FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Visual Programming Language for Orchestration with Docker

Bruno Piedade

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Filipe Figueiredo Correia, Assistant Professor

Second Supervisor: João Pedro Dias, Guest Assistant

July 31, 2020

Visual Programming Language for Orchestration with Docker

Bruno Piedade

Mestrado Integrado em Engenharia Informática e Computação

July 31, 2020

Abstract

With the widespread of cloud-based infrastructure and microservice architectures along with *DevOps* practices, development and operations have become fundamentally more intertwined. As a response to the ever-changing requirements, a variety of tools emerged and among them, containers, were fundamental in providing a more light-weight alternative to virtualization. Docker is currently one of the most adopted and used solutions for container implementation and management in the software industry and Docker Compose supports the orchestration of multi-container applications via text-based configurations files.

Even though the orchestration of simple architectures may be straight-forward, more advanced concepts such as volumes for persistence storage of data can appear daunting for inexperienced developers. Furthermore, the text-based nature of existing solutions allows naive mistakes and decreases the readability of orchestration configurations as their complexity increases, either in number or heterogeneity of services.

Visual programming approaches have been used to handle working with abstractions for domain-specific or general-purpose programming. We can already find numerous instances of visual programming approaches in the operations field. In particular, some which support Docker Compose orchestration. However, these approaches are incomplete since the adopted visual notations do not fully capture the underlying concepts of Docker Compose and therefore may fail to maximize the potential gains of such an approach. Furthermore, there seems to be a lack of adoption of these alternative approaches as suggested by the results of a survey we conducted to gauge what challenges developers face when working with these technologies.

Thus, we see the definition of a complete visual programming approach for specifying and visualizing Docker-based architectures may provide a higher degree of abstraction with the potential of easing the developer's efforts and reducing the error rate and development time.

To this end, we developed a prototype named *Docker Composer*, functioning as the programming environment for the complete visual approach, by leveraging knowledge from visual programming and model-driven engineering. The prototype addresses the limitations of state-of-the-art solutions by featuring rich visual notations that encompass all of the Docker Compose artifacts and simplifies the management of stacks in a single environment.

The prototype was then used as a means to validate the approach in a controlled experiment conducted among novice developers. The study considered two treatments and aimed to compare the prototype with the conventional toolchain in regards to performance and the perception of ease of use, usefulness, and intention to use. The results indicate that the prototype presents some benefits in reducing the development time and error-proneness, primarily for stack definition activities, and provides a more streamlined development experience, supporting our hypothesis. Furthermore, the participants found the prototype easier to use, considered it useful, and manifested willingness to use it in the future.

Resumo

Com a disseminação da arquitetura baseada em nuvem e arquiteturas de microsserviços, juntamente com as práticas *DevOps*, o desenvolvimento e operações tornaram-se fundamentalmente mais interligados. Como resposta aos requisitos em constante mudança, surgiram várias ferramentas e, entre elas, *containers*, foram fundamentais para fornecer uma alternativa mais leve à virtualização. Atualmente, Docker é uma das soluções mais adotadas e usadas para implementação e gestão de *containers* na indústria de software e Docker Compose suporta a orquestração de aplicações com vários *containers* por meio de ficheiros de configurações textuais.

Embora a orquestração de arquiteturas de baixa complexidade possa ser simples, conceitos mais avançados, como volumes para armazenamento persistente de dados, podem parecer intimidantes para programadores inexperientes. Além disso, a natureza baseada em texto das soluções existentes permite erros ingénuos e diminui a legibilidade das configurações de orquestração à medida que sua complexidade aumenta, tanto em número quanto em heterogeneidade de serviços.

As abordagens de programação visual têm sido usadas para trabalhar com abstrações para programação específica de domínio ou de uso geral. Podemos já encontrar inúmeras instâncias de abordagens de programação visual no campo de operações. Em particular, alguns que oferecem suporte à orquestração com Docker Compose. No entanto, estas abordagens são incompletas, pois as notações visuais adotadas não capturam completamente os conceitos subjacentes a Docker Compose e, portanto, podem não maximizar os potenciais ganhos de tal abordagem. Além disso, parece haver uma falta de adoção destas abordagens alternativas, conforme sugerido pelos resultados de uma investigação que realizamos para avaliar os desafios que os programadores enfrentam ao trabalhar com estas tecnologias.

Desta forma, acreditamos que a definição de uma abordagem de programação visual completa para especificar e visualizar arquiteturas baseadas em Docker pode fornecer um maior grau de abstração com potencial de facilitar os esforços do programador e reduzir a taxa de erro e o tempo de desenvolvimento.

Neste sentido, desenvolvemos um protótipo denominado *Docker Composer*, que funciona como ambiente de programação para a abordagem visual completa, com base em conhecimentos das áreas de programação visual e de engenharia orientada a modelos. O protótipo aborda as limitações das soluções de estado da arte, apresentando notações visuais ricas que abrangem todos os artefactos de Docker Compose e simplifica a gestão de *stacks* num único ambiente.

O protótipo foi então utilizado como meio para validar a abordagem numa experiência controlada realizada com programadores principiantes. O estudo considerou dois tratamentos e teve como objetivo comparar o protótipo com o conjunto de ferramentas convencionais em relação ao desempenho e à perceção de facilidade de uso, utilidade e intenção de uso. Os resultados indicam que o protótipo apresenta alguns benefícios na redução do tempo de desenvolvimento e propensão a erros, principalmente para atividades de definição de *stacks*, e fornece uma experiência mais fluída, o que suporta a nossa hipótese. Além disso, os participantes consideraram o protótipo mais fácil de usar, útil e manifestaram vontade de usá-lo no futuro.

Acknowledgements

I thank my supervisor, professor Filipe Correia and my second supervisor, professor João Dias for their guidance, collaboration, and criticism. Without their knowledge and ideas, this dissertation would have not been the same.

My deepest gratitude to my family for their patience and support in my decisions through all the years. I am who am today because of them.

Lastly, but not least, I thank everyone, colleagues and friends, who accompanied me throughout this academic journey.

Bruno Piedade

“Imagination means nothing without doing”

Charlie Chaplin

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CLI	Command Line Interface
CaaS	Containers as a Service
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
GUI	Graphical User Interface
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
IoT	Internet of Things
LHS	Left Hand Side
M2M	Model-to-model
M2C	Model-to-model
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSE	Model Driven Software Engineering
MW-U	Mann-Whitney U
PIM	Platform Independent Model
PLC	Programmable Logic Controllers
PSM	Platform Specific Model
PaaS	Platform as a Service
QVT	Query/View/Transformation
RHS	Left Hand Side
SaaS	Software as a Service
UML	Unified Modeling Language
VM	Virtual Machine
VMF	Visual Programming Framework
VP	Visual Programming
VPL	Visual Programming Language
YAML	YAML Ain’t Markup Language

Chapter 1

Introduction

The goal of this chapter is to present an overview of this dissertation. It starts with the description of context in which this dissertation fits (Section 1.1), followed by the definition of the problem that it tries to tackle (Section 1.2), the motivation behind the solution (Section 1.3), the proposed main goals (Section 1.4) and contributions (Section 1.5). Lastly, an outline of how the remainder of the dissertation is structured is given (Section 1.6).

1.1 Context

As the user-base and complexity of applications grow the infrastructure upon which they are built grows accordingly. This, in turn, results in huge challenges in managing and scaling such infrastructure. DevOps has emerged in the software engineering ecosystem as a set of practices with the goal of combining development and operations seamlessly, shortening development life cycles, and providing continuous deployment all the while ensuring high software quality [26]. At the same time, cloud technologies have become commonplace, following the advent of cloud service providers such as Amazon Web Services (AWS)[1], Microsoft Azure[2] and Google Cloud Platform[3]

[1]More information available at https://aws.amazon.com/
[2]More information available at https://azure.microsoft.com/
[3]More information available at https://cloud.google.com/

offering unprecedented flexibility and deployment velocity through diverse hosting options suitable to satisfy a broad variety of consumer requirements and needs [7]. Automation is a core principle of DevOps, targeting, among other challenges, infrastructure management [26]. Infrastructure as Code (IaC) is the practice of managing infrastructure through configuration files as part of the code-base [34]. Although initially designed for configuring and managing bare-metal infrastructure (popularized by configuration management tools such as Chef[4] and Puppet[5]), this practice is also used nowadays for managing and provisioning infrastructure resources in cloud environments.

Alongside these developments, another key aspect was the appearance of containers, revolutionizing the way systems are structured and enabling microservices architectures [3]. Containers allow the virtualization of services and applications in a more lightweight way when compared to virtual machines [38]. One of the key container implementation and management technologies is Docker which currently plays a massive role in this context [51]. With the ever-growing scale and complexity of the software engineering world, the need for efficient and useful tools for supporting the developers whose daily job is to manage such systems and optimizing these tasks is critical.

1.2 Problem Definition

Docker Compose[6] is a tool for defining and running multi-container Docker applications. Usually, the developer defines the intended configuration by editing a YAML file containing all the information regarding the services that constitute the application, the corresponding images, and how they are related to each other as well as volumes for data persistence and networks for the connections between services. The stack can then be run, conventionally through the command-line interface (CLI), resulting in the creation or execution of the declared resources, including the container or set of containers for each of the declared services.

Although this process may be fairly straight-forward for the setup of low complexity service stacks, the textual nature of the configuration files may present some challenges as the complexity increases, be it in number or heterogeneity of services. In such cases, it is more clear that understanding dependencies between services becomes difficult as definitions begin to get scattered within the file. Furthermore, some advanced aspects of the configuration such as port mapping and volume management might be overwhelming and confusing for inexperienced users.

A complete visual approach for editing and visualizing such configurations might be valuable in aiding the developer in successfully setting up comprehensive and complex applications by providing a higher degree of abstraction in a friendlier and more intuitive environment. Thus, we believe the increased understandability provided by a complete visual approach [14] may result in higher efficiency, particularly, by speeding up development time and reducing error proneness. In this sense, a complete visual approach has the potential to prove useful for a broad audience of

[4]Chef, available at https://www.chef.io/
[5]Puppet, available at https://puppet.com/
[6]More information available at https://docs.docker.com/compose/compose-file/

end-users ranging from first-time developers who wish to understand how the technology works to more experienced users who might take advantage of the visualization aspects to have a clearer overview of the configuration.

1.3 Motivation

Historically, visual approaches have been used for numerous purposes, for example, in manufacturing industries when configuring programmable logic controllers (PLC) via ladder and sequential function charts and usage of visual notations in software engineering such as the Unified Modeling Language (UML) while more recent applications can be found for educational purposes and in the Internet of Things (IoT) area [23]. In fact, there are already numerous examples of visual approaches found in the operations field for multiple purposes and tasks, for instance, the management of cloud and container resources including some which focus on Docker technologies [46]. In this sense, there is evidence supporting the viability and usefulness of such an approach.

Additionally, there is currently a high concern with misconfigurations resulting from IaC scripts following the widespread of this process to automate infrastructure provisioning and management [53]. Although Docker Compose configurations files do not fit precisely in this definition, they are very similar to this notion, albeit applied to service orchestration. The previously identified benefits of visual approaches can be crucial in alleviating such misconfigurations.

1.4 Main Goals

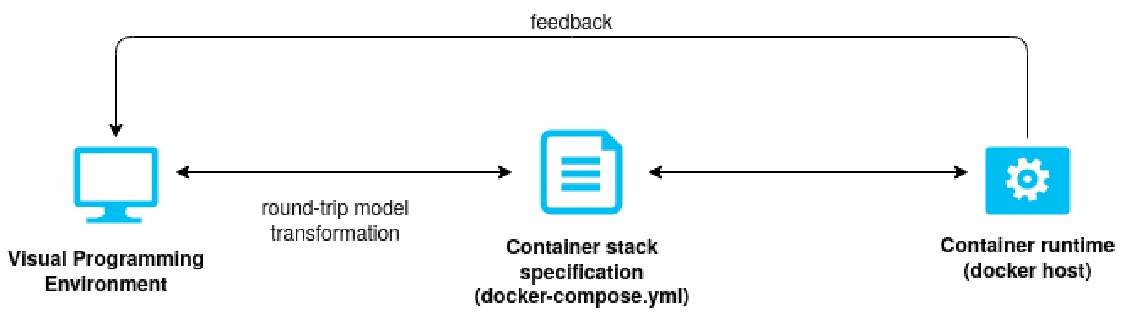

feedback
round-trip model transformation
Visual Programming Environment
Container stack specification (docker-compose.yml)
Container runtime (docker host)

Figure 1.1: High-level architectural overview.

The overall purpose of this dissertation is to explore and research the benefits of a visual programming approach applied to the orchestration of service stacks based on Docker Compose by evolving and expanding the efforts already made in this field. With this objective in mind, we propose to design a more complete visual programming approach including the definition of the associated graphical elements and visual notations for the artifacts and relations inherent to Docker Compose. Next, the development of a prototype for leveraging the language. Finally, we propose an experimental design for the validation and assessment of the practical usefulness of the proposed visual approach using the prototype to conduct experiments with the end-users by measuring the efficiency (e.g. time of development and number of errors made by the developers) between solutions.

Figure 1.1 shows a high-level perspective of the envisioned architecture. As can be observed, the solution will leverage knowledge mainly from two fields: Visual Programming (VP) and Model-Driven Engineering (MDE). Due to this, these areas are thoroughly explored in the following two chapters.

1.5 Contributions

The contributions of this dissertation are threefold:

- **A study of the challenges of working with Docker Compose.** A study conducted with students to identify practical issues developers find when working with Docker technologies;
- **A visual programming environment for orchestration with Docker Compose.** A prototype serving as a visual programming environment was developed leveraging the proposed complete visual approach for orchestration with Docker Compose;
- **An experimental design to validate the approach.** We have designed and conducted a user study among students to empirically validate the complete visual approach.

1.6 Dissertation Structure

In this section, the remainder of the dissertation structure is outlined as follows:

Chapter 2, **Background**, compiles a collection of the most relevant concepts and issues that are essential in understanding the rest of the dissertation.

Chapter 3, **State of the Art**, provides a review of the state-of-the-art related to the context of this work.

Chapter 4, **Preliminary Work**, describes the research conducted to identify the issues developers face when working with Docker Compose technologies.

Chapter 5, **Problem Statement**, presents a focused and detailed view of the problem addressed by this dissertation and solution prospects including the hypothesis and research questions.

Chapter 6, **Solution Prototype**, details the implemented prototype which illustrates the proposed solution to the problem.

Chapter 7, **Empirical Study**, documents the experimental procedure and showcases the findings for the user study conducted to validate the solution prototype.

Chapter 8, **Conclusions and Future Work**, briefly summarizes the main conclusions derived from the research along with the main contributions and proposed future work.

Chapter 2

Background

This chapter has the main purpose of defining key concepts, essential to the understanding of this work and upon which it is built. The concepts are as follows: cloud computing and infrastructure (Section 2.1), virtualization and containerization (Section 2.2), software visualization (Section 2.3, visual programming languages (Section 2.4) and model-driven software engineering (Section 2.5).

2.1 Cloud Computing and Infrastructure

Cloud computing is a broad concept that has risen as a paradigm for hosting and delivering services over the Internet [1]. In more detail, it can be defined as the combination of the *Software as a Service* (SaaS) delivered over the Internet and the hardware and systems software data-centers that provide those services, known as a cloud [7]. According to Abbasov [1], cloud computing has five fundamental characteristics:

- **On-demand self-service:** A user should be able to acquire resources automatically without human interaction with the service provider.
- **Broad network access:** The resources of the cloud should be available through the Internet and accessed through diverse terminals.

- **Resource pooling:** The physical and virtual computing resources (e.g. storage, processing) are abstracted from the end-user and assigned as needed. The end-user has no control and knowledge over the resources and may only be able to specify a location.

- **Rapid elasticity:** Resources scale elastically on demand giving the illusion of being infinite and can be provisioned in any quantity at any time.

- **Measured service:** Resources are automatically controlled and optimized using appropriate metering capabilities and its utilization is transparent for both the provider and consumer.

Another important characteristic of cloud computing is the type of services provided which range in a large spectrum based on resource abstraction [1]. These include the following:

Software as a Service (SaaS). The infrastructure is abstracted from the consumer removing control at this level. It aims to provide a hosting environment suitable for the deployment of complete applications accessible through the Internet from diverse terminals. Management is done in a single virtual environment for the optimization of resources in regards to availability, speed, security, maintenance, and disaster recovery.

Platform as a Service (PaaS). Development platform for the full "Software Lifecycle". It differs from SaaS because it provides support for both complete and in-progress applications.

Infrastructure as a Service (IaaS). Provides the consumer with direct control of resources at the infrastructure level (e.g. processing and storage). Relies heavily on virtualization for integration and decomposition of physical resources to accommodate demand.

Finally, in regards to the deployment model, whose definition is based on the location of the infrastructure and the entity responsible for managing it, this may be one of four [1]:

Private cloud. Operated by a single organization regardless of location. Reasons for adoption include, among others, utilizing and optimizing in-house resources and ensuring data privacy.

Community cloud. Shared between multiple organizations under the same infrastructure and core values. Can be hosted by a third-party or a member of the community.

Public cloud. Most common model. Allows consumers to pay on-demand based on the provider's policies and charging-model.

Hybrid cloud. A combination of two or more of the previous clouds connected by technologies that enable data and application portability.

One of the biggest benefits of cloud computing is its elasticity and pay-by-usage model, which adapts to the needs of its consumers. This provides companies of all sizes the opportunity of experimenting and take risks without the heavy burden of managing all required infrastructure and hardware behind it, shifting the focus from management and maintenance to the core business.

Along with the widespread of cloud computing technologies and concerns nourished by the DevOps community, the Infrastructure as Code (IaC) practice emerged. It is the practice of maintaining system configurations and provisioning deployment environments using source code [34].

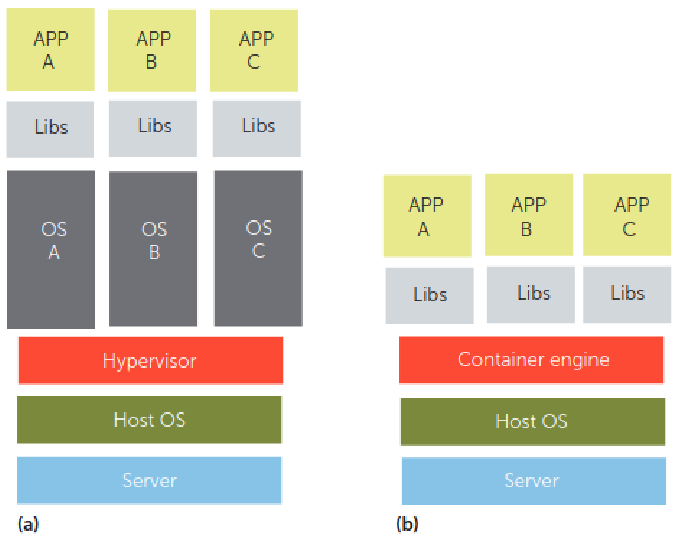


Figure 2.1: Comparison of (a) hypervisor and (b) container-based deployments. From [10]

IaC scripts are handled much like the rest of the code-base and promote the participation of developers in operations.

2.2 Virtualization and Containerization

As established in the previous section, cloud computing presents several difficult challenges for the providers of these technologies. For instance, achieving the proposed scaling elasticity and seemingly infinite capacity requires the virtualization of the resources to hide the implementation of how they are multiplexed and shared [7]. In fact, major aspects are heavily dependent on virtualization. Current solutions include virtual machines (VMs) and containers.

Containers are lightweight and executable standard units of software which package up code and all its dependencies [59]. Although containers and VMs try to solve an identical issue and are indeed very similar, the main difference lies in the virtualization level utilized [48]. VMs aim to emulate hardware while containers emulate operating systems, making them more lightweight and portable all the while keeping the same resource isolation and allocation benefits [38]. In fact, some sources claim the benefits of using containers in comparison to VMs, most notably, in regards to the throughput and response time [40, 38].

Figure 2.1 displays the differences in the deployment of an application between hypervisor (VM) and containers.

It is in this context that Docker currently stands as one of the most adopted container implementation and management technology [51]. Containers play an important role nowadays from development to production and are crucial in enabling microservices architectures [3].

