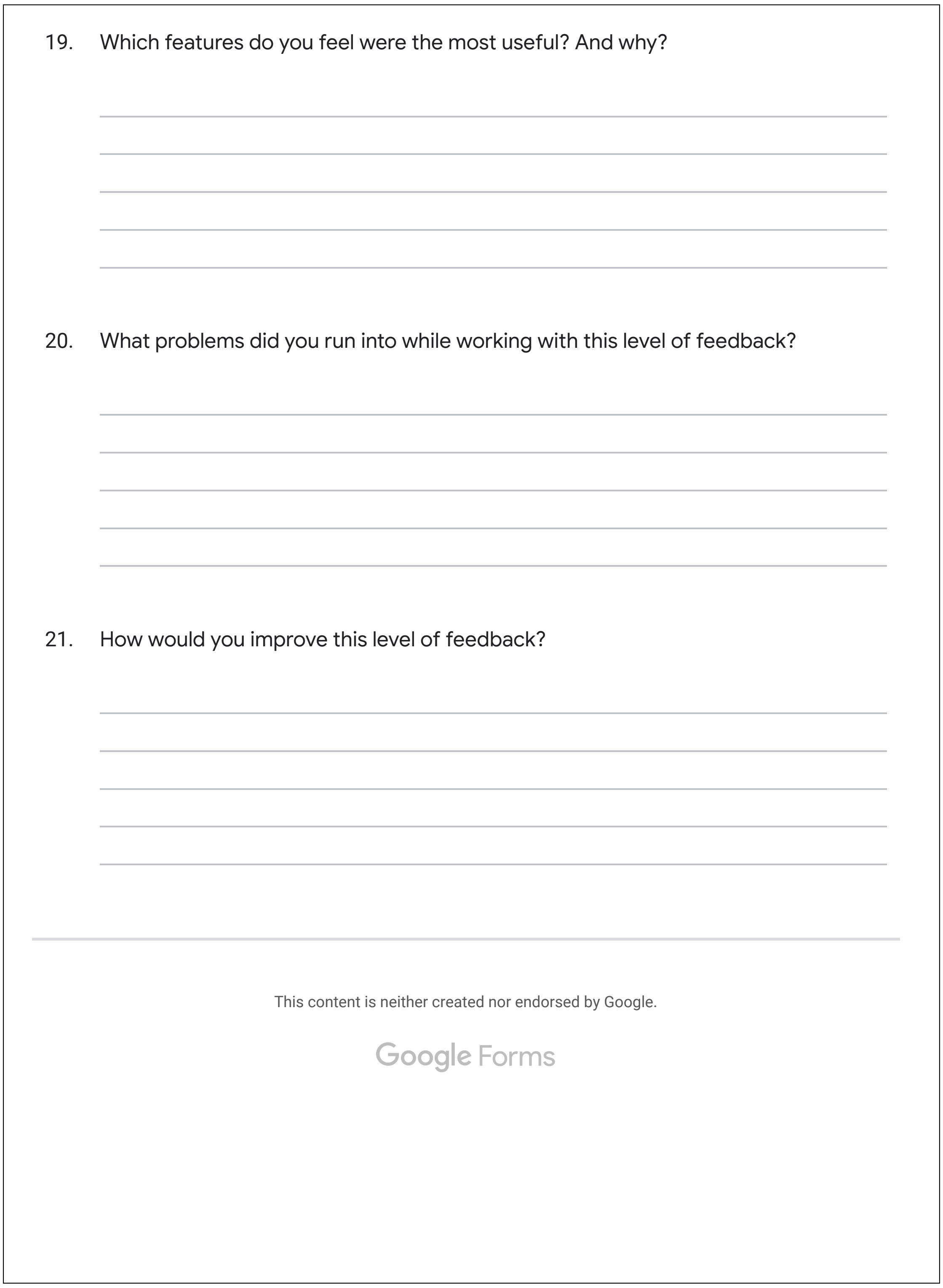
Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...the quantity of information displayed inside VSCode.	◯	◯	◯	◯	◯
...the way the information was displayed inside VSCode.	◯	◯	◯	◯	◯

18. During the execution of the tasks, the feedback provided in the IDE helped me... *

Mark only one oval per row.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...finding out what parent image is the most suitable.	◯	◯	◯	◯	◯
...finding out what are the dependencies of the system that must be added to the docker image.	◯	◯	◯	◯	◯
...finding out what are the Dockerfile commands that I need.	◯	◯	◯	◯	◯
...trying to understand if the resulting container is working as intended (e.g., running commands and tests on the container).	◯	◯	◯	◯	◯
...trying to understand why the resulting container is not working as intended.	◯	◯	◯	◯	◯
...finding out which commands are responsible for the container misbehaviour.	◯	◯	◯	◯	◯
...rebuilding the image and re-running the container to confirm that it is working as intended.	◯	◯	◯	◯	◯



19. Which features do you feel were the most useful? And why?

20. What problems did you run into while working with this level of feedback?

21. How would you improve this level of feedback?