It is for this exact reason why Docker, more accurately Docker Compose, was chosen as the container technology for which to develop the visual orchestrator being proposed in this work,

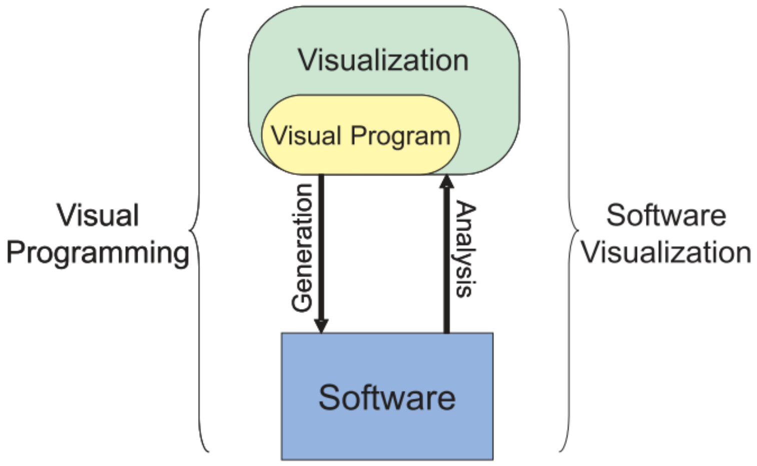


Figure 2.2: Visual programming and software visualization. From [24]

that is, its current high adoption and relevancy in the software development and engineering space. Taking into account existing container technologies, when compared to alternatives, Docker stands as the most pertinent.

2.3 Software and Program Visualization

A substantial amount of research is found in the area of program visualization and the broader area of software visualization. The distinction between both lies in what artifacts are considered. In software visualization, besides the source code itself, other artifacts, namely requirements, architectural design, and bug reports are also taken into account [24].

Software visualization and visual programming (discussed in the next section) are strongly related and complement each other, as shown in Figure 2.2. While the former usually produces static visualizations for software systems the latter allows visual manipulation of elements to generate software systems. When applied simultaneously round-trip visualization is achieved [24].

In general, these visualization approaches have the overall intent of increasing the comprehensibility of software systems by providing a method to visualize the code and its relationships usually following some specific visual metaphor, resulting in a higher abstraction level by translating difficult concepts into more comprehensible equivalents. Furthermore, some explore liveness concepts [62, 2] in an effort to improve the feedback provided to the developer. This approach is particularly useful when applied to complex and large-scale systems for which a global view may be valuable in understanding the system as well as debugging.

One of the most widespread is the Unified Modeling Language (UML). UML has been the *de facto* standard for, among other purposes, visualizing software architectural designs and artifacts [11].

Another example is Cloudcity that aims to explore the benefits of model-driven engineering techniques and a combination of live programming and software visualization approach applied to cloud management through a city metaphor [43].

More software visualization approaches have adopted a city metaphor in the past, following in *CodeCity*'s footsteps, originally developed by Wettel et al. The city metaphor was chosen after several empirical studies validated its adequacy and efficiency in software visualizations. According to this metaphor, software elements are mapped to city elements such as buildings and districts. Additional efforts include CityVR, which experimented with virtual reality (VR) integration to provide a higher engagement level [43, 44, 4].

2.4 Visual Programming Languages

Although the definition of a visual programming language (VPL) can be broad, vague and even somewhat contradictory among distinct authors throughout its evolution, the main agreed-upon characteristics are the reliance on and usage of graphical elements and visual notations, such as icons and diagrams, for both conveying information and serving as the interaction medium with the developer applied to some application, with the overall purpose of providing an abstraction over some programming task. The major distinction between visual and text-based programming languages lies in the exploration of multiple dimensions for semantic expression of the former when compared to the latter [14, 15, 50]. Examples of such added dimensions include time relationships expressing "before-after" relations and spatial relationships.

Even though this definition may apparently imply the elimination of text on the surface, in reality, this is a misconception as the overall goal is to strive for improvements in programming language design in a multidimensional context. In fact, most VPLs include some textual elements as an auxiliary dimension to provide a more complete and comprehensible view. Nevertheless, to be classified as such, VPLs require significant parts of the program structure to be represented graphically [14, 25].

2.4.1 Key Concepts

We'll start by presenting and discussing some key concepts, characteristics, and features which are shared across literature.

Purpose. VPLs are split between two purposes. *General-purpose* providing a visual approach for software development, suitable for producing executable programs of reasonable size, with equivalent freedom of a high-level text-based programming language and *domain-specific* when applied to a specific area for a single or set of tasks (e.g. software engineering or scientific visualization).

Visualization metaphor. A crucial characteristic of a VPL is its adopted visualization metaphor, that is, how the domain concepts are mapped from the internal model to their corresponding visual representations [8, 24]. The overall goal of applying a visualization metaphor is to increase comprehensibility over otherwise foreign and difficult notions by transposing these concepts into more understandable analogous concepts which are more familiar to the end-user. One example is the city metaphor used for code visualization, mentioned previously.

Representation model. This is a broad topic, ranging from dimensionality, whether 2D or 3D, to how the elements are distributed, organized, and connected on the screen, such as graph-based and box-based diagrams.

Control flow. As in traditional text-based programming languages, VPLs follow one of two concepts for flow of control: *imperative* and *declarative*. In a *imperative* approach one or more control-flow or dataflow diagrams are used to convey how the thread of control flows through the program, being particularly useful for representing parallelism. On the other hand, in a *declarative* approach, the developer is only concerned with what computations are performed and not how the operations are carried out, avoiding explicit state modifications by the use of single assignment.

Abstraction. Two abstraction types are widely adopted: *procedural abstraction* and *data abstraction*. On one hand, in regards to *procedural abstraction*, this type can be further subdivided into two levels, high and low level. High-level VPLs are commonly found in domain-specific applications but are not complete programming languages, meaning, it is not possible to write and maintain an entire program from the ground up with these languages without the combined use of additional underlying non-visual modules. In contrast, low-level languages, do not allow the developer to combine fine-grained logic into procedural modules and are useful in domain-specific applications as well. General-purpose VPLs usually feature both levels of abstraction to best suit the distinct concepts of a general programming language. On the other hand, *data abstraction* is exclusively applied in general-purpose VPLs and its definition is very similar to the one when applied to conventional programming languages, that is, the idea of simplifying a body of data into a reduced, yet more comprehensible, representation. However, when applied to the concept of a VPL, this type of abstraction includes the added restrictions of being defined visually, have a visual representation, and provide interactive behavior.

As previously established, the main purpose of VPLs is to provide an abstraction over some programming tasks and allow the developer to work at this higher abstraction level, theoretically presenting benefits when compared to traditional text-based programming. In particular, according to Burnett [14], the most common goals include increasing the understandability for a certain target audience, reducing error proneness, and increasing the development speed. This can be achieved by exploring four common strategies:

- **Concreteness.** Allow the direct and visual exploration of data. One example is the effects of some portion of a program being automatically displayed on a specific object or value.

- **Directness.** Reducing the number of steps between the intention and the goal. In practice, it equates to minimizing the sequence of interactions required for the user to achieve some objective. One example is the difference between having to navigate through multiple menus in comparison to using a keyboard shortcut for the same task.

- **Explicitness.** Explicitly represent more data without the need of inferring it. For instance, considering a graph-based visual notation and 3 elements a, b and c which follow a transitive

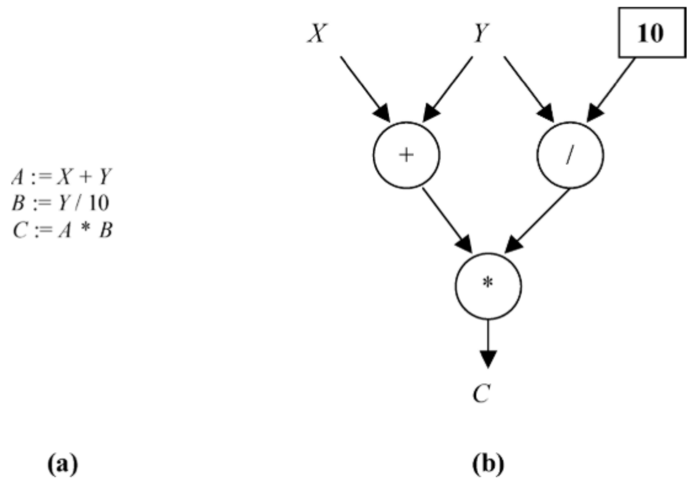


Figure 2.3: A program and its dataflow equivalent. From [37]

relation R such that if *a* R *b* and *b* R *c*, then *a* R *c*, the distinction lies between explicitly displaying the relation between **a** and **c** and not.

- **Immediate Visual Feedback.** Automatically display up-to-date information whenever some change occurs. This refers to the feedback loop between the programming environment and the developer, therefore, it can be directly associated with liveness levels as defined by Tanimoto [62] according to Johnston et al. [37].

2.4.2 Categories

A very important subset of VPLs is those which follow the Dataflow Model [31, 37]. Its origin dates back to the 1970s and was originally intended to exploit parallelism [37]. In this model, a program is seen as a directed graph in which nodes correspond to primitive instructions or functions (e.g. arithmetic operations) and arches represent data dependencies between the instructions upon which units of data called tokens flow behaving as unbounded queues. Inbound arches correspond to the input of a function, while outbound correspond to the data output. When all of its inputs contain data, a function node, considered as *fireable*, is executed some undefined time afterward and the output is placed in some or all of its outbound arches. The main advantage of this model lies precisely in its parallelism potential as multiple nodes can be fireable simultaneously. Programming languages which follow this paradigm usually adopt a graph-like representation. Figure 2.3 showcases a program snippet (a) and its equivalent in a dataflow notation (b).

During the 1990s, a surge of VPLs appeared [37] and its impact can still be felt today, as seen with many recent VPLs, built for a wide array of tasks and domains. The representation as a graph is very flexible and a fitting way of visually representing many concepts making it useful in many applications whether for general-purpose purposes and domain-specific languages.

Throughout the evolution of VPLs, multiple classification schemes have been proposed. One of the most recent is the scheme proposed by Boshernitsan et al. [12] which results from the combination and refinement of previous efforts made by important authors in the field, such as Chang [20], Shu [58], and Burnett [16]. This scheme will be adopted henceforth for all classifications of VPLs made during this work, particularly in the state-of-the-art review (Chapter 3). According to these authors, based on their characteristics, including provided features and used paradigms, VPLs can be organized in five non-mutually exclusive categories. Most VPLs belong to one of the first two categories and both general-purpose and domain-specific languages can be found in each category.

Purely visual languages. These VPLs are characterized by their strong reliance on visual techniques, that is, interaction through the manipulation of visual elements. Additionally, they allow direct compilation from the visual model, without the need for translation to some intermediate text-based language and support debugging and execution in the same environment.

Hybrid text and visual languages. Mix of text-based and visual languages. Considers both languages which are fundamentally text-based but include complementary graphical elements and fundamentally visual-based (similarly to purely-visual) but are afterward converted into a high-level text-based language. Furthermore, some support edition of a program while alternating between two views for the visual and textual representation while maintaining consistency between both.

Programming-by-example systems. Follows the *programming-by-demonstration* paradigm allowing the user to create and manipulate visual objects to *teach* the system how to perform tasks. This category considers two types: *Programming by example* systems that try to infer the program from examples of input and output and *Programming with examples* systems that remember programmer's commands for later use, without inferring anything [50].

Constraint-oriented systems. Act on constraint scenarios or environments by the definition of a set of rules. Examples include simulation design and graphical user interface (GUI) development.

Form-based systems. Derived from spreadsheets for its visualization and programming paradigm. Programming is done by changing a set of interconnected cells over time. Although not explicitly specified in the definition by Boshernitsan et al., this category can be seen in a more broad perspective to include languages which follow the *Form-based and spreadsheet-based* paradigm in general [16]. This means that, besides languages that adopt a spreadsheet metaphor, similar languages that generalize sheets into forms are also considered.

To more clearly illustrate the characteristics and concepts inherent to each VPL category as established in the scheme proposed by Boshernitsan et al. [12], some examples are briefly discussed. To note, even though much work can be found during past decades towards compiling surveys in regards to this area, to the best of our knowledge, there is currently a lack of up-to-date examples covering more recent advances in this field. As a result, the VPLs presented in this section were chosen based on their popularity and relevancy. In particular, one or a combination of ranking

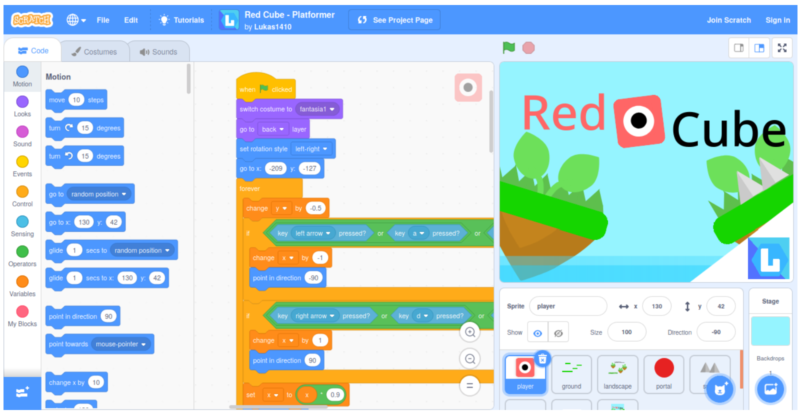


Figure 2.4: Sample of the Scratch programming environment UI. From the project Red Cube available at https://scratch.mit.edu/projects/361704410

metrics on the hosting website, such as Github[1]'s star ranking system, when such a metric was available. Otherwise, the criteria for the inclusion of the tool was the novelty of its approach or metaphor adopted. In addition, some grey literature was also considered for review.

Purely visual languages. Scratch[2] is an educational visual programming environment that lets users create interactive and media-rich projects, primarily aimed at teaching young people programming concepts [45]. It was developed by the MIT Media Lab and publicly launched in 2007. It is built upon a block-like interface that allows the connection of command blocks to express the program's logic. Each block corresponds to some operator. The programming environment consists of a single-window split into four main panes: a command palette containing the available command blocks, a pane for the scripts, a staging area for the output, and a pane for all the sprites in the project. It also features additional panels accessible by folder tabs to view and edit the costumes and sounds owned by the selected sprite. It achieves level 4 liveness, allowing the users to interact with the system at any time while it is running, encouraging *tinkerability*, in other words, experimentation and self-discovery similar to how one may tinker with mechanical or electronic components [45]. Figure 2.4 displays a sample of the UI.

Hybrid text and visual languages. Blockly[3] is JavaScript library for VPL based on interlocking logic blocks similar to the previously mentioned Scratch. However, it matches more closely the syntax of traditional programming languages supporting common constructs such as if conditions and loops. It can output the corresponding code in multiple languages including JavaScript and Python. Many projects are built with Blockly mostly for educational purposes. One example is **Micro:Bit**[4]. This project introduces a micro-computer programmable through the block notation or with a traditional scripting programming language, namely JavaScript and Python. Most notably, the web programming environment offered supports programming between the block-based

[1]https://github.com/
[2]Scratch, available at https://scratch.mit.edu/
[3]Blockly, available at https://developers.google.com/blockly
[4]Micro:Bit, available at https://microbit.org/

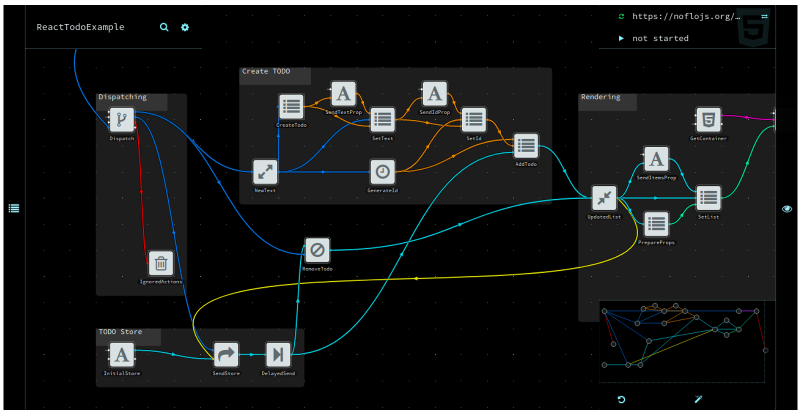


Figure 2.5: Sample of the FlowHub programming environment UI. From the React ToDo example available at `https://noflojs.org/example/`

notation and the traditional text-based language in parallel for the same source code.

Another notable example is **NoFlo**[5]. NoFlo is an open-source JavaScript implementation of Flow-Based Programming (FBP), a programming approach reminiscent of dataflow programming [37, 49]. It is designed for general-purpose programming and applications can be found in various domains and purposes, ranging from IoT to multimedia and even controlling a drone. Figure 2.5 showcases an example of NoFlo's FlowHub environment UI. In regards to the UI, some noteworthy design considerations include:

1. Customization of blocks through user-defined icons and names as well as descriptions for inputs and outputs, making the concepts more comprehensible, especially when interpreting an unknown project for the first time. However, even though a more detailed description for a component can be set in its definition, this description is not displayed in the corresponding details panel.

2. Inclusion of a simplified mini-map for displaying an overview for the complete layout of the diagram.

Finally, **Node-RED**[6] is one of the most widespread tools used in IoT, originally developed by IBM[7] and currently part of open source OpenJS Foundation[8]. It works as a Flow-Based Programming tool for wiring together hardware devices, APIs, and online services. Created flows can be exported and are stored in JSON format. Figure 2.6 displays an example of a flow.

Programming-by-example. Pygmalion is usually considered as one of the first languages in this category [12]. Additional examples include Chimera, Cocoa, and Rehearsal World [12, 14, 50].

[5]NoFlo, available at `https://noflojs.org/`
[6]Node-RED, available at `https://nodered.org/`
[7]`Moreinformationavailableathttps://www.ibm.com/`
[8]`Moreinformationavailableathttps://openjsf.org/`

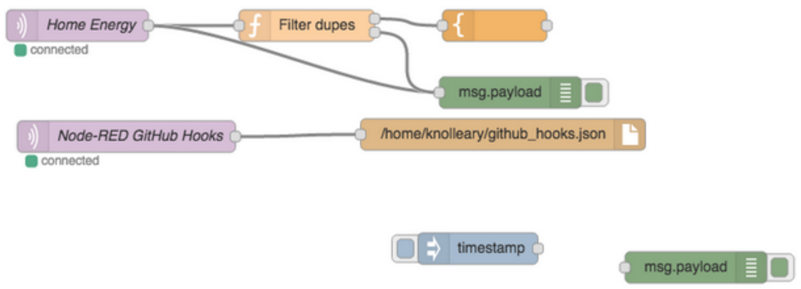


Figure 2.6: Example of a Node-RED flow. Retrieved from Github https://github.com/node-red/node-red

Based on the conducted research and to the best of our knowledge, no recent VPLs can be found in this category.

Constraint-oriented systems. From a historical perspective, these systems have been mostly used for performing simulations, quintessential examples being ThingLab and ARK, and developing GUIs [12]. A modern example is **Bubble**[9] designed for enabling non-programmers to build web applications. This tool includes, among other aspects, a GUI builder usable through a combination of drag-and-drop mechanisms and form-based definitions.

Another recent example is **IFTTT**[10], a web-based service for connecting other web services and some physical devices through simple conditional statements called *applets*. Common services include Gmail, Facebook, and Instagram. When creating an *applet*, the user configures a service and the trigger condition as well as the desired action to perform on the target service. *Applets* can be shared among users of the application.

Form-based. Forms/3 is one of the progenitors in this category [12]. This general-purpose VPL follows the spreadsheet metaphor of cells and formulas to represent data and computation respectively [12].

To summarize, in the current landscape, among the visual programming language categories, purely-visual and hybrid constitute the vast majority and most are domain-specific languages rather than general-purpose. This is directly in line with practical experiences that support the hypothesis that visual approaches are most useful when applied to a particular field since it takes advantage of a clearer and direct communication style tailored for specific problems [14]. By a large margin, the most common visualization notations were some variation of a graph-based approach. Moreover, recent advances in the education field have thoroughly explored block-based representations, partially due to open-source libraries such as Blockly that facilitate the development of tools in this programming style.

[9]Bubble, available at https://bubble.io/
[10]IFTTT, available at https://ifttt.com/

2.5 Model-driven Software Engineering

Model-driven Software Engineering is a software engineering paradigm in which the whole development life-cycle is supported by high-level abstraction (models) representing different views of the system, abstracting the underlying infrastructure and code. These models can be converted in a (semi)automatic process (model transformation) between distinct abstraction levels from an initial high-abstraction model down to its executable equivalent [22]. The potential benefits of this approach include:

- **Improved portability** by separating the application knowledge from the specific implementation technology.
- **Increased productivity** through automated mapping.
- **Improved quality** through the reuse of proven patterns and best practices.
- **Improved maintainability** due to a better separation of concerns.
- **Better consistency and traceability** between models and code.

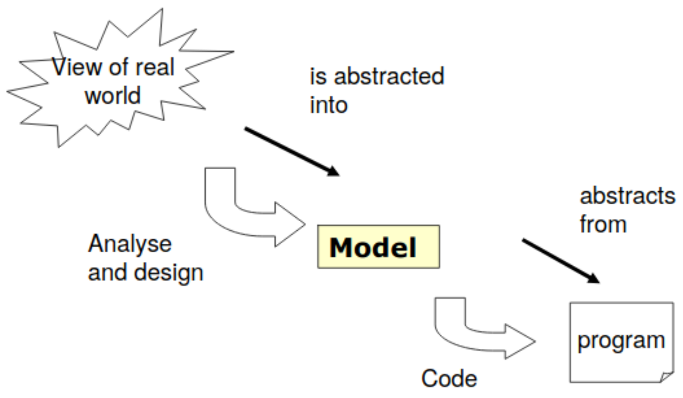


Figure 2.7: Diagram of models in software engineering. From https://researcher.watson.ibm.com/researcher/files/zurich-jku/mdse-01.pdf

Figure 2.7 diagrams the concept of models in software engineering showcasing how a model abstracts low-level concepts (usually code) but itself is an abstraction of a view in the real world.

Numerous initiatives can be found in the domain of MDSE. One of the most notable is Model Driven Architecture (MDA) by the Object Management Group (OMG)[11]. The MDA standard specifies the usage of the MetaObject Facility (MOF) language for modeling high-level Platform-Independent Models (PIMs) which are mapped to low-level Platform Specific Models (PSMs) used to generate artifacts usually code [28]. MDA considers a set of MOF-compliant transformation languages such as the Model to Text Transformation (MOFM2T) language to enable

[11]More information available at https://www.omg.org/

model-to-code transformations and the Query/Views/Transformation (QVT) standard for model-to-model transformations [39]. The QVT standard specifies three languages: the declarative QVT Relation (QVTr) to design relations between models, the imperative QVT Operation (QVTo) to write unidirectional transformations and the lower-level declarative QVT Core (QVTc). QVTr is a higher-level language built on top QVTc [39].

OMG adopts a four-layered architecture based on model abstracting level ranging from layer M0 to layer M3 [42]. M0 corresponds to actual real-world objects, that is, instances of business objects; M1 is the model which abstracts the instances of M0; M2 to the meta-model to which M1 conforms and finally M3 as the meta-metamodel that defines the concepts in M2 [13].

As defined by Kleppe et al. [42], "A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

A transformation rule consists of two parts: a left-hand side (LHS) which accesses the source model and a right-hand side (RHS) which expands in the target model. Both sides can be represented using any combination of three approaches: (1) *variables*, sometimes referred to as *metavariables*, hold elements from the source and/or target models; (2) *patterns*, model fragments with zero or more variables and (3) *logic*, computations and constraints on model elements.

At the top level, two types of transformations can be identified: *model-to-model* (M2M) and *model-to-code* (M2C), more accurately *model-to-text* since non-code artifacts may be generated, although both terms are often used interchangeably [22]. Model-to-code transformations, in particular, can be seen as a special case of a model-to-model transformation for which a metamodel for the target programming language is provided and are in general very similar to what compilers perform when translating a high-level programming language into a lower-level equivalent. Although both types are identified, in actuality, based on the definition of a model, M2C is, in fact, a subset of M2M transformations, as code can be seen as a model itself, although at a lower abstraction level closer to machine operations [47].

Furthermore, Cabot et al. [18] identified a set of properties to determine whether a transformation behaves as a mathematical function. In particular, if a transformation is executable, total, deterministic, functional, exhaustive, injective, and bijective. Furthermore, these properties, along with others, are also applicable to individual rules.

Czarnecki et al. [22] proposed a taxonomy for the classification of model transformation approaches along with some applications exemplifying practical instances for each in the context of MDA. This taxonomy will be adopted during this work for the state-of-the-art review on these techniques and is briefly described below.

In regards to model-to-model, five approaches were identified: direct-manipulation approaches, relational approaches, graph-transformation-based approaches, structure-driven approaches, and hybrid approaches.

- **Direct-manipulation.** Is the most low-level approach. Offers an internal model representation and some API to manipulate it and requires the users to implement transformation rules and scheduling mostly from scratch using a programming language such as Java.

- **Relational.** Defines declarative constraints with executable semantics between the source and target models, similar to logic-programming.
- **Graph-transformation-based.** Operates on graphs specifically design to represent UML-like models. Rules are defined using graph patterns for both the LHS and RHS.
- **Structure-driven.** Split into two phases: (1) creating a hierarchical structure of the target model and (2) setting attributes and references in the target. Scheduling and application strategy are defined by the framework and rules cannot have side-effects.
- **Hybrid.** Combines techniques from previous approaches.

In summary and a less granular view, we find three main approaches: *declarative* and *imperative*, which function similarly to how traditional programming languages classified in these paradigms behave, and *graph-based* which operate on graph-based models through graph algebra [39]. Graph-based approaches are in general more complex but provide more flexibility, useful in solving some complex problems such as bidirectional transformations.

In regards to model-to-code transformations, two main approaches were identified: visitor-based and template-based.

- **Visitor-based.** Similar to direct-manipulation approaches in the sense that a simple visitor mechanism is provided to access and traverse the internal representation of the model to generate text for the source model components.
- **Template-based.** Templates are used, usually consisting of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion. The LHS logic can either be direct manipulation of an API or declarative queries. Usually offers user-defined scheduling. Because textual templates are independent of the target language, the generation of any textual artifacts is simplified. Additionally, templates promote re-usage and are in general more intuitive since their definition closely matches the structure of the resulting artifact.

The majority of the available MDA tools support template-based code generation [22]. Code generation is usually unidirectional, meaning that, no way to reverse-engineer the code and synchronize with the model is provided. As a result, tools frequently warn the developer that code is overridden on generation [33]. Moreover, some provide custom code protection measures such as the separation between implementation edited by the developer and interfaces exclusively generated by the tool [39]. In comparison, visitor-based approaches are more basic and highly rely on the programmer to formulate the complete logic behind code generation.

Chapter 3

State of the Art

This chapter is focused on reviewing and describing the existing state of the art relevant to the context of the problem. The goal is to grasp the current landscape to not only understand what solutions and approaches currently exist in the domain but also to extract knowledge and inspiration for the proposed solution.

More precisely, this review is focused on achieving two primary goals: examining some transformations techniques utilized in MDSE as well as their practical applications (Section 3.1) and surveying existing visual solutions mainly within the Information Technology (IT) operations field and in a broader sense DevOps, with a heavy focus on those concerned with services management, particularly those which rely on or are built for Docker technologies (Section 3.2). Both objectives contribute towards a clearer vision and understanding of what the current landscape is for the two core areas from which knowledge will be leveraged for the solution. A summary is included at the end of each section, presenting an overview, discussion, and main conclusions regarding the previous analysis.

Both peer-reviewed publications and gray literature were considered for review.

3.1 Model-driven Software Engineering

This review covers the state of the art in regards to MDSE transformation techniques and how these techniques are employed in practical cases. The former is useful to support informed implementation decisions by leveraging the accumulated experience in the field while the latter offers insight into what the latest relevant practical contributions in the software industry have been as well as what are the current concerns among the academic community.

3.1.1 MDSE Transformations

In this section, we explore some practical examples of transformations techniques utilized in MDSE. The purpose of this review is to understand what the most appropriate approach or combination of techniques may be for the bidirectional transformation between the internal source-code model of a Docker Compose configuration and its YAML equivalent. Therefore, this review tries to answer the following question:

What approaches exist that can support a round-trip model to code transformation?

This issue is related to the implementation of the import/export feature as well as the syncing mechanism between the two representations expected in the proposed solution. With this purpose in mind, we will primarily focus on M2C transformation approaches, since this is where the central issue lies.

We begin by reviewing practical tools for M2C approaches as identified by Czarnecki and Helsen [22] to more clearly illustrate how tools cope with these issues. Unfortunately, many of the examples have been discontinued and its documentation is no longer publicly available. Therefore, at least for those, the only reference material left is what is presented in the original work and other contemporary papers since no method is available to empirically test the tools. However, in 2015, Kahani and Cordy [39] performed a comprehensive survey on MDE enabling tools outlining the major differences between them.

Visitor-based. An example is Jamda[1]. This framework provides a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism to generate code. Additional model element types can be introduced by subclassing existing Java classes [22]. In a regular workflow, users use UML models usually in the XMI format most frequently manipulated with some UML modeling tool such as MagicDraw[2] or Modelio[3]. Currently, the only supported language for the generated output is Java.

Template-based. Many tools can be found which adopt this approach. One example is AndroMDA[4] which uses this approach for code generation. It supports both .NET and Java as output languages.

We will now focus on reviewing transformation approaches to enable round-trip synchronization through, among other tactics, bidirectional transformations (BX). The study of this subject expands beyond the scope of MDSE since these mechanisms are useful in solving problems found in numerous other areas including databases and programming languages [21].

Initial efforts include work by Stevens [61]. The author discussed how bidirectional transformations could be achieved using the QVT standard, more specifically QVT Relational, including the proposition of a framework for "coherent transformation". In the study, the author considers transformations that are bidirectional but not necessarily bijective. Transformations are specified

[1]Jamda, available at http://jamda.sourceforge.net/
[2]MagicDraw, available at https://www.nomagic.com/products/magicdraw
[3]Modelio, available at https://www.modelio.org/
[4]AndroMDA, available at https://www.andromda.org/

in an appropriate language (e.g. QVTr) and can be interpreted both as the relation between two models and as forwards or backwards transformations. A set of assertions are considered: (1) transformations should be deterministic, (2) a transformation may depend on the current value of the target and source models which will be replaced, reinforcing the notion that transformations may not be bijective and (3) transformations have to be total. This resulted in the following definition:

"Let R be a transformation between metamodels M and N,consisting of a relation $R \subseteq M \times N$ and transformation functions $\overrightarrow{R} : M \times N \longrightarrow N$ and $\overleftarrow{R} : M \times N \longrightarrow M$. Then R is a coherent transformation if it is correct, hippocratic and undoable."

In 2008, Angyal et al. [5] proposed a mechanism to maintain round-trip synchronization for M2M and M2C transformations. Our interest lies primarily in M2C, that is, in the context of the proposed solution, the transformation between the PSM, stored in memory as an Abstract Syntax Tree (AST)), to code (e.g. some `docker-compose.yml`, which also has to be periodically parsed and converted to an AST). Two ASTs have to remain in memory at all times, the internal PSM (M) and the last synchronized code state (C_0). The synchronization process, for a given synchronization step, is achieved as follows:

1. The difference between C_1 and the current code AST (C_0), generated from the current state of the code file, is computed (ε_1).
2. The difference between C_0 and M is also computed (D_2).
3. ε_1 is atomically propagated to M and ε_2 is atomically propagated to C_1, achieving consistency.
4. C_0 is updated to store C_1.

In 2010. Hidaka et al. [30] were among the first to propose a solution for bidirectional transformation applied to graphs based on UnCAL, a graph algebra. The solution explores trace information to achieve well-behaved transformations (i.e. consistent transformations in both directions). A year later, the authors expanded upon their work by developing the framework GRoundTram, leveraging the achieved solution [29].

Later, in 2015, Hoils et al. [33] explored the concept of higher-order transformations (HOTs) to address the BX problem. The result was a framework based on bidirectional higher-order transformations (B-HOTs). The framework is independent of the transformation language used due to the usage of binding specifications.

3.1.2 Model-driven Cloud Infrastructure

Cloud computing has been the target of numerous model-driven approaches following its massive proliferation and evolution which promoted the need for new and better solutions. Such approaches have surfaced in an effort to improve the way resources are managed in cloud environments by working at a higher abstraction level provided by high-level models as an alternative or complement to conventional low-level configurations.

In 2012, Ardagna et al. [6] proposed the initial vision of MODAClouds, a framework to tackle the challenges inherent to the lack of interoperability between cloud service providers and resulting lock-in to some provider. The goal was achieving a service-agnostic approach fit for running applications in cross-provider multi-clouds settings, appropriate for strict high availability and flexibility non-functional and business requirements. To this end, a model-driven approach was proposed, allowing the design of software system model artifacts at distinct abstraction levels which are ultimately transformed into code and automatically deployed in the target cloud platforms. Furthermore, it leveraged common cloud design patterns and best practices to aid the users in supporting decision making during the modeling process by offering mechanisms to measure and check if the implementation satisfies the requirements and optimizes target cloud environment selection based on the characteristics of the application.

In 2014, Ferry et al. [27] tackled the challenges of interoperability in multi-cloud settings as part of the same programme as MODAClouds. The result was the Cloud Modelling Framework (CloudMF), a model-driven approach for managing applications in multi-cloud environments tailored with DevOps principles in mind. The approach was built on two components: the DSML *CloudMl* to model the applications and the run-time environment *models@run-time* to manage the systems.

In the end, this effort became part of the MODAClouds initiative along with two other sibling projects PaaSage and ARTIST. PaaSage adopted the Cloud Application Modelling and Execution Language (CAMEL) which integrates and extends existing Domain Specific Languages (DSLs) including CloudML [55].

Also in 2014, Bergmayr et al. [9] explored a similar idea, although less ambitious in scope, when proposing the cloud-specific extension to UML's deployment language named Cloud Application Modeling Language (CAML) tailored towards representing concepts in this domain developed in the context of the previously mentioned ARTIST project. The resulting extension follows MDA's principles by separating between cloud provider-independent and cloud provider-specific deployment models, matching the PIM and PSM respectively. Working prototypes were developed for Eclipse[5] and Enterprise Architect[6]. The authors also argue in favor of the usefulness of CAML for blueprints definition, taking advantage of UML's template reusability. When compared to CloudML, CAML does not include a run-time component to effectively deploy the modeled applications.

Regarding more recent advancements, Sandobalin et al. [56] proposed ARGON (An infRastructure modellinG tool for clOud provisioNing) in 2017, a DevOps support tool leveraging IaC practices and its benefits for infrastructure provisioning while simultaneously minimizing its drawbacks through a model-driven approach. The project defines a DSL to model infrastructure inde-

[5]More information available at `https://www.eclipse.org/`
[6]More information available at `https://sparxsystems.com/`

pendently of the provider, later used to generate the target tool script. It builds upon previous efforts (such as the mentioned [9]) but innovates by operating on top of existing Configuration Management Tools (CMTs) such as Chef and Puppet, creating a toolchain in which pipelines can be configured to automatically update on change. Afterward, in 2018, the authors proposed an extension to the DSL to support a more comprehensive set of resources.

In a follow-up work [57], the authors compared the practical effectiveness of ARGON in comparison to the well-established scripting-based Ansible[7]. In this study, the authors conducted a series of experiments with 67 Computer Science students. In summary, the results show that ARGON was more effective for specifying cloud infrastructure resources and perceived as "easier to use and more useful". Overall, the tool proved to be a success and the authors believe in its usefulness for real-world applications in the industry.

Model-driven approaches have also been applied directly in the context surrounding Docker containers. Paraiso et al. [52] explored the applicability of a model-driven approach to address some of the drawbacks of these technologies especially in regards to, among others, deployment processes and maintenance in production environments. The result was a modeling tool built in a 3 component architecture: (1) *Docker Model*, a model for containers and their dependencies, (2) *Executing Environment*, the target infrastructure hosting the containers and (3) the *Connector* working as the bridge and providing synchronism mechanics between the previous two. An Eclipse-based prototype was developed named *Docker Designer*, featuring a GUI to work with these technologies.

3.1.3 Discussion

To define what the most appropriate transformation approach should be, it is imperative to first clearly state the requirements. These are:

- **Bidirectional transformations.** The internal model should be transformable to text (`docker-compose.yml`) and changes made to the file are propagated back to the model so that both are synchronized (i.e. allow round-trip).
- **Technological.** If some approach requires additional code or libraries (e.g. the need for a templating engine in template-based methods) or is exclusively tied to some specific technology.

Both M2M and M2C approaches are viable, but in M2M a metamodel for the target language, that is the YAML-based definition of a docker-compose, must be provided.

Based on the previously analyzed efforts, there are multiple possible approaches to achieve the expected synchronism. In particular, we find tree-based and graph-based approaches according to the internal structure of the models. Taking into account that `docker-compose.yml` files (and YAML files in general) are structured as trees, it only seems natural to adopt a tree-based approach. To this end, the algorithm proposed by Angyal et al. [5] appears to be an adequate fit. Both

[7]Ansible, available at `https://www.ansible.com/`

the internal model and `docker-compose.yml` (after being parsed) can be represented as trees. These trees can be manipulated in conjunction since both operate at a similar abstraction level. Therefore, the complexity is offset to calculating the difference between the trees as described in the algorithm. Code generation is somewhat trivial and can be achieved through a template-based or direct manipulation technique. A foreseeable challenge, however, is maintaining synchronism between the additional meta-information in the internal model relative to visualization details (e.g. the relative positioning of elements) and making this data persistent. This means that the PSM has additional details that are not considered in the code.

In regards to applications of model-driven approaches, the bulk of the initial efforts were aimed at resolving the vendor lock-in issue and facilitating work in multi-cloud settings. More recently, contributions by Sandobalin et al.[56, 57] combine automation of IaC with high abstraction level programming. This approach is very much in line with what is being proposed in this dissertation where the goal is to provide a higher-level abstraction perspective to orchestrate services built upon IaC like scripts (i.e. `docker-compose.yml`).

3.2 Visual Approaches in Operations

In this section, we present a review of visual tools geared towards IT operations and management tasks, mostly focusing on container technologies, in particular, Docker technologies. The motivation behind this review is primarily to understand what state-of-the-art visual approaches exist in this field and how they help users perform the intended tasks. Of most interest, to explore the tools that adopt a visual approach to orchestrate Docker service stacks similarly to the approach being proposed in this work. Therefore, the core question the review tries to answer is:

"What visual approaches are available to orchestrate and manage Docker service stacks?"

However, in actuality, the considered scope is extended beyond tools that directly work with Docker technologies. This is because other visual solutions can be found in the broader field of operations which try to solve comparable challenges through similar visual approaches. For example, visual orchestrators for managing and provisioning cloud infrastructure resources.

Based on their purpose and subject matter, we can identify three major non-mutually exclusive technology groups: **monitoring**, provisioning and management of **services** and of **infrastructure**. Figure 3.1 displays a Venn diagram demonstrating how some of the surveyed technologies are distributed among the three groups. The diagram includes only some examples for each category. The tools that are displayed were chosen because they comprise a good representation of their corresponding group. Each category is discussed in further detail during the following sections, including a comprehensive table featuring all analyzed tools at the end of each section.

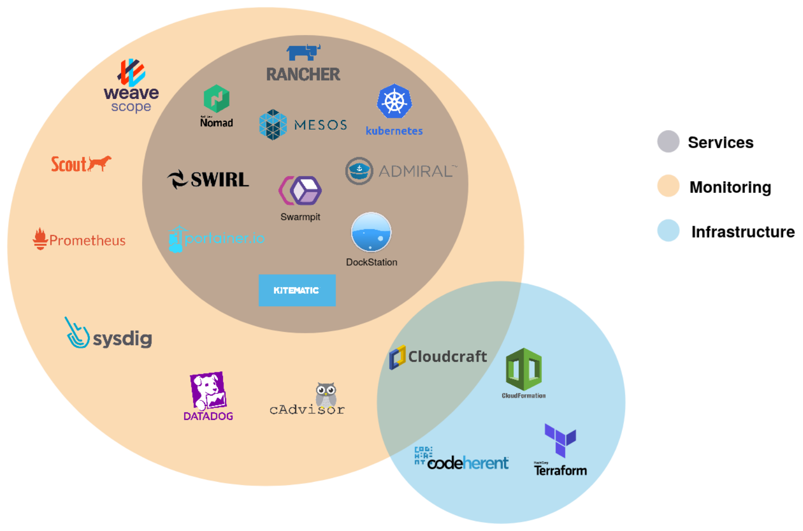


Figure 3.1: Venn diagram of monitoring, infrastructure management and service management technology groups in operations

3.2.1 Monitoring

A large number of monitoring tools, 29 in total, were identified during the review. The research approach was focused on searching for tools with container monitoring support. Out of the total, some of the solutions found were not considered for review because either a) were simple tools focused on a single task (e.g. scheduling alerts), b) featured minimal container monitoring support or none at all or c) only provided a technology stack composed of other technologies for convenience. In the end, 12 tools were considered and are showcased in full in Table 3.1.

The main purpose of these tools is to provide meaningful feedback regarding the state of a system at any given time. Although the scope varies, they frequently cover both the underlying infrastructure as well as the services themselves, regardless of deployment (i.e. containerized or legacy), to observe and verify the quality of the service provided over time.

Besides displaying textual data, most support the visualization of graph-based representations for various performance metrics for both the hardware of the infrastructure and the services themselves such as memory usage and CPU load. In addition, some feature the visualization of the network topology of the system even through automatic discovery from a single node. Another common functionality is the inclusion of configurable alerts with the purpose of notifying the system's maintainers whenever some potentially dangerous event occurs. Specifically, if the established target conditions were violated, for example, whenever some parameter or set of parameters exceeds the established thresholds. Some even include artificial intelligence (AI), most commonly, machine learning approaches, with the main goal of improving anomaly detection, for instance, by automatic outlier identification and ultimately aid the system admins in their work.

The main interest in the study of such tools lies in, for the context of this dissertation and for those that support it, the network topology visualization. This may serve as a basis for possible

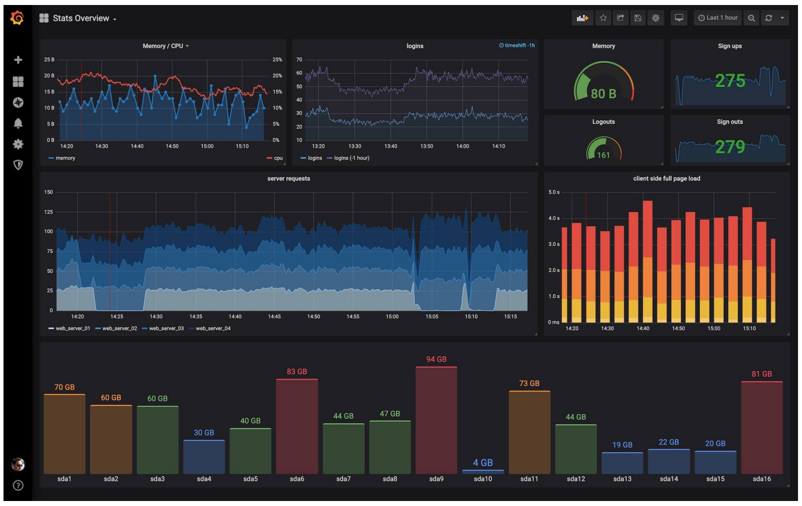


Figure 3.2: Sample of Grafana's dashboard. From https://grafana.com/

visual metaphors and notations applied to the visualization of configurations in the developed solution.

Although numerous tools were surveyed, only the following couple are analyzed in more detail. This is because they comprise a good representation for their purpose, that is, full-stack level monitoring and container only monitoring, and support most features that define this category of tools. Even though some minor differences can be identified between the remainder of the tools, any further analysis would be mostly redundant and therefore unnecessary. This level of detail is sufficient to answer the previously stated objective and a more extensive analysis would be outside of the scope for container monitoring.

Prometheus[8]**.** One of the most proliferated open-source monitoring solution. Many other monitoring solutions depend upon Prometheus for data metric retrieval such as Sysdig and Weavescope. It is built for full-stack monitoring independently of architecture. Data is collected via a pull model and is exposed as a web service which can be consumed, through the query-language *PromQL* by clients including the integrated web UI and Grafana[9] (displayed in Figure 3.2) for data visualization. Additionally, it also includes an *Alertmanager* to handle and manage alerts.

Datadog[10]**.** Among the numerous existing monitoring tools, Datadog represents one of most comprehensive example in this category, spanning a vast array of the previously noted features, making it one of most complete tools available. Besides core functionalities such as text and graph-based data feedback as well as support for alerts, the most notable feature is the ability to visualize the infrastructure including all the nodes and how they are connected in the network along with the corresponding throughput for each connection updated in short intervals. Each node can be

[8]Prometheus, available at https://prometheus.io/
[9]Grafana, available at https://grafana.com/
[10]Datadog, available at https://www.datadoghq.com/

expanded in further detail to the individual containers running in the node (when applicable) and eventually to the processes being executed, whether in a container or directly in the host itself. Therefore, it is usable both with containerized and non-containerized applications.

Weavescope[11]**.** Unlike most tools which target application monitoring regardless of how the service is supplied, that is, whether it is containerized or not, Weavescope is primarily focused on monitoring and troubleshooting container-based applications for Docker and Kubernetes. It allows the visualization of the processes, containers, whether standalone or part of a stack identified by matching colors, and Docker Swarm services present in the host, displayed in graph-based diagrams with each element represented as a node and their dependencies as arrows. For each type, a set of filters is provided as well as additional options such as displaying the Docker networks connected to each container. Every node can be clicked and expanded, displaying a window with detailed information, depending on its type, ranging from general data to performance metrics and graphs. Additionally, it supports some limited container management operations such as restarting and stopping containers and executing commands directly from the web page.

Table 3.1 shows an overview of the monitoring tools surveyed, capturing how the main features are distributed across distinct tools. By partial monitoring, we consider feedback exclusively for containers while full monitoring offers feedback for all levels of an application including network, node, container, and process monitoring.

Name	License	Target Infrastructure	Topology View	Alerts
cAdvisor	Open-source	Containers	No	No
Scout	Proprietary	Full	No	Yes
Data Dog	Proprietary	Full	Yes	Yes
Prometheus	Open-source	Full	No	Yes
Sysdig	Proprietary	Full	Yes	Yes
Sensu monitoring framework	Open-source	Full	No	Yes
Sematext	Proprietary	Full	Yes	Yes
Dynatrace	Proprietary	Full	Yes	Yes
Broadcom AIOps	Proprietary	Full	No	Yes
Site24x7	Proprietary	Full	No	Yes
AppDynamics	Proprietary	Full	Yes	Yes
Weavescope	Open-source	Containers	Yes	No

Table 3.1: Monitoring tools comparative overview. Appendix A contains a full list of sources

Of the tools surveyed, most are proprietary with only four being open-source. Approximately half support visualization of the topology and almost all include alerts.

Monitoring is a critical task in operations to maintain applications in production. Current solutions are well developed and established, offering a wide variety of options fit for most requirements and use-cases. cAdvisor and Prometheus are the most prominent open-source monitoring

[11]Weavescope, available at `https://github.com/weaveworks/scope`

solutions. Some projects aim to combine the strengths of both by providing convenient technology stacks. Examples include dockprom[12] and swarmprom[13] for local and distributed environments respectively.

3.2.2 Services

In this category, we consider tools built for container management and orchestration tasks. Therefore, these tools operate at CaaS and PaaS levels. In particular, mainly those which support Docker technologies. Due to the subject matter of this work, only tools which provide a true graphical user interface (GUI) or at least some non-textual visualization component were considered.

We can subdivide this category into two groups: **Docker GUIs** and **Orchestration Frameworks**.

3.2.2.1 Docker GUI

This group of tools provides a GUI for managing Docker resources, whether in local or remote environments and usually for development purposes. The majority feature a visually interactive method to manage Docker resources, namely, containers, images, volumes, and networks. In fact, these work primarily as a straightforward wrapper for the CLI commands, from its most basic level, that is, common functions such as container creation or deletion, to stack level orchestration and management for multiple service-based applications via Docker Compose. Additionally, some examples built purely for visualization were considered as well.

Kitematic[14]. Aims to provide a straight-forward form-based GUI for Docker container focused management. Besides container management actions such as creating, restarting, and deleting a container, it supports the management of images, volumes, and networks. Furthermore, it includes automatic port-mapping and integration with Docker Hub for direct image searching. Overall, Kitematic presents a set of basic features mostly fit for users who do not require extensive configuration options and is not optimized for configuration and management of service stacks. It currently stands as one of the most common visual tools for Docker being part of the official Docker Toolbox for Windows and Mac and is, as a result, frequently one of the first visual tools Docker users experience.

Portainer[15]. A web-based GUI for the management of stacks, containers, images, networks and volumes through a mostly form-based interface for both local and remote environments including swarm clusters [46]. In addition, it also features access level management and templates built on top of Docker Compose for service stack composition written in JSON. Regarding stack management, it supports a built-in textual editor for the definition of Docker Compose configurations

[12]dockprom, available at `https://github.com/stefanprodan/dockprom`
[13]swarmprom, available at `https://github.com/stefanprodan/swarmprom`
[14]Kitematic, available at `https://kitematic.com/`
[15]Portainer, available at `https://www.portainer.io/`

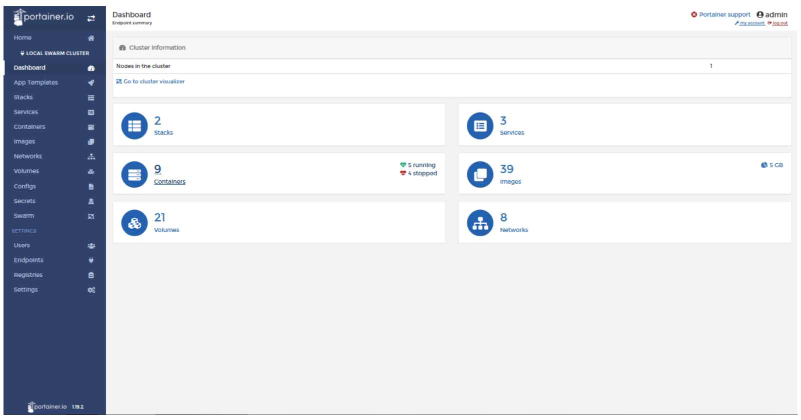


Figure 3.3: Sample of Portainer's main dashboard. From `https://www.portainer.io/`

as well as the option of importing existing files stored either locally or in remote git repositories. For each container, it is possible to inspect it, display its logs, visualize some statistics such as CPU and memory usage and execute commands directly through the web client.

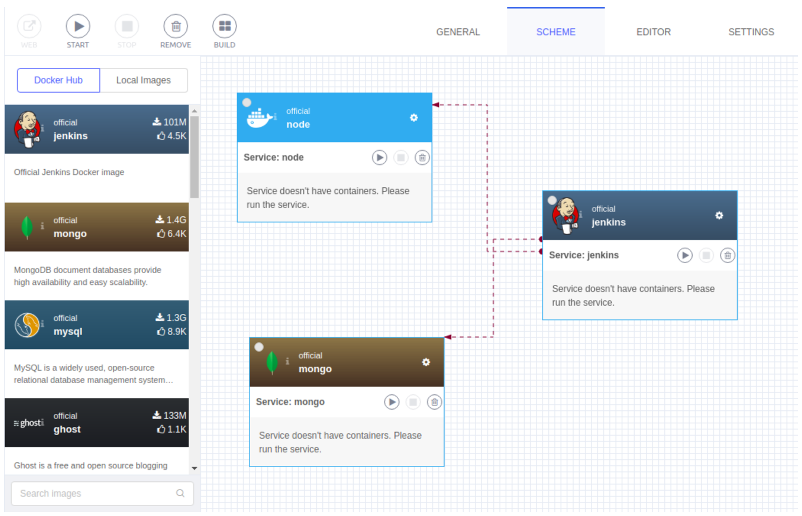
Figure 3.3 displays the main dashboard of Portainer's web UI in swarm mode. This dashboard functions as the hub to navigate to the corresponding page to inspect and interact with each Docker artifact (e.g. containers).

Portainer currently stands as one of the most popular visual Docker management tool based on its star rating on Github, providing a comprehensive set of features which wrap most Docker CLI actions in a web environment. However, it is fully reliant on purely form-based stack definitions.

Dockstation[16]. Provides a GUI for the handling of Docker containers both in local and remote environments aimed primarily for development purposes. It supports basic container management such as creation and deletion of containers as well as some container monitoring utilities including performance graphs, similarly to other tools such as the previously mentioned Portainer.

The most unique (and relevant) feature is related to what the tool defines as a "Project", essentially a Docker Compose service stack specified in a `docker-compose.yml` file. After the creation of a new project, during which an existing `docker-compose.yml` file can optionally be imported, the developer has access to four main tabs: "General", "Scheme, "Editor" and "Settings". In the "Scheme" tab, the developer can not only visualize the overall services scheme featuring a visual representation of each service as well as their relationships and dependencies in a graph-like diagram but also edit each service via forms for some of its properties, specifically, environment variables, volumes, and ports. Whenever the user edits some property, the changes are then reflected on the associated `docker-compose.yml` accessible in the "Editor" tab. In addition, the opposite is also possible, that is, the direct edition of the `docker-compose.yml` file with the changes reflected on the corresponding scheme.

[16]Dockstation, available at `https://dockstation.io/`



Official Jenkins Docker image

MongoDB document databases provide high availability and easy scalability.

MySQL is a widely used, open-source relational database management system....

Ghost is a free and open source blogging

Service doesn't have containers. Please run the service.

Service doesn't have containers. Please run the service.

Service doesn't have containers. Please run the service.

Figure 3.4: Sample from Admiral's visual orchestrator.

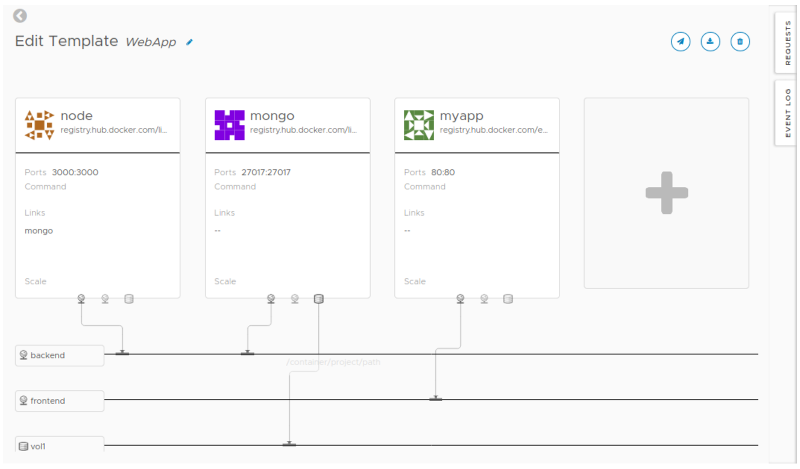
Figure 3.4 showcases the "Scheme" perspective in the "Project" tab for a simple stack and some additional UI elements. To the left of the scheme lies the image palette selector, allowing the user to search and select either local or remote images stored in the DockerHub registry. The top action bar includes quick actions which trigger Docker Compose commands such as *docker-compose up* for the start button. In the scheme itself, the boxes represent the services and the dotted arrows represent the *depends_on* relation between the services. To add a service to the scheme, the user can drag the intended image from the palette to the scheme area. In the current version, it is not possible to visually add dependencies between services, requiring the user to instead use the editor and add the dependency textually in the `docker-compose.yml`.

Currently, the visual orchestrator provided by this tool stands as the closest solution to what is being proposed in this work, featuring the most complete visual approach available. However, we can identify the following limitations regarding its approach:

- Lack of visual representation of volumes in the "Scheme" view. Volumes are strictly specified via the form for each service and have no visual notation.

- Lack of visual representation of networks in the "Scheme" view. Although custom networks can be specified in the "Editor" view, these do not feature any visual notation in the "Scheme" view, both for the volume itself and in the service edition form. Therefore, it is not possible to configure networks via visual interaction in which case the default network is used instead.

- Low directness and liveness level. It requires the shift between the "Editor" and "Scheme" views for either to update whenever the other is changed. Additionally, the user has to explicitly click on a save button in the editor for the changes to persist whenever they switch to a different tab, with no shortcut available.


Edit Template *WebApp*

node
registry.hub.docker.com/li...
Ports 3000:3000
Command
Links
mongo
Scale

mongo
registry.hub.docker.com/li...
Ports 27017:27017
Command
Links
--
Scale

myapp
registry.hub.docker.com/e...
Ports 80:80
Command
Links
--
Scale

Figure 3.5: Sample from Admiral's template visual orchestrator.

Admiral[17] Functions as a web-based GUI for container management and provisioning over a cluster of infrastructure. Unlike Dockstation, this tool is mainly deployment and production-oriented. Besides the provisioning of single containers, it supports the composition of multi-container stacks by the definition of "Templates". These include four main components: containers, volumes, networks, and closures. Each can be created and configured via a form-based interface for the specification of the corresponding details. It is then possible to visually connect each container or closure with a network or volume by dragging the mouse from the source component to the target element. Finally, each template can be either directly provisioned to a configured cluster or exported in one of two formats: YAML Blueprints and Docker Compose (version 2 exclusively). The opposite is also possible, that is, importing a template from a file in either of the two aforementioned formats, upon which, the stack can be visualized and edited.

Figure 3.5 displays a simple template containing 3 services, 2 networks, and 1 volume. The user can add a new component (container, network, volume, or closure) by hovering over the empty box with the plus icon and clicking on the desired element. Upon which they are redirected to the corresponding form to edit is properties. Each of the services contains a set of network and volume anchor points, located at the bottom, for the total number of networks and volumes declared in the configuration (3 in this instance). These allow the user to attach the services to their corresponding volumes and networks. Dependencies between services, known as links, are displayed and directly editable as a property located in the service itself. To edit more advanced properties, the user must expand the service and access its edition form.

The main drawbacks of the provided visual editor are its limited graphical interaction, for instance, the inability to rearrange any of visual elements (with the exception of network connections) and lack of visual representation for the dependencies between containers (either depends_on or links) which are instead text-based, specified via a drop-down located in the service representation.

[17] Admiral, available at `https://github.com/vmware/admiral`.

3.2.2.2 Orchestration frameworks

The next set of technologies have the common goal of functioning as high-level orchestration frameworks for automating deployment and management of distributed and microservices systems deployed in a cluster of nodes of infrastructure, whether in the cloud or bare-metal datacenters. Orchestration frameworks usually provide essential features such as load balancing, fault tolerance, and autoscaling [19, 41]. Although these frameworks are frequently container agnostic, they often utilize Docker containers. Docker Inc. itself offers its alternative, Docker Swarm [63], which is strictly controlled via a CLI. However, there are some third-party GUIs available which primarily focus on providing a visual alternative, wrapping the commands offered, and providing some additional features such as monitoring capabilities. Examples include Swirl and Swarmpit.

Swirl[18]. Web-based GUI for management of Docker, focused on swarm clusters. Provides limited management options for local Docker resources (i.e. Images, Containers, and Volumes). Namely, the ability to list all and search for some resource for each artifact type as well as inspect and delete individual resources. Comparatively, Swarm management is at the core of the tool. It supports the management of the nodes in clusters, networks, individual services, and complete stacks. Stack definition is done textually either through the built-in editor or by importing an existing `docker-compose.yml`. Other registries (besides Docker Hub[19]) can be configured through the GUI and specified during the service or stack creation. Additionally, user permission options are provided for restricting access to resources through an extensive configuration scheme based on user roles, essentially user groups with a specific set of permissions. When creating a user role, it is possible to pinpoint, for each artifact, what individual actions can be performed (e.g. deleting and updating) and what properties can be visualized. Roles are then assigned to users and users can have multiple roles. It also features some monitoring utilities by integrating with Prometheus.

Swarmpit[20]. A web-based GUI for the management of a Docker Swarm cluster mainly focused on deployment. It supports visualization and management at the application level, namely, services, volumes, and secrets and at the infrastructure level, in particular, nodes and networks. It allows deployment of full Docker Compose based stacks specified in either a `docker-compose.yml` file or through the built-in editor as well as individual services integrating multiple registries including Docker Hub for image searching. Furthermore, it includes some standard monitoring utilities for the nodes and their containers.

Kubernetes[21]. According to the official website, Kubernetes is a "portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation". In other words, it is a deployment and production focused

[18]Swirl, available at https://github.com/cuigh/swirl
[19]More information available at https://hub.docker.com/
[20]Swarmpit, available at https://github.com/swarmpit/swarmpit
[21]Kubernetes, available at https://kubernetes.io/

management platform for applications built on container-based technology stacks, primarily Docker [38, 17] developed by Google [63]. It defines its own vocabulary and set of high-level concepts. Examples include Pods which refer to a group of containers that share an IP, namespace, and storage volume and Services. Additionally, a web-based UI is offered for both resource management and monitoring.

Apache Mesos[22]**.** Platform providing a common interface for integration with cluster computing frameworks [32]. Among the many frameworks available, Aurora[23] and Marathon[24], are some of the most popular for container orchestration. These frameworks implement scheduling and job management logic among other essential features such as rollbacks. Both frameworks feature a GUI for inspecting jobs.

Hashicorp Nomad[25]**.** Minimalist general-purpose workload orchestrator for containerized and legacy applications. Is architecturally more simple when compared to the previous approaches. Works as a single binary for both the clients and servers and requires no external services for coordination or storage. "Supports multi-datacenter and multi-region configurations for failure isolation and scalability.

Rancher[26]**.** This tool primarily offers a solution for managing multiple Kubernetes clusters independently of the underlying infrastructure, taking advantage of the benefits of Kubernetes and promising an easier learning curve and improvements in three key areas: *cluster operations and management*, *intuitive workload management* and *enterprise support*. It effectively adds an abstraction layer on top of Kubernetes, enabling system admins to manage all clusters from a single location. Additionally, it provides its CI/CD pipeline, monitoring, and logging capabilities and a GUI Although previous versions supported multiple orchestration frameworks such as Mesos and Docker Swarm and even a custom orchestrator denominated "Cattle" based on Docker Compose stacks [46], this feature was ultimately removed in favor of an exclusive focus on Kubernetes technologies.

Many more orchestration frameworks are available in the market, both proprietary and open-source.

Table 3.2 displays an overview of the surveyed tools concerned with services. Some tools are included in addition to the ones previously analyzed in further detail in this section. As established in Section 2.4, the category refers to the classification scheme proposed by Boshernitsan et al. [12]. Regarding the liveness level, the scale established by Tanimoto was adopted [62, 2]. We can conclude that the majority of tools are form-based. The only exceptions, Dockstation and Admiral, which are also classified as hybrid. The liveness level is consistent across the solutions.

[22]Apache Mesos, available at `http://mesos.apache.org/`
[23]Aurora, available at `http://aurora.apache.org/`
[24]Marathon, available at `https://mesosphere.github.io/marathon/`
[25]Hashicorp Nomad, available at `https://www.nomadproject.io/`
[26]Rancher, available at `https://rancher.com/`

Name	Environment	Category	Monitoring Features	Visual Orchestration	Liveness Level
Kitematic	Desktop	Form-based	Text	No	2
Dockeron	Desktop	Form-based	None	No	2
Seagull	Web	Form-based	None	No	2
Portainer	Web	Form-based	Charts	No	2
Docker Compose UI	Web	Form-based	Text	No	2
Swirl	Web	Form-based	Charts	No	2
Swarmpit	Web	Form-based	Charts	No	2
DockStation	Desktop	Form-based, Hybrid	Charts	Yes	2
Admiral	Web	Form-based, Hybrid	Charts	Yes	2
Rancher	Web	Form-based	Charts	No	2
Kubernetes	Web	Form-based	Charts	No	2
Mesos	Web	Form-based	Charts	No	2
Nomad	Web	Form-based	Charts	No	2

Table 3.2: Service level visual tools comparative overview. Appendix A contains a full list of sources

3.2.3 Infrastructure

The focus of these tools lies in the orchestration and management of infrastructure resources. These resources correspond to the base components of IT services, namely, physical (i.e. hardware and facilities), software, and networks. They may either be maintained internally and deployed in owned data-centers, externally within cloud computing environments or a combination of both, known as a hybrid infrastructure.

Some well-established examples include Puppet and Chef originally for the management of traditional data-centers and afterward supporting cloud and hybrid infrastructure. More recently, the proliferation of cloud technologies promoted a surge of numerous tools built for managing those resources. Some of the most notable, such as AWS CloudFormation[27] and Terraform[28], are mostly text-based at their core and follow the Infrastructure as Code (IaC) practice which in turn supports Infrastructure as a Service (IaaS). Although these tools operate at a level below containers, there are some visual approaches which are, therefore, worth researching. In particular, those that are hybrid, which very closely match the challenges inherent to a visual approach for Docker Compose orchestration.

CodeHerent[29]. Visual development environment, leveraging a hybrid visual programming language for the edition and visualization of Terraform configuration files. Although, initially, this tool adopted a box-based representation in which the artifacts are hierarchically organized in

[27]More information available at `https://aws.amazon.com/cloudformation/`
[28]More information available at `https://www.terraform.io/`
[29]CodeHerent, available at `https://codeherent.tech/home`

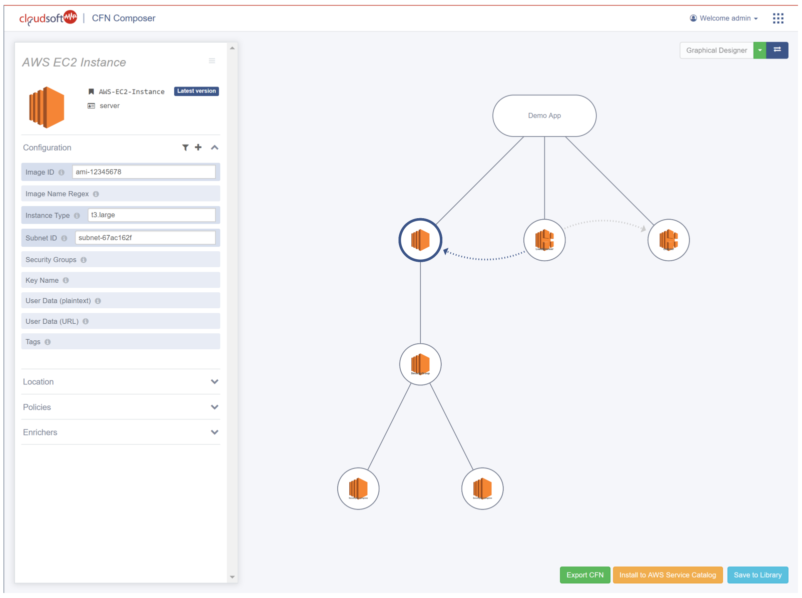


Image ID ami-12345678
Image Name Regex
Instance Type t3.large
Subnet ID subnet-67ac162f
Security Groups
Key Name
User Data (plaintext)
User Data (URL)
Tags

Figure 3.6: Sample of CloudSoft Visual Composer's interface.

boxes, more recent iterations opt for a graph-based diagram for the representation of the distinct artifacts and their relationships. It includes direct integration with git, supporting basic version control. In essence, it tries to solve a very similar problem with a similar visual approach to the one proposed in the solution.

CloudSoft Visual Composer[30]**.** Similarly to CodeHerent, this tool aims to provide a hybrid visual approach for the configuration and provisioning of infrastructure. Unlike CodeHerent, however, it is exclusively focused on AWS technologies, in particular EC2 CloudFormation templates. It follows a tree-like diagram for the representation of the distinct artifacts which make up the infrastructure, with nodes corresponding to resources starting from a single root node representing the application, upon which all others descend, and multiple types of connections such as arrows for dependencies and references between resources. The user can add a node as a descendent of other, be it the root or not, by either clicking on an icon underneath the node or by dragging a resource from the palette to the parent node and edit its properties in a form-based interface. Furthermore, it includes snippets of documentation directly accessible on each element by hovering above help icons. Similarly to other hybrid visual approaches already discussed, it supports view switching between the visual composer and a built-in textual editor of the corresponding YAML file.

Cloudcraft[31]**.** Drawing tool for the creation of AWS cloud infrastructure diagrams mostly for documentation purposes. It is possible to directly access any resource on the AWS Web Console by clicking on the corresponding graphical element. Additionally, it includes a monitoring

[30]CloudSoft Visual Composer, available at `https://cloudsoft.io/software/cfn-composer/`
[31]Cloudcraft, Available at `https://cloudcraft.co/`

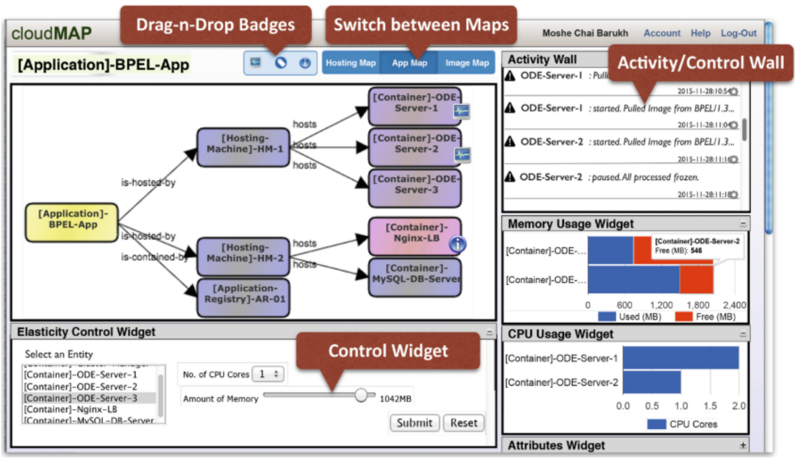


Figure 3.7: Sample application map from CloudMap's interface. From [64]

component, allowing the connection between the diagram to the corresponding infrastructure once provisioned, supporting the visualization of up-to-date data regarding the state of infrastructure.

CloudMap. According to authors, Weerasiri et al. [64], this tool "offers a refreshing “visual” attempt at simplifying the way DevOps can navigate and understand cloud resource configurations, as well as monitor and control such resources" [64]. In essence, it aims to provide a centralized system for cloud management and monitoring by integrating crucial features usually found across multiple independent technologies. In particular, the resulting visual notation is split between three maps, each corresponding to some organizational pattern:

- **Image Map.** "Visualizes the recursive dependency of Images within a Registry."
- **Application Map.** "Visualizes the organization and inter- action of Hosting Machines and/or Containers of an Application." Figure 3.7 showcases a sample of the interface including an application map.
- **Hosting-Machine Map.** "Visualizes the organization of Containers and Applications within a specific Hosting Machine."

The resulting visual notation is based on a graph-like diagram for representing resources and their connections. The reasoning behind this choice and a more detailed description of what each visual element represents is supplied by the authors in the source work [64].

Due to its nature, this tool fits in all of the 3 established categories, however, because its main focus is on infrastructure management, it is featured in this category.

Table 3.3 displays an overview of the surveyed tools concerned with infrastructure. As previously established, the motivation behind the review of tools in this category was exploring highly visual approaches concerned with infrastructure management. Two tools, CodeHerent and Cloudsoft Visual Composer, offer a true visual orchestrator for provisioning cloud resources. These tools closely match what is proposed in this work since they provide a visual orchestrator built on top of textual-based technologies albeit applied to distinct yet closely related concepts. Even though many other tools exist in this field, these fall outside of the defined scope for review.

Name	Environment	Target Infrastructure	Purpose
CodeHerent	Web	Cloud	Provisioning
Visual Composer	Web	Cloud	Provisioning
Cloudcraft	Web	Cloud	Documentation
CloudMap	N/a	Hybrid	Monitoring & Management

Table 3.3: Infrastructure level visual tools comparative overview. Appendix A contains a full list of sources

3.2.4 Discussion

Based on the previous analysis, we can currently find numerous instances of visual approaches to aid the developer in all aspects of software management and operations from infrastructure provisioning to service management and monitoring. Some even consider the whole spectrum of activities, serving as a single hub for all production-related tasks. A common pattern among all is the inclusion of some sort of monitoring component.

Regarding the tools which are concerned with *service* management and provisioning, in particular, with containerized workloads, we find instances targeted towards deployment and production environments as well as development environments. However, most tools heavily rely on form-based or purely textual methods to define and orchestrate technology stacks, as seen in Table 3.2, thus they do not feature a truly visual method for this purpose. Those that do (Dockstation and Admiral) are still either underdeveloped or incomplete in working as visual orchestrators, often overly relying on forms and lacking visual notations for certain artifacts and therefore do not offer a complete visual approach. Moreover, the idea of a visual orchestrator for resource configuration and provisioning tasks has also been explored for *infrastructure* provisioning and configuration in cloud environments as seen in Table 3.3.

Additionally, in both domains, *services* and *infrastructure*, there is a severe lack of exploration of distinct visual notations, with most adopting some variation of graph-based diagrams. Notwithstanding the adequacy of such notations, we believe there may be potential in exploring others or even visual metaphors which may be easier to understand, especially for inexperienced developers that may have difficulty in understanding certain concepts.

Chapter 4

Preliminary Work

The purpose of this chapter is to describe and present the results of the preliminary research work performed to identify the issues developers face when working with Docker and Docker Compose technologies. For this purpose, a survey was devised and distributed among students.

First, we present the motivation (Section 4.1), then the specific goal (Section 4.2), research questions of this survey (Section 4.3) and the chosen methodology (Section 4.4). We then describe the data collection process (Section 4.5), present the data analysis method (Section 4.6), the results of the data analysis (Section 4.7) and lastly draw the final conclusions (Section 4.8).

4.1 Motivation

The motivation behind this research was strongly related to the rest of the work developed in this dissertation. On one hand to verify and strengthen the overall motivation by testing whether developers truly feel difficulty when working with Docker Compose technologies, justifying the need for an alternative solution to work with these technologies. On the other hand, to extract useful knowledge for design considerations during future development steps.

In more detail, the motivation was threefold:

- Understanding if developers feel difficulty when working with Docker Compose in practice, therefore substantiating the need for an alternative;
- If so, to understand what the biggest challenges are so that the solution is carefully tailored to target, among others, these issues;
- Identifying if developers use existing ancillary or alternative tools and solutions for Docker Compose development and, if so, what tools are used.

Although the motivation is highly tied to the overall goal of this work, we believe the results may be of interest to other researchers looking for data related to the difficulties of working with Docker Compose. For instance, the results might be useful in comparing how Docker Compose performs in relation to other orchestration frameworks (e.g. Kubernetes).

4.2 Specific Goal

The goal was, on one hand, to identify current issues and challenges that developers face when using Docker Compose by measuring the difficulty felt when working with this technology and what strategies and possible solutions are adopted to overcome those difficulties. And on the other, to identify what ancillary or alternative tools developers use when working with Docker Compose in practical scenarios.

4.3 Research Questions

The established goal can be unfolded into the following research question:

RQ1 *"What tasks take significantly longer than expected when working with Docker Compose?"*

RQ2 *"What strategies and approaches are adopted to solve problems?"*

RQ3 *"What ancillary tools and software are used by developers when working with Docker Compose in practice and what is their role?"*

These research questions are directly in line with the defined goal and are roughly equivalent to the two sub-objectives identified.

4.4 Methodology

The chosen approach was to develop an online survey. The survey was created in the Google Forms service[1] and took an estimated duration of approximately 5 minutes to fill in. The survey is available in Appendix B in full.

[1]More information available at `https://www.google.com/forms/about/`

The survey was structured in two question groups: firstly the *Personal Context* group, with the intent of gauging the current degree of experience with Docker Compose of each participant and secondly the *Working with Docker Technologies* group, which focused on identifying and measuring the difficulties themselves.

When possible, questions were formulated according to a five-point Likert scale [36], ranging from *strongly disagree* to *strongly agree*. For the questions where this was not possible, free text and multiple-choice answers were considered as well.

4.5 Data Collection

The survey was carried out by a total of 68 participants. The participants were 4th year students enrolled in the Integrated Master in Informatics and Computing Engineering (MIEIC) degree program at the Faculdade de Engenharia do Porto (FEUP). The students had at least some experience with Docker Compose since they were finishing a subject in which they had to work with Docker technologies. Taking this into account, we believe the sample is representative of novice Docker users.

The data was collected in two steps: 1) initially through a procedure performed in person during classes and 2) later distributed online to the same target audience. This approach had the goal of reaching a wide number of participants but still keep within the defined time limit.

Regarding the first step, this was performed physically in 6 classes with an average of 24 students per class. The procedure started by writing the URL to access the survey on the classroom's whiteboard followed by a brief introduction of the survey, explaining the context, goal, and motivation along with the filling procedure. This step took roughly 1 hour in total between all classes.

The second step of data collection was done online to offer a chance for those that couldn't originally fill the survey (for some reason) to do so. An email, containing the URL to access the survey, was distributed to the target students.

4.6 Data Analysis Methods

The first step in the analysis process was performing a global review of the collected data. This process was facilitated since Google Forms automatically generates some graphs and statistics which provided some insight into the results.

This preliminary review demonstrated that some participants did not have any prior experience manipulating a `docker-compose.yml` file themselves. With the intent of producing more meaningful results, we felt that it was not useful to consider these answers. As a result, these entries were excluded for the subsequent analysis of the *Working with Docker Technologies* question group, which considered a total of 48 answers (instead of the initial 68).

Afterward, a set of additional statistics were calculated, in particular, the mean ($\bar{x}$), median ($\tilde{x}$), and interquartile range (IQR) range for the Likert scale questions and standard deviation (σ)

for the numeric scale questions. These statistics provide more insight into the distribution of the answers and the most likely value.

The results of this analysis allowed us to extract the necessary conclusions for each of the research questions. The answer to the first research question (**RQ1**) was evaluated through the computed mean and additional statistics obtained from the Likert-scale questions which individually targeted common tasks and activities when developing a `docker-compose.yml`. The second research question (**RQ2**) was evaluated directly through the analysis of the answers to the open-ended question of what strategies were used to debug a malfunctioning Docker Compose configuration. The third research question (**RQ3**) was evaluated directly through the analysis of the answers to the open-ended question addressing what tools the participants had used and how they helped them.

4.7 Data Analysis

In this section, we present the analysis of the collected data and demonstrate the achieved results.

4.7.1 Personal Context

We began by analyzing the results of the *Personal Context* questions group. These results provide insight into the background of the participants.

	$\bar{x}$	$\tilde{x}$	IQR
PC1	3.38	3.5	1

Table 4.1: Summary of descriptive statistics of the answers to the Likert scale questions in the *Personal Context* question group.

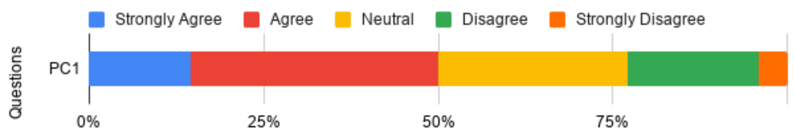


Figure 4.1: Distribution of answers to the Likert scale questions (PC1) in the *Personal Context* question group.

	$\bar{x}$	σ
PC2	2.67	1.894
PC3	1.92	1.485
PC4	2.06	1.767

Table 4.2: Summary of descriptive statistics of the answers to the numeric scale questions in the *Personal Context* question group.

PC1 At this point in time I am experienced in writing a docker-compose.yml file for a software system;

PC2 Number of projects which included a docker-compose.yml;

PC3 Number of projects with docker-compose.yml files created by others (colleagues or third parties);

PC4 Number of projects with docker-compose.yml files created/updated by the participant.

List 4.1: Question identifiers for Table 4.1, Figure 4.1, Table 4.2 and Figure 4.2.

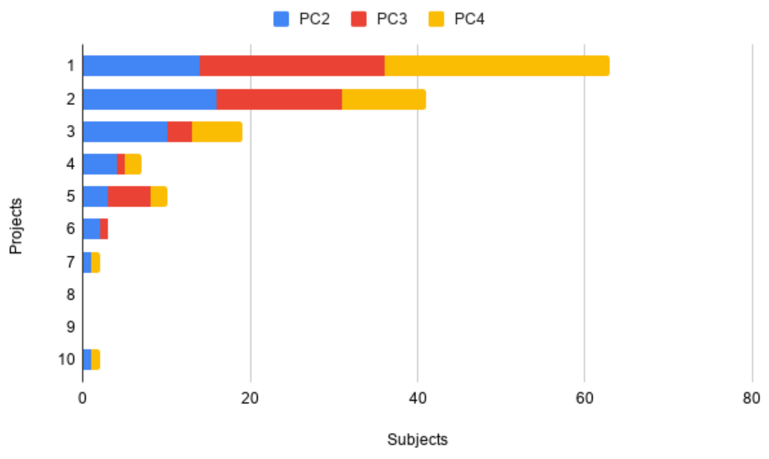


Figure 4.2: Distribution of answers to the numeric scale questions in the *Personal Context* question group.

Figure 4.1 displays the distribution of answers to the Likert scale background questions. Considering this data and the corresponding statistics in Table 4.1, the students' perception of their experience suggests an intermediate level of skill. However, the quantitative results of the answers related to the number of projects in which Docker Compose was used (displayed in Table 4.2 and Figure 4.2) reveal a contradiction. Particularly, when considering the low mean of projects in which a participant edited a `docker-compose.yml` (BG4), we can confirm that the participants generally had low experience and could be considered as novices. This outcome was expected as, based on their academic path until that moment, the students only had limited exposure to this technology. Thus, we believe that the students overestimated their skills.

4.7.2 Working with Docker Technologies

The questions in this group addressed two distinct activities: writing and reading a `docker-compose.yml` file. This approach provided more granular insight into how the perception for both might differ.

	$\bar{x}$	$\tilde{x}$	IQR
W1	3.29	3	1
W2	3.17	3	1
W3	3.60	4	1
W4	3.44	3	1
W5	3.27	3	1
W6	3.31	3	1
W7	3.31	3	1
W8	3.23	3	1

Table 4.3: Summary of descriptive statistics of the answers to the Likert scale questions for writing related tasks in the *Working with Docker Technologies* question group.

W1 Finding out what are the keys that I need;
W2 Finding out what images are available;
W3 Trying to understand why the services are not working as intended;
W4 (Re)starting the services to confirm that they are working as intended;
W5 Configuring the properties of each service (e.g. port mapping, name, ...);
W6 Configuring the dependencies between the services (e.g. depends_on);
W7 Configuring volumes and how they are attached to the services;
W8 Configuring networks and how they are connected to the services.

List 4.2: Question identifiers for Table 4.3 and Figure 4.3. All statements are preceded by "I spend a lot of time...".

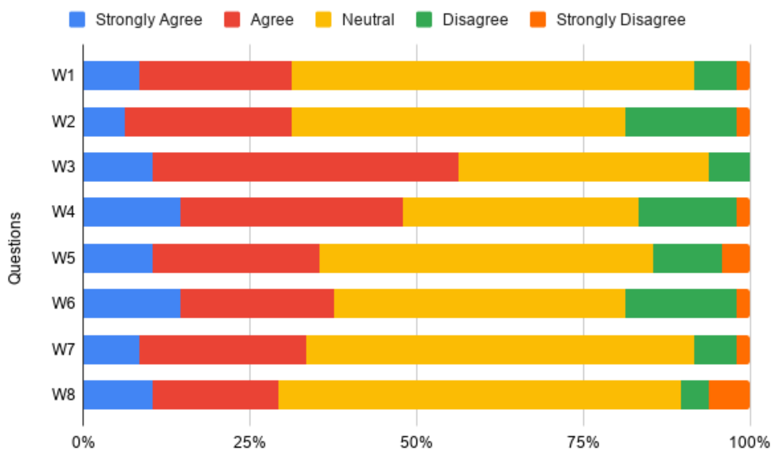


Figure 4.3: Distribution of answers to the Likert scale questions for writing related tasks in the *Working with Docker Technologies* question group

	$\bar{x}$	$\tilde{x}$	IQR
R1	3.02	3	1.25
R2	3.19	3	1
R3	3.17	3	1
R4	3.31	3	1

Table 4.4: Summary of descriptive statistics of the answers to the Likert scale questions for reading related tasks in the *Working with Docker Technologies* question group.

R1 Trying to understand what the services are;
R2 Trying to understand the dependencies between services (e.g. depends_on);
R3 Trying to understand what volumes are used and how they are attached to the services;
R4 Trying to understand what networks are used and how they are connected to the services.

List 4.3: Question identifiers for Table 4.4 and Figure 4.4. All statements are preceded by "I spend a lot of time..."

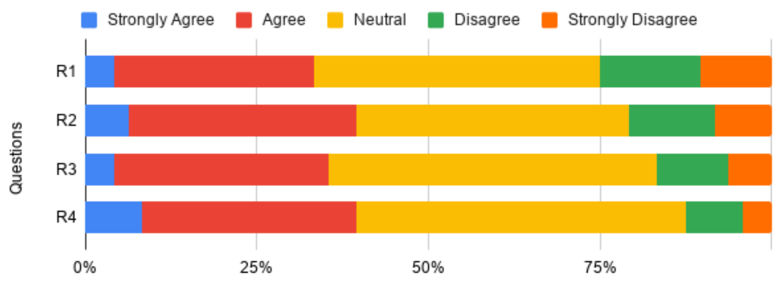


Figure 4.4: Distribution of answers to the Likert scale questions for reading related tasks in the *Working with Docker Technologies* question group.

Figure 4.3 and Figure 4.4 display the distribution of the answers for both activities (writing and reading). We can see that generally the answers focus around the neutral sentiment (3) and appear slightly skewered towards agree sentiment (4). To better understand these results we looked at other statistics.

Table 4.3 and Table 4.4 display the computed statistics for each question. We can conclude that the answers are mostly consensual since the interquartile range does not differ significantly. The medians also reveal that the general sentiment is neutral towards almost all of the tasks. The only exception is the results to question related to debugging activities (W3) which point to an agreement in the difficulty felt.

Taking into account these results, we can answer the question **RQ1**: the general sentiment is neutral. A possible conclusion from this outcome is that the time spent on these tasks matches the expectation of the participants of how much time they should spend to perform them. An alternative interpretation may be that the participants do not have stronger opinions because they feel that working with this technology comprises a less meaningful part of the development process. Yet another possibility is that the participants do not have enough experience to have a clearer opinion as they have yet to fully explore the technology.

In any case, we did register a slight inclination towards an agreement sentiment. However, this was **not** statistically significant.

Regarding the strategies adopted for debugging malfunctioning configurations, the most common was trial and error followed by searching online for a possible solution. Taking these results into account, we can answer **RQ2**. The results suggest that the participants resort mostly to crude and unrefined strategies which are usually inefficient. This behavior may be an indication that current debugging options offered by Docker Compose could be too limited.

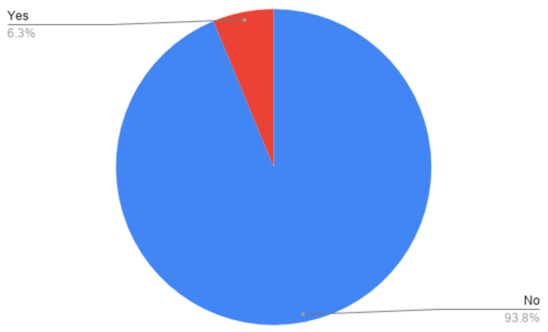


Figure 4.5: Distribution of answers to the question "Do you use any plugins/tools when developing docker-compose.yml files?".

As displayed in Figure 4.5, the vast majority (93.8%) do not use any ancillary tool or software when developing `docker-compose.yml` files and those that do (6.3%), reported only using Visual Studio Code's plugin for Docker[2]. This data provides the answer to **RQ3**.

4.8 Conclusions

The obtained results suggest that the participants do not particularly feel strongly about the existence or nonexistence about any of the potential issues the survey attempted to identify. However, we still identified a slight inclination towards a sentiment of finding tasks time-consuming in general. In particular, the most notable result is related to debugging activities. Based on these results, it becomes difficult to more clearly pinpoint specific tasks which are the cause of bigger struggles (or otherwise). Thus, we consider that the answer for question **RQ1** to be inconclusive, at least for the set of questions included in the survey.

In regards to the remainder of the research questions (**RQ2** and **RQ3**), in summary, the following conclusions can be drawn:

[2]Visual Studio Code Docker Plugin, available at `https://code.visualstudio.com/docs/containers/overview`

- There is a lack of usage of ancillary tools for supporting Docker Compose development.
- The most common strategy do debug a Docker Compose orchestration is by "trial and error".

Chapter 5

Problem Statement

In this chapter, we describe and discuss the problem in a more focused view, starting with the identification of the main issues and limitations of current solutions (Section 5.1) followed by the research statement (Section 5.2), intended target audience (Section 5.3), the solution perspective (Section 5.4) and finally a description of the adopted research methodology (Section 5.5).

As previously established, the main goal of this dissertation is the conceptualization of a more complete visual approach for definition and orchestration of service stacks then what is currently available, including the development of a prototype for leveraging the approach.

5.1 Current Issues

Based on the state of the art review performed in Chapter 3, we can conclude that current visual solutions which support orchestration of Docker Compose stacks, more specifically those which feature a visual orchestrator (i.e. Dockstation and Admiral), have yet to fully explore the potential of such an approach due to the following reasons:

- **Incomplete visual notations.** The most pressing issue found was the lack of visual representations for some of the artifacts and dependencies supported in the Docker Compose standard. Although, when looking in a broad perspective, most aspects are addressed between the various tools (considered individually, Admiral appears to be the most complete), there is no single one which features all simultaneously. Therefore, the abstraction provided by such notations is *leaky* [60], in other words, it does not fully hide the underlying complexity, reducing its effectiveness.

- **Limited visual editions.** None of the solutions allows the edition of all properties supported in Docker Compose through the visual interface. This applies to Admiral, which is not Docker specific and adopts a more generic nomenclature as well as Dockstation which offers very restrictive options. While Dockstation circumvents this limitation through the inclusion of the parallel textual editor, we argue that the resulting workflow does not provide a streamlined experience since a user must switch between the textual and visual perspectives instead of being able to edit stack purely by manipulating the VPL.

- **Sub-optimal directness.** Current solutions present a higher degree of indirectness than desired in regards to the steps required to manipulate a stack visually. For instance, the navigation steps required to create or edit an artifact in Admiral. The user must first click on a button which leads to a completely new page with the corresponding property inputs. After altering the definition, the user must confirm their action for it to take effect. This indirectness is not only inconvenient, hindering the workflow, but also hides invaluable information which could otherwise always be visible.

The results of the research conducted in the previous chapter (Chapter 4) suggest that most users do not use alternatives (visual or else) to work with Docker Compose. We postulate that the lack of tools that support a complete visual approach may have an influence on the lack of adoption of ancillary tools.

The research also revealed that, although no clear conclusions could be drawn in regards to the issues, it is interesting to note the only exception was related to debugging activities for which the participants expressed some difficulty. Although the objective of this dissertation does not directly tackle this issue, we believe that the more complete and explicit notations of a visual approach may aid in alleviating this difficulty by conveying certain concepts more clearly which could help to pinpoint the source of bugs more efficiently.

Taking these issues into account, we believe there is room for improvement by addressing and expanding upon these aspects.

5.2 Research Statement

The author claims the following hypothesis:

H: *A complete visual programming approach for developing orchestration recipes improves the overall developer experience and reduces the error proneness and development time.*

By a *complete visual approach*, we assume an approach that aims to maximize the gain of the visual notation through common visual strategies (Section 2.4) and minimize the "leakage of the abstraction" [60] by leveraging graphical representations for all relevant concepts in the domain of Docker technology. In particular, considering all of the artifacts, namely, containers, images, volumes, networks, configs, and secrets as well as their relationships and dependencies (e.g. *depends_on*).

By *orchestration recipes* we assume, text-based configuration files which describe the orchestration of service stacks. In particular, and for the scope of this dissertation, those specified according to the Docker Compose YAML file format[1].

By *developer experience*, we assume the overall ease-of-use and intuitiveness of the full experience, considering the steps and actions needed to successfully configure some orchestration setup.

By *error proneness* and *development time*, we assume the number of errors and execution attempts and the amount of time required respectively necessary to successfully configure some orchestration setup when compared to existing solutions.

Based on the previous statement, we establish the following research questions:

RQ1 *"Does a complete visual approach for the orchestration of (Docker) service stacks reduce the development time?"* In this question, we aim to understand if a visual approach is truly useful in reducing the time of development of a Docker Compose configuration.

RQ2 *"Does a complete visual approach for the orchestration of (Docker) service stacks reduce the number of errors?"* In this question, we aim to understand if a visual approach is truly useful in reducing error proneness while orchestrating some Docker Compose configuration.

5.3 Target Audience

The proposed solution targets all developers who wish to utilize Docker technologies for their needs regardless of their degree of knowledge and expertise in regards to these technologies since all may benefit from the aforementioned potential advantages. However, we foresee added interest for inexperienced users who can maximize its usefulness by taking advantage of the more intuitive and friendly environment, resulting in a more linear learning curve and minimizing trial and error. Nonetheless, even more experienced users can potentially benefit, particularly when working with more complex configurations.

5.4 Solution Perspective

For the development of the proposed prototype we can identify some foreseeable key conceptual and implementation challenges, in particular:

[1]More information available at `https://docs.docker.com/compose/compose-file/`

- **Conceptualizing the most adequate visual metaphor and notations.** It is essential to extensively research and explore what visual notations to use for representing the distinct Docker Compose elements (i.e. the artifacts, their properties and how they are connected between each other) as well as how they can be graphically manipulated by the user. These decisions play a key role in the readability and usability of the end result.

- **Finding the optimal approach for the transformation from model to code and vice-versa.** The proposed solution requires an appropriate strategy applied to the model-to-text and text-to-model transformations for exporting to and importing from `docker-compose.yml` files respectively. This can be achieved by researching existing transformations approaches found in MDSE.

5.5 Methodology

For the remainder of this dissertation, we will adopt the engineering research methodology. According to Zelkowitz and Wallace [65], this methodology is defined as follows:

"Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement."

With this definition in mind, we will begin with the development of a prototype leveraging the proposed complete visual approach (Chapter 6). This prototype will then be used as the means to empirically validate the approach. For this purpose, a parallel design user study will be conducted with novice programmers in which relevant performance and perception-based metrics will be evaluated to compare the approach with the conventional text-based solution. These metrics will be subjected to statistical tests and ultimately used to test whether the established hypothesis holds (Chapter 7). The results will be discussed and our findings documented along with the suggestions for improvements and other unexplored ideas as proposed future work (Chapter 8).

Chapter 6

Solution Prototype

In this chapter, we explore and describe the developed prototype solution including its architecture and features as well as discuss the reasoning behind the major decisions we faced throughout the development process.

6.1 Overview

The developed prototype is a visual programming environment featuring the designed complete visual approach for visualizing and orchestrating service stacks with Docker Compose and Docker technologies, named *Docker Composer*. The goal was to design an approach to allow users to define Docker Compose stacks visually through the manipulation of an interactive model of a stack in opposition to defining the stack textually as is conventional.

As initially planned, the resulting VPL is categorized as hybrid since the internal model of the stacks is first translated into an intermediate textual representation, the Docker Compose YAML file, and only afterward instantiated based on this textual definition.

The motivation to implement a prototype was twofold: (1) to serve as a reference implementation to demonstrate the approach in practice and (2) use the prototype as a means to empirically validate the approach (documented in Chapter 7).

6.2 Architecture

The prototype was initially developed as a standard web application to be executed locally in the browser. However, as development progressed, the prototype was eventually migrated to an Electron application. This decision was made for multiple reasons but mainly due to the need to access the file system and local Docker Engine of the host. Electron is a perfect fit for this end since it not only solved these issues but retained the ability to use the rich environment of web technologies and facilitated cross-operational distribution and portability. Although the option sacrifices some performance, we firmly believe that we have more to gain than to lose with this tradeoff.

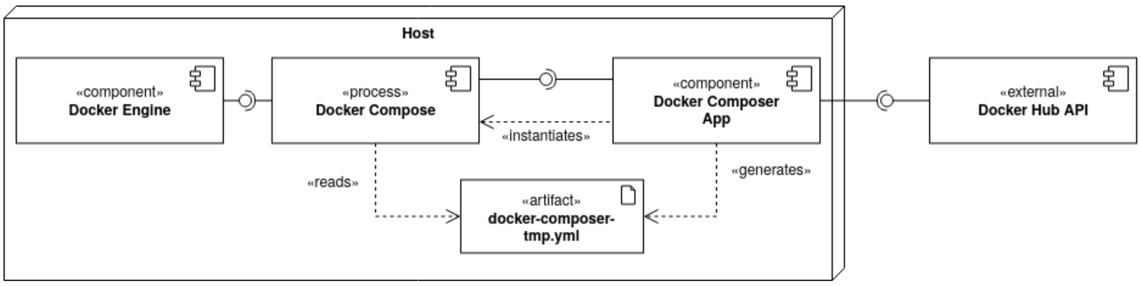


Figure 6.1: Deployment diagram of the prototype.

Figure 6.1 displays a deployment diagram showcasing a high-level view of the prototype and its interactions within and outside the host environment. Within the host environment, the prototype generates Docker Compose YAML artifacts and creates processes running shell instances for the execution of Docker Compose CLI commands which in turn communicate with the Docker Engine. Outside of the host environment, the prototype performs requests to the remote Docker Hub's public API in order to receive information about the images hosted on this service.

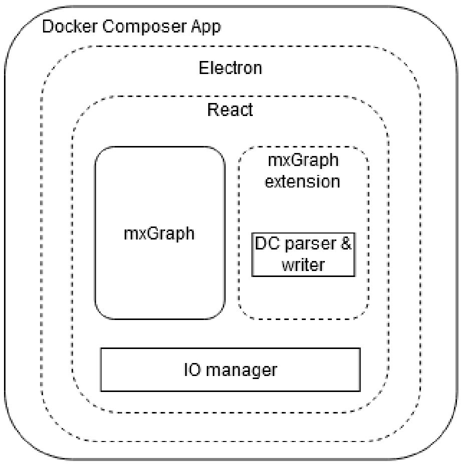


Figure 6.2: High-level architecture of the prototype. Displays the main technologies used and developed modules.

The resulting architecture is fairly straightforward and is displayed in Figure 6.2. The prototype essentially equates to a single-page React[1] web application running in an Electron[2] environment. The application makes use of the diagramming library mxGraph[3] along with the developed extension to fit our requirements as well as an IO module to handle file management and other file system related operations. In more detail, the developed extension configures and expands the library to achieve the desired behavior and includes a module for parsing `docker-compose.yml` files and converting them to the model and writing the model back to a `docker-compose.yml` file.

As a React application, the UI is built with a set of interconnected dynamic components. The root component (App) renders a set of child components, each corresponding to the individual panels of the main UI view, described in Section 6.4.

6.3 Technological Decisions

One of the first and most important decisions was defining how to render and manage the visual components. Two options were considered: (1) using some existing visual programming/drawing framework or (2) developing the engine from the ground up. The first step was exploring what frameworks/libraries were available and whether these satisfied the expected requirements established in the conceptual phase. The technological options in this domain are vast and to better organize the technologies, these were split between three groups based on their purpose and degree of freedom: visual programming frameworks (VPF, such as Rete.js[4] and Scratch), diagram drawing libraries and general drawing libraries. The issue then came down to finding the best balance between the freedom and customization provided by drawing libraries and the established conventions and control of frameworks. After a thorough evaluation, we concluded that the existing visual programming frameworks did not offer sufficient customization to achieve the desired behavior as most were dataflow-oriented. In comparison, diagrammatic drawing libraries offered a much more appealing tradeoff. Of the available technologies, and after careful consideration of the pros and cons, we ultimately chose mxGraph as it appeared to be the most mature and feature-rich of the open-source solutions.

Table 6.1 displays a summary of some of the considered libraries/frameworks and their supported features. Taking account that we prioritized open-source options since the aim was to publish the tool with an open-source license as well, it becomes clear that the most feature-rich is mxGraph as this library supports all of the considered features. Due to this reason as well as its high degree of customization and maturity, this library was chosen.

Furthermore, React was chosen as the library to develop the remainder of the user interface. React is a declarative, component-based Javascript library for building user interfaces, created and maintained by Facebook. While we consider this decision ultimately useful in simplifying the

[1]More information available at `https://reactjs.org/`
[2]More information available at `https://www.electronjs.org/`
[3]More information available at `https://jgraph.github.io/mxgraph/`
[4]Rete.js, available at `https://rete.js.org/`

Name	License	Purpose	Seriali -zation	Undo/ redo	Palette (Dnd)	Zoom	Auto layouts	Multiple selections	Groups	Resizable nodes	Mini map	Context menus	Tooltips
Rete.js	Open-source	VPF	Yes	No	Yes	Yes	No	No	No	No	Yes	Yes	No
React-digraph	Open-source	Diagram	Yes	Yes	No	Yes	No	No	No	No	No	No	No
React-diagrams	Open-source	VPF	Yes	No	Yes	Yes	Yes	Yes	No	No	No	No	No
Draw2D	Open-source	Diagram	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes
JointJS	Open-source	Diagram	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No
mxGraph	Open-source	Diagram	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GoJS	Proprietary	Diagram	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
jsPlumb (community)	Proprietary	Diagram	No	No	No	No	No	No	No	No	No	No	No

Table 6.1: Summary of supported features of the considered frameworks/libraries.

development of the more dynamic UI components, there was an initial difficulty barrier to integrate React with mxGraph and some additional challenges arose throughout development. For instance, we wanted to render the nodes in mxGraph as React components. While this was eventually achieved, it was not as trivial as expected and required additional mechanisms to maintain the components in sync between the node components and the remainder of the UI, since the former was declared outside of the scope of the main UI window component managed by React.

In actuality, the mxGraph library was more prescriptive than anticipated, sometimes more closely resembling a framework. In particular, it includes its own internal model (essentially of a graph) upon which the internal controller operates. It also features dedicated modules to facilitate the development of external components commonly found in visual tools such as palettes and toolbars. In practice, the usage of these modules is optional and these were left unused in the prototype and were instead developed with React, as previously mentioned.

For the purposes of the prototype and due to time constraints, we opted to simplify the architecture and keep the model for Docker Compose directly as part of the UI model since the library promoted this usage. However, it is important to recognize that this approach may not be ideal or flexible enough for a more full-fledged application, since it directly ties the model to the UI library, effectively locking-in the use of this library. A future improvement would be to achieve a true Model-View-Controller (MVC) architecture in which the model would be fully independent of the view. We believe this would be fairly straight-forward to achieve (although time-consuming, hence why it was not accomplished for the purposes of a prototype) even while keeping the usage of mxGraph, through the development of an independent model for Docker Compose and a custom controller to bridge the gap between the view and its own model.

6.4 Feature Design

This section is dedicated to exhibiting the main features implemented in the prototype as well as discussing the design considerations.

Figure 6.3 displays a sample of the UI for the prototype's main view. The view is split into five panels (as labeled in Figure 6.3): the **toolbar**, **image palette**, **properties editor**, **graph editor**, and **terminal**.

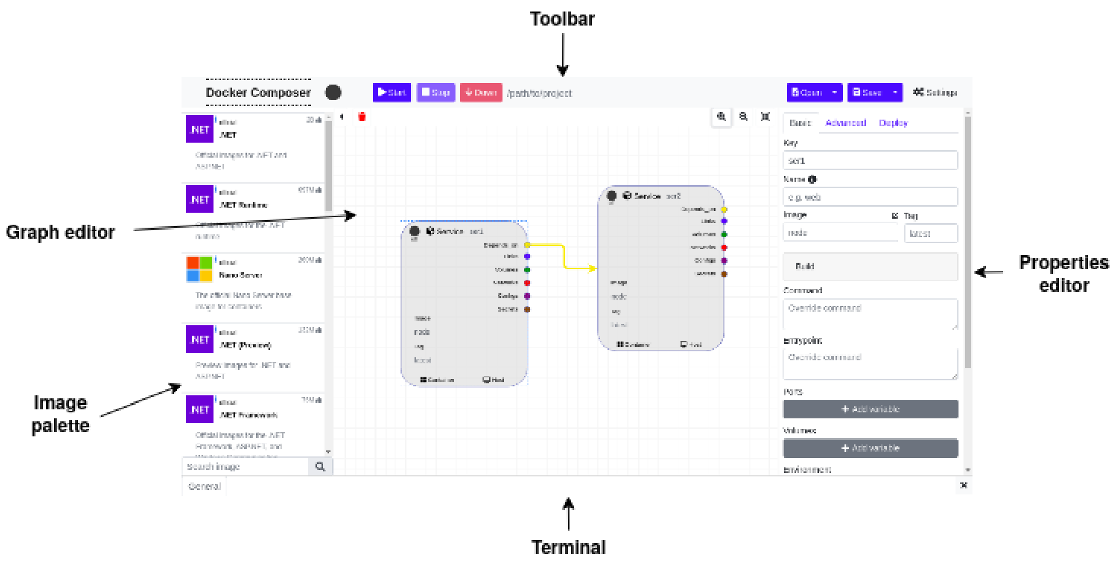


Figure 6.3: Layout of the prototype's main view.

- **Toolbar.** Located at the top of the screen. Includes to the left an area dedicated to controlling status LED which lights up different colors according to the state of the running stack and a set of buttons to control the stack (such as starting and stopping) and on the right buttons for file management. Lastly, in the settings menu, it is possible to set the working directory and adjust some output preferences when exporting files.

- **Image palette.** Allows searching for images hosted on Docker Hub and the addition of new services by clicking and dragging the target image and dropping it in the graph editor area. Includes quick access links to the image page hosted on Docker Hub by clicking on the info icon near the image.

- **Graph editor.** Split between the toolbar at the top and the canvas below. The toolbar is used to perform actions such as adjusting the zoom level and clearing the graph editor. The canvas displays an interactive visual map of the stack containing the various artifacts that comprise the stack and their relationships.

- **Properties editor.** Useful to access and edit the various properties of the currently selected object (artifact or connection) in the graph editor. It also provides quick access to some helpful links such as the Docker Compose docs and image information on Docker Hub.

- **Terminal.** Displays the output produced by the services (containers) once created and started. It contains a "General" tab with the combined output of all services and additional logs (identical to the output of executing *docker compose up*) and individual tabs for the output of services that comprise the stack.

6.4.1 Visual Map

The interactive visual map constitutes the core of the prototype and corresponds to the content displayed in the graph editor's canvas. This map corresponds to the visualization of the model of a Docker Compose stack and closely matches its textual counterpart. All of the five artifacts (declared in the top-level, e.g. services and named volumes) are represented visually as nodes, and relationships and dependencies between artifacts are represented as directed connections between nodes.

Visual Notation

In the end, a more straightforward graph-based visual notation was chosen as we felt that there was a natural mapping between the stack and its YAML textual counterpart. We struggled to find any value in using a more unorthodox metaphor adequate to this context that could potentially make the underlying concepts more understandable. In addition, while a more unorthodox visual metaphor might have had some benefits, we feel that such an approach might also obscure some of the meaning behind the original concepts which we consider important to convey to users, especially to beginners who are still learning.

This decision directly influenced the visual notation of each artifact and their relationships. The visual design of nodes was also inspired by well-established VPLs. In particular, we used Blender Nodes[5] as the primary source of inspiration.

A color scheme was also established to more intuitively express the type of connections between artifacts.

Services

The services are the basic building block of a Docker Compose stack and arguably the most important artifacts. For this reason, special attention was given to the visual notation of its node. We carefully considered what the most useful properties were to be displayed explicitly in the node and what could remain obscured in the properties editor. To keep the node size reasonable, avoid visual clutter, and not overwhelm users with too much information we ultimately decided to include inputs for image fields (name and tag) and visualization for port mappings.

Figure 6.4 displays the visual notation of a service node. The node includes a set of anchor points located on the right edge. Each anchor point is used as the source point to set connections between the service and some target artifact. This can be achieved by dragging left click with the mouse from the source point to a compatible target artifact. These connections are typed, meaning that only certain artifacts are expected as targets and the tool only allows this type of connection. To make the type of the connection more explicit, the colors of the anchor points match the color of the artifacts according to the defined color scheme, except for *depends_on* (yellow) and *links* (blue) dependencies which connect two services.

[5]Blender Nodes, more information available at https://docs.blender.org/manual/en/2.79/render/blender_render/materials/nodes/introduction.html

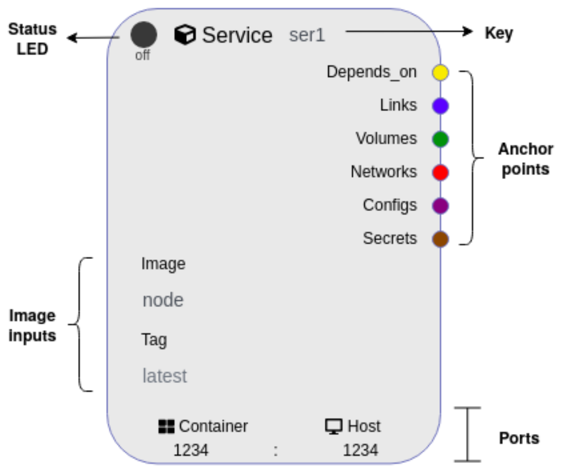


Figure 6.4: Visual representation of a service artifact node.

Others

All the remaining artifacts (shown in Figure 6.5) are represented with a similar visual notation, only differing in color, size, and labels which match the artifact type. All nodes include an input to set the key for ease of use.

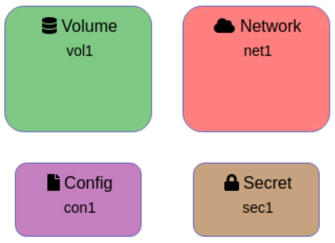


Figure 6.5: Visual representation of a volume, network, config and secret artifact nodes.

6.4.1.1 Map Layout

Importing a stack from a `docker-compose.yaml` file requires an automatic layout capability to distribute artifacts. This imposed a challenge due to the unorthodox structure of the generated map. The resulting map constitutes a directed graph where only some nodes can be connected to others (i.e. service nodes) and nodes may not be connected to anything at all. The prototype adopts a circular layout as it was deemed usable for the purpose of testing the approach.

6.4.2 Static Validation

The prototype provides some static validations while editing a stack. Examples include duplicate key detection and invalid property value formats (e.g. values specified as time durations and memory sizes). These are conveyed to the users through warning icons which appear near the visual representation of artifacts. It is possible to hover over these icons to visualize a full summary of the warnings. These inconsistencies are purely presented as warnings and are not enforced as errors and ultimately provide additional feedback to users.

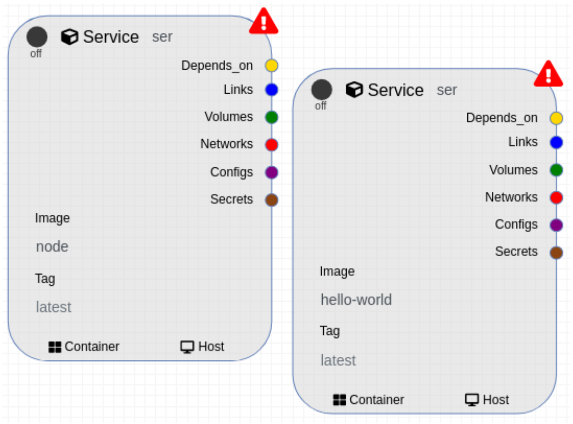


Figure 6.6: Example of the static validation notation. Both services include the warning icon because they use define the same key (ser).

Figure 6.6 displays an example of static validation. The warning in this example results from the use of the same key (ser) for both services.

Additionally, some property inputs are controlled to maintain the values consistent. One example is the definition of port mappings which disallows setting host ports without first setting a container port. This mechanism is achieved by controlling whether inputs are disabled or not.

6.4.3 Supported Versions

The goal was to support all 3.x versions of Docker Compose (up to 3.8 at the time of writing). For this purpose, we had to include inputs for all available properties across 3.x versions and manage parsing and writing according to syntax variations between versions. This addressed the limitation identified with the state-of-the-art visual solutions in Section 5.1.

6.4.4 File Management and Serialization

Two complementary import and export options were implemented. One method allows importing and exporting from and to Docker Compose YAML files while the other is useful for opening and saving from and to a custom storage method managed by the prototype. The second method, open and save, is useful to persist custom layouts of stacks designed in the tool and essentially works by serializing the model in an XML file. Without this method, information related to layout specifications such as node positioning would be lost between operations since a Docker Compose YAML definition does not consider these specifications.

We considered an alternative approach to achieve the same behavior by adding comments describing the required specifications directly in the YAML file. While this approach presented the benefit of keeping the stack always consistent since there was only one source of truth (i.e. the YAML file), it also created other potential issues in collaborative scenarios. For instance, considering a development team in which only some developers would use the visual solution while the remainder opted for the conventional text-based toolchain, the developers who used the textual toolchain could accidentally modify some of these comments and corrupt the expected structure of the file, therefore invalidating this data. Moreover, this approach would also have added more unnecessary clutter to the textual definition. Thus, we ultimately decided to maintain the methods separate and include both management options.

This feature required the development of a Docker Compose YAML parser and generator. The generator translates the internal model to code while and parser reads the contents of the YAML file and converts it into the corresponding model. For the custom storage, we took advantage of the existing serialization options provided by mxGraph and extended it to satisfy the requirements and achieve the desired behavior.

A visitor-based approach was adopted for code generation. This decision was made due to two main reasons: (1) the relatively low complexity of a Docker Compose YAML file syntax and (2) the structure of the internal model which closely mirrors that of the output itself for each artifact.

6.4.5 Executing Commands from the UI

To streamline the full orchestration process in a single environment, the prototype featured a set of buttons to manage and control the stack. These included three actions: start, stop, and down. These actions are essentially equivalent to the Docker Compose CLI commands *docker-compose up*, *docker-compose stop* and *docker-compose down*. We decided to keep executions directly tied to prototype's environment meaning that each execution is a dry-run and files produced (i.e. the `docker-compose.yaml` file) and Docker artifacts instantiated (e.g. containers and volumes) for this purpose are only temporary and deleted after exiting the tool or loading/importing another stack.

In practice, this feature was achieved by running these commands in a new process generated by the application, as demonstrated in Figure 6.1. The output of the commands is then parsed accordingly in the application.

6.4.6 Visual Feedback

Visual feedback is provided for tracking the state of running stacks in real-time. Besides the output displayed in the terminal area, LEDs display the status of each individual service and of the overall stack. A color code was defined to represent the various states.

In practice, this feature was achieved by parsing the output of the commands and updating the status accordingly. This approach was chosen because, although the Docker Engine API provides endpoints which expose the current state of services (among other information), Docker Compose includes some exclusive outputs during its execution which provide important insight into its behavior and this information is also relevant to maintain the state of the stack up to date. Thus, since we had to parse the output regardless, we opted to achieve this behavior strictly through this approach.

6.4.7 Docker Hub Integration

It is possible to search for images on Docker Hub repository directly from the image palette. Once the target image is found, the user can then drag that image from the palette to the canvas in the graph editor to directly create a new service node with the image field appropriately set. Quick links to access the page on Docker Hub are also provided. This feature was partly inspired by other state-of-the-art solutions and implemented in the prototype for ease of use. To achieve this effect, we resorted to the Docker Hub's public API. In practice, this proved more difficult to achieve than anticipated since this API is not documented and partly restricts its usage from foreign sources.

6.5 Practical Example

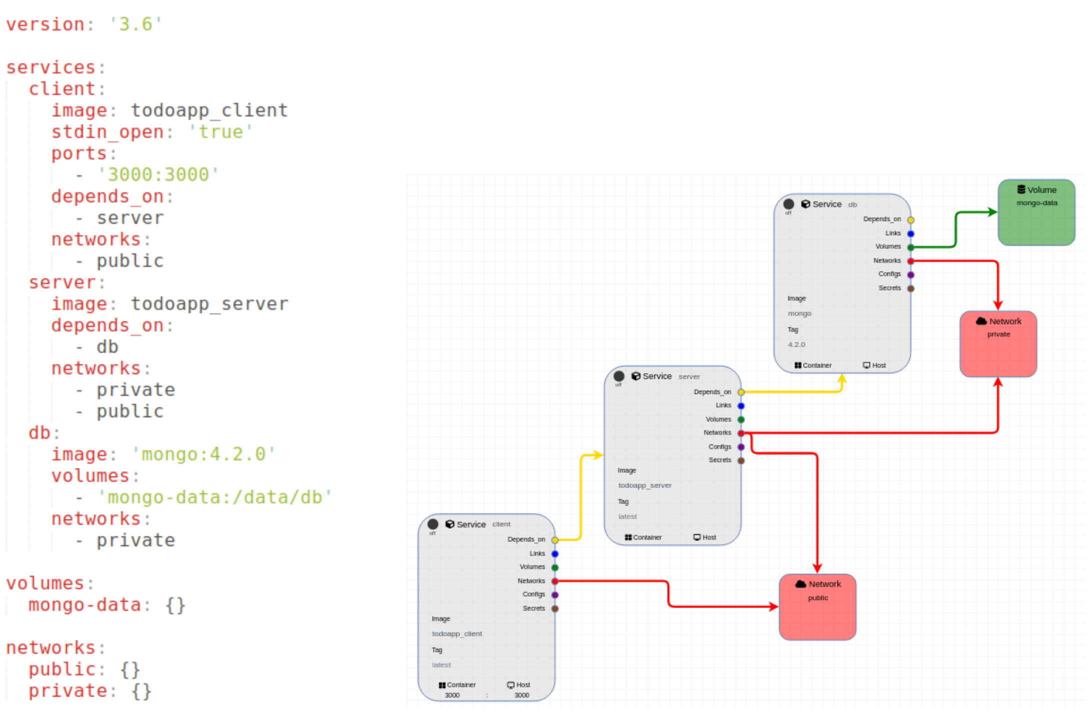
In this section, we present a practical example to more clearly compare and demonstrate the differences of representation between the conventional text-based approach and the designed visual approach. The stack used in this example was adapted from the scenario designed for one of the tasks which was part of the validation process (described in Chapter 7).

The stack follows a server-client architecture comprised of three services: a web frontend service (client), a backend web service (server), and MongoDB database service (db). We also include two custom networks, named private and public, to isolate the backend from the frontend as well as a named volume, called *mongo-data*, for data persistence.

Figure 6.7 showcases a side-by-side comparison between the textual representation (a) and the equivalent visual representation (b). While it may not be immediately clear, both representations convey the same information. While the visual approach makes the artifacts themselves and their connections more evident, some information becomes obscured, namely, the remaining properties (e.g. *stdin_open* on the client service). To mitigate this limitation of the visual approach, it is possible to hover on top of any artifact to visualize the full-textual output for that artifact.

6.6 Availability

The prototype is publicly available as open-source, under the MIT license and can be found on Github at `https://github.com/Kubix20/docker-composer`. The repository contains the source code, a release of the up-to-date version, and includes a *Wiki* containing more detailed information about the tool itself, development setup instructions and, a simplified manual with more in-depth descriptions for how to use the tool.



```yaml
version: '3.6'

services:
  client:
    image: todoapp_client
    stdin_open: 'true'
    ports:
      - '3000:3000'
    depends_on:
      - server
    networks:
      - public
  server:
    image: todoapp_server
    depends_on:
      - db
    networks:
      - private
      - public
  db:
    image: 'mongo:4.2.0'
    volumes:
      - 'mongo-data:/data/db'
    networks:
      - private

volumes:
  mongo-data: {}

networks:
  public: {}
  private: {}
```

(a) Textual (b) Visual

Figure 6.7: Comparison of the textual and visual representation of a stack

6.7 Discussion

This chapter was dedicated to exploring the developed solution prototype including its architecture and implemented features along with the reasoning behind our choices.

The designed visual approach differs from other state-of-the-art solutions as none offers a complete visual notation for the artifacts and dependencies considered in Docker Compose. This was the primary goal during the conceptual stage and throughout the development process itself.

Name	Services	Volumes	Networks	Configs	Secrets
Docker Composer	✓	✓	✓	✓	✓
Dockstation	✓	✗	✗	✗	✗
Admiral	✓	✓	✓	✗	✗

Table 6.2: Comparison of supported visual notations for Docker Compose artifacts between the developed prototype and state-of-the-art visual solutions, Dockstation and Admiral.

Table 6.2 displays the comparison of the visual notations featured in Docker Composer and those of the more closely related state-of-the-art tools — Dockstation and Admiral.

Dockstation includes the most similar notation in design to that of the developed approach, however, in its current state, is heavily limited in what it allows to define visually, lacking notations for fundamental artifacts such as volumes and networks.

Admiral on the other hand is a more powerful and mature tool mainly for deployment purposes. However it is not Docker Compose specific and, as a result, does not feature nomenclature directly matching that of Docker Compose and limits its usage due to some unsupported properties. Nevertheless, it does indeed feature a rich visual notation spanning most artifacts. However, it does not support the ability to configure configs and secrets visually.

To conclude, while other comparable state-of-the-art solutions support more features which go beyond the immediate focus of this dissertation, we were successful in achieving the goal of developing a more complete visual approach for orchestration with Docker Compose featuring a rich visual notation for the concepts we considered essential. We also addressed the remaining limitations identified in Section 5.1 while designing the prototype. More specifically the resulting solution allows the visual edition of most properties considered in Docker Compose and the design of the interface minimizes the indirectness of actions to manipulate a stack.

Chapter 7

Empirical Study

This chapter is dedicated to describing and documenting the experimental process and showcasing the results for the user study conducted to empirically validate the designed complete visual approach. The study considered two treatments — **control** in which participants solved a set of orchestration-related tasks with Docker and Docker Compose technologies by using conventional methodologies and toolchains (i.e. textual editor and terminal) and **experimental** in which participants solved the same set of tasks but used the solution prototype (described in Section 6) instead of the conventional toolchain.

7.1 Goals

The objective of this user study was to provide answers to the research questions that stem from the hypothesis (Section 5.2). To answer **RQ1**, we will consider the times to completion for each task. To answer **RQ2**, we will consider the execution attempts. In more detail, each question will be answered in the context of three distinct activities, namely, interpreting, debugging, and creating a Docker Compose stack.

Additionally, a set of perception-based questions (PBQ) was also considered for the study. We believe that it is useful to evaluate these questions as it could provide more insight into the overall value of the complete visual approach applied for this purpose. Otherwise, even if our findings may suggest that the approach improves performance it would be somewhat meaningless if participants do not find the experience enjoyable or if they have no intention of using it in the future at all. For this reason, we will also attempt to answer the following questions:

PBQ1 *Do developers find a complete visual approach easier to use than the conventional method?*

PBQ2 *Do developers find a complete visual approach useful?*

PBQ3 *Do developers want to use a complete visual approach?*

This set of questions are aligned with the definition of the *development experience* assumption stated in the hypothesis (Section 5.2). The answers to these questions will allow us to test if the designed approach satisfies this assumption.

This study also fits within the broader field of visual programming languages and more specifically the subset of VPLs for orchestration. The results of this study should contribute to demonstrating the viability and practical usefulness of visual approaches in this domain. This is particularly relevant as empirical work in this field is still fairly limited [54]. Although the theoretical benefits have been thoroughly evaluated in the past, there is still a severe lack of studies to assess whether these truly translate to practical scenarios.

7.2 Design

The evaluation focused on three basic activities in software engineering: interpreting and analyzing, debugging, and implementing applied to (in this context) Docker Compose stacks. For each activity, a task was prepared, featuring an illustrative scenario. Besides performance-based metrics such as effectiveness, completion times and execution attempts, the participants were asked to fill a form after completing each task to evaluate the perception-based metrics not measurable or quantifiable by other means, namely Perceived Ease of Use (PEOU), Perceived Usefulness (PU) and Intention to Use (ITU). PEOU refers to how much effort would be required to use the prototype, PU refers to how well the prototype satisfies the participant's needs and expectations and ITU refers to the degree that the participant wishes to use the tool in the future.

7.2.1 Participants

Recruitment was done among students, in parallel with the experimental sessions. Initially by direct communication and later through a wide-spread e-mail invitation addressed to all potential students (approximately 368). The goal was to recruit subjects who had at least some prior experience with Docker and Docker Compose, as tasks required at least some basic knowledge of these technologies. We considered potential students as those which we knew *a priori* had had exposure to these technologies based on their academic path. The main reason why we targeted students was because of their availability and ease of reach since the experiment was conducted in an academic setting. This strategy was also helpful in meeting time constraints.

In the end, a total of 16 5th year MSc in Informatics and Software Computing students participated in the experiment.

The participants were randomly distributed in half (8-8), between two groups corresponding to the treatments: control (CG) and experimental (EG). Both groups were instructed to solve the same set of tasks. On one hand, the participants in the control group were given access to a text editor to edit the stack and to a command-line shell to access the Docker and Docker Compose CLIs (i.e. the conventional toolchain). On the other, the participants in the experimental group had access to the experimental prototype to manage the stack as well as a command-line shell to execute additional commands if required (Docker related or not). In addition, both groups had complete access to the official Docker and Docker Compose documentation as well as any other resources on the internet for reference.

To verify the balance of skills between both groups, participants were asked to fill a background questionnaire which inquired them about their current experience with the technologies of interest that we had foreseen to potentially have an impact on the results (i.e. confounding factors). These included experience with visual programming tools, orchestration frameworks in general and Docker and Docker Compose. The data collected was then used to check that the two groups were in fact balanced in their skill sets.

7.2.2 Data Sources

The following data sources were used:

- The answers to the background questionnaire;
- Performance measurements during tasks, specifically global and individual task times, number of context switches, context times and number of execution attempts;
- Participant's solutions and answers to the tasks;
- The answers to the assessment questionnaire.

7.2.3 Environment

The experimental sessions were conducted remotely. We opted for a remote workstation, set up in advance with the required software and materials, made available to the participants to use for the entirety of the experimental process accessible through TeamViewer [1]. These resources included: a browser with the form already opened (Firefox was chosen for this effect), a text editor set up in the appropriate directory (Sublime Text was used for this effect), a command-line shell set up in the appropriate directory for both groups and the prototype tool for the experimental group.

7.2.4 Task Definition

As previously stated, the goal was to evaluate the behavior of the tool for three basic activities: analyzing, debugging, and implementing a stack. This effort was translated into 4 tasks each

[1] More information available at https://www.teamviewer.com/

featuring a corresponding scenario. In Task 1 (T1) a functioning stack was provided and the goal was to analyze its structure and understand the overall behavior. In Task 2 (T2) a buggy stack was provided and the goal was to debug and fix the faulty behavior. Task 3 focused on implementing and was divided into T3.1 and T3.2. The goal of T3.1 was to build a simple stack from the ground up and the goal of T3.2 was to alter the stack to use secrets. T3.1 and T3.2 correspond to an implementation and increment activity respectively.

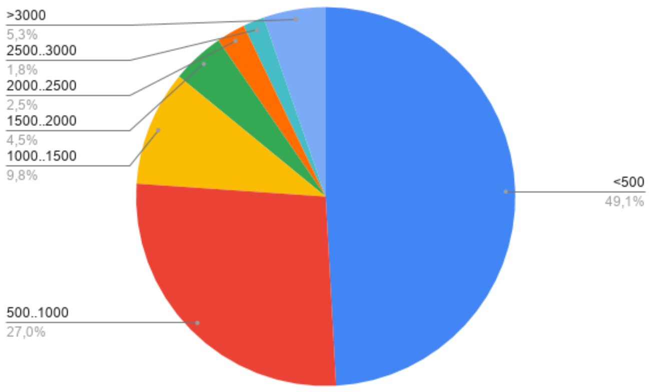


Figure 7.1: Distribution of Docker Compose YAML files on Github by size, expressed in kB.

One of the biggest concerns related to task definition was how to balance scale, complexity, realism, and expected time to completion. To guide our choices, a brief research was conducted to identify the scale distribution of Docker Compose stacks in real-life projects. The research was performed by searching projects hosted on Github. Due to several limitations of Github's advanced search engine, the obtained results were merely a rough estimate, however, this was deemed accurate enough for this purpose. The results (displayed in Figure 7.1) demonstrate that approximately 75% of the considered projects include stacks with file sizes up to 1kB, which we consider as low complexity (usually 1 to 2 services). As a result, we had more confidence in including and designing relatively trivial scenarios. At the same time, we also believed we needed some scenarios with a higher degree of complexity to take full advantage of (some of) the expected benefits of the solution. As a result, the tasks featured scenarios with stacks of varying degrees of complexity.

Another important consideration was how to communicate the purpose and inner-workings of services since Docker Compose YAML stack definitions do not provide sufficient information to demonstrate the behavior of individual services (which is hidden in the image definition). Three approaches were considered, either documenting the images, providing the source code, or a mix of both. In the end, the documentation approach was chosen as it seemed the most time-efficient although less realistic. With this approach, we could steer the focus of the participants to the aspects of interest while increasing the scale of the stack since it required less effort to understand the expected behavior.

7.2.5 Procedure

A full session took between 50 minutes to 2 hours per participant. Each session was conducted individually with the researcher overseeing and observing the full procedure. Communication between the researcher and the participant was done via a remote voice call. The subjects were encouraged to think aloud throughout the session so that the researcher could more clearly understand and follow along with their rationale. This strategy was also useful in identifying potentially unforeseen issues with the experiment's design.

Once the connection to the remote workstation was established, the participant had access to the instructions for the full procedure in a Google Form available in the remote environment. The procedure was organized in the following sections:

- **Background questionnaire.** This questionnaire contained a set of questions to assess the current degree of experience with technologies which we had foreseen to potentially be confounding factors.
- **Tutorial.** Before solving the actual tasks, the participants had to follow a simple tutorial reviewing some basics of Docker Compose. This was mostly targeted to the experimental group so that they had some prior hands-on experience with the prototype. However, to maintain consistency between both groups, participants in the control group also had to achieve the same goal with the conventional toolchain. The difference lied in the instructions provided, which were adapted according to the toolchain being used.
- **Tasks.** Participants were instructed to solve a set of four orchestration-related tasks. To maintain the total duration reasonable, time limits were set for each task. Participants were asked to advance to the next task whenever this time limit was exceeded.
- **Assessment questionnaire.** After solving the tasks, participants were asked to fill a questionnaire to assess their experience and evaluate the procedure of working with the tools. The questionnaire in the experimental group differed from the control since it included an additional set of questions to specifically evaluate the solution prototype.

The materials used can be found in full in Appendix C, in particular, the Google Form containing the description and instructions for the complete procedure.

7.2.6 Data Collection

The results of the background questionnaire were collected as a mix of answers to a 5-point Likert scale, linear numeric scales, and multiple-choice questions directly from the results of the form response sheet.

Performance measurements for tasks were recorded manually by the researcher. An application named Turns Timer[2] was used to simultaneously register the time spent on individual activities

[2]Turns Timer, is an Android application available at `https://play.google.com/store/apps/details?id=com.deakishin.yourturntimer`

and the number of changes between contexts. This was achieved by attributing a timer for each context. The sum of all the timers was the total time spent on that task. The considered contexts were as follows:

- **Script.** Time spent looking at the instructions and task description.
- **Documentation.** Time spent in the official Docker and Docker Compose documentation and Docker Hub.
- **Composer.** Time spent in the solution prototype, Docker Composer (only applicable to the experimental group)
- **Browser.** Time spent on the browser when accessing service's UIs and other documentation resources outside of those specified in the Documentation context.
- **Editor.** Time spent on the text editor to access and edit the materials.
- **Terminal.** Time spent on the terminal, mostly for executing Docker and Docker Compose CLI commands.

Participants were asked to register the start and end time for each task in the form as a redundancy precaution in case some data was lost or incorrectly recorded by the researcher. In addition, the number of execution attempts was also registered by the researcher. These performance metrics, namely, durations and execution attempts, addressed **RQ1** and **RQ2**.

Participants were also asked to save their solutions in the workstation. This was done for subsequent review if needed. The solutions considered the answers given and the developed `docker-compose.yaml` files as requested in the tasks.

PBQ1, **PBQ2** and **PBQ3** focused on perceptive-based metrics. For this purpose, the assessment questionnaire was included in the procedure. This questionnaire mostly contained Likert scale questions as well as a few open-ended questions. The former questions focused on three perception-based metrics: Perceived Ease of Use (PEOU), Perceived Usefulness (PU), and Intention to Use (ITU). It is important to note that only PEOU was measured in both groups (through a set of equivalent questions) while PU and ITU were exclusively measured in the experimental group in regards to their opinion of the prototype. This approach was adopted for PU and ITU (in opposition to measuring the same variables in both groups) because it was difficult to formulate questions which would always produce consistent answers across both groups since these metrics intrinsically assume some reference point which may be open to interpretation. We believe that, without any guidance, while the participants in the CG would more likely compare their perception in comparison to the non-existence of Docker Compose, participants in the EG would most likely compare it to the conventional method. Trying to enforce a uniform reference point would also be somewhat unnatural since it would require the participants in the EG to compare their perception to the non-existence of the prototype and Docker Compose itself. As an alternative, the questions to evaluate PU were formulated as to compare the participant's perception of the prototype in relation to the conventional method. In the latter open-ended questions, participants could share any other observations and considerations outside of the scope considered from previous questions. These were primarily useful in detecting potentially overlooked issues with the experimental procedure and even unforeseen validity threats.

A major concern was how to best collect and assess the perception-based metrics. For this purpose, we opted to follow a similar approach to that of Sandobalin et al. [57]. The work conducted by the authors closely matches the one being performed in this study since it is also an empirical study to compare model-driven to code-centric approaches for IaC. The biggest benefit of this choice was taking advantage of an already validated approach to evaluate a mostly subjective matter which is related to cognitive issues and, thus, difficult to assess quantitatively. As a result, the formulated questions to evaluate these perception-based metrics are similar to the ones used in that work, albeit adapted to the context of this user study.

7.2.7 Data Analysis

Data analysis was executed with the support of the SPSS (Statistical Package for the Social Sciences) platform[3]. All of the data (resulting from the data sources) was first manually compiled into a single spreadsheet. This spreadsheet was then loaded on the platform. A set of statistical tests were performed primarily to assess the existence of significant differences between the two groups. For this purpose, Mann-Whitney U (MW-U) and McNemar tests were performed on the variables of interest according to its characteristics.

Some data processing was also executed in Excel, such as calculating sums of variables along with most graph generation.

For all tests, a probability of 5% ($\rho < 5\%$) was considered for accepting the alternative hypothesis, in other words, a 95% level of confidence was set for the null hypothesis.

7.2.8 Pilot Experiments

Two pilot experiments were conducted to validate the quality and cohesion of the materials and of the experimental procedure itself. The first pilot demonstrated that tasks were too complex to fit within the target time-frame in the original design. As a result, the tasks were redesigned and simplified to better match the expected estimates. In the second pilot, the materials closely matched the ones used in the sessions themselves. The results of this pilot were useful in fine-tuning and refining some minor details in the materials (such as typos and other small inconsistencies) as well as streamlining the data collection process, in particular, the usage of the Turns Timer application to register the individual context times.

7.2.9 Replication

We have compiled a pseudo *replication package* to facilitate and encourage the independent replication of this experimental design. This package is available at https://github.com/

[3]SPSS, available at https://www.ibm.com/analytics/spss-statistics-software

Kubix20/docker-composer_user_study. It includes all of the materials, namely, the Google Forms with the instructions and the materials provided to participants during the tasks (for both groups) along with the raw and compiled datasets. We are designating this package as pseudo since it is partly incomplete. For instance, it is difficult to share the exact operations to reproduce the calculations as most data processing was performed with SPSS. However, we believe that the data analysis described in this chapter is thorough enough to allow the replication of this analysis. The same rationale applies to the remaining aspects left unaddressed in the package (e.g. recruitment).

7.3 Data Analysis

The collected data was mainly quantitative and was the target of the statistical tests performed. In particular, most of the hypotheses required a significance test and for this purpose, Mann-Whitney U (MW-U) and McNemar tests were run against the variables of interest based on their characteristics. The adopted notation during the analysis denotes H_0 as the null hypothesis and H_1 as the alternative hypothesis, u for the U statistic of Mann-Whitney U tests, and ρ as the probability of rejecting H_0. We also denote σ as the standard deviation and $\bar{x}$ as the mean.

7.3.1 Background

For this analysis, the answers to the background questionnaire were considered. The goal was to assess whether participants in both groups were balanced in experience and skills to ensure that differences in task performance could only be attributed to possible experience differences across both groups.

Table 7.1 displays a summary of the obtained results for the Likert and numeric scale questions. Considering the alternative hypothesis which states that the control group is different than the experimental group (CG $\neq$ EG) for every background question, the results demonstrate that there is **not a significant difference** of experience and skills between both groups except for BQ6. We believe we can disregard the disparity in BQ6 since it only applies to production scenarios which were not the focus of scenarios featured in the tasks (which were instead in line with development). Furthermore, the answers to BQ6 are based on the participant's perception of their knowledge and are always more subjective. The qualitative data which measures the number of projects (BQ7, BQ8, and BQ9) suggests that there is not a significant difference and we argue this is a stronger indicator of the participants' experience.

To consolidate the previous data, in particular the answer to BQ2 (which is purely based on the subject's self-assessment of their skills), with more quantifiable data, the participants were also asked to specify what orchestration frameworks (besides Docker Compose) they had used in the past. Since it was difficult to analyze the data considering how the experience with different tools might matter and what was the actual degree of experience with each tool (e.g. a subject may only have used a tool once), we opted to compare the number of tools between participants in both groups. For this purpose, we used a Mann-Whitney U significance test.

	CG		EG		MW-U		
	$\overline{x}$	σ	$\overline{x}$	σ	H_1	u	ρ
BQ1	2.88	0.398	3.13	0.398	$\neq$	29.0	0.372
BQ2	2.13	0.581	1.63	0.263	$\neq$	30.5	0.431
BQ3	4.00	0.189	4.13	0.350	$\neq$	25.5	0.214
BQ4	3.25	0.412	3.13	0.389	$\neq$	30.5	0.434
BQ5	3.25	0.412	2.75	0.458	$\neq$	25.0	0.223
BQ6	2.88	0.295	1.63	0.263	$\neq$	9.0	0.060
BQ7	4.38	0.822	4.88	0.515	$\neq$	26.0	0.262
BQ8	2.88	0.895	2.63	0.925	$\neq$	31.0	0.458
BQ9	3.50	1.052	3.75	0.675	$\neq$	25.5	0.244

BQ1. I consider myself experienced with visual programming tools.
BQ2. I consider myself experienced with orchestration frameworks.
BQ3. I consider myself experienced with the Linux operating system.
BQ4. I consider myself experienced with Docker
BQ5. I consider myself experienced with Docker Compose for development purposes.
BQ6. I consider myself experienced with Docker Compose in production environments.
BQ7. Until now, approximately in how many projects have you worked on which have used Docker Compose?
BQ8. Until now, approximately in how many projects have you created/updated a docker-compose.yml file?
BQ9. Until now, approximately in how many projects have you used docker-compose.yml files created by others (colleagues or third parties)?

Table 7.1: Summary of the answers to the Likert and numeric scale questions in the background questionnaire. Contains the mean and standard deviation for each group and the results of the Mann-Whitney U test.

	CG		EG		MW-U		
	$\overline{x}$	σ	$\overline{x}$	σ	H_1	u	ρ
OF	0.25	0.463	0.38	0.518	$\neq$	28	1.000

OF. Number of orchestration frameworks used

Table 7.2: Summary of the number of previously used tools specified in the background questionnaire. Contains the mean and standard deviation for each group and the results of the MW-U test.

Figure 7.2 and Table 7.2 display the data and results respectively. Considering the alternative hypothesis which states that the control group is different than the experimental group (CG $\neq$ EG) for the number of tools used, the results demonstrate that there is **not a significant difference** of experience and skills between both groups. The subjects reported having low experience with orchestration frameworks (BQ2). This was to be expected, as the use of other orchestration frameworks besides Docker Compose was not a requirement during their academic path and varied case-by-case. However, it is interesting to note that for the few participants that had in fact used another orchestration framework before, all specified Kubernetes.

Another question to complement the data obtained from the perceived experience with Docker

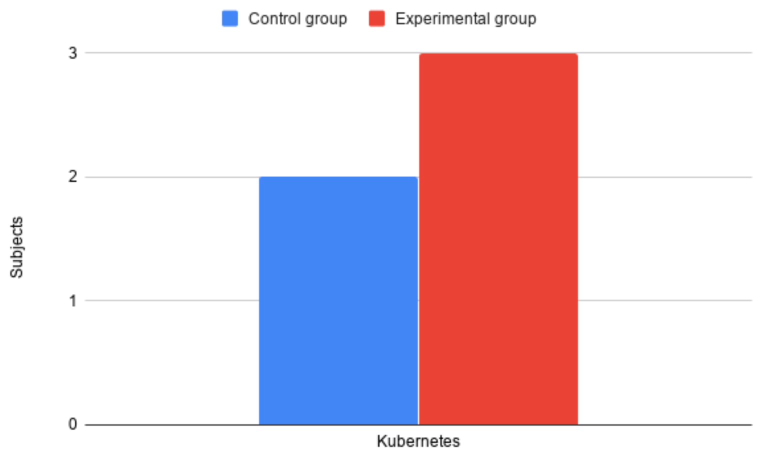


Figure 7.2: Distribution of used orchestration frameworks by group. Note that some participants specified Docker Compose but those results were not included in the graph since all subjects had prior experience with Docker Compose.

Compose inquired subjects about what individual Docker Compose artifacts they had configured in the past.

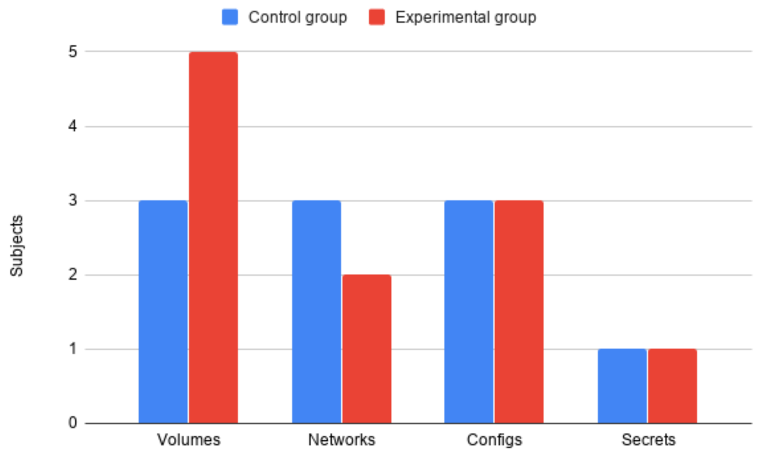


Figure 7.3: Distribution of configured Docker Compose artifacts by group.

The answers are displayed in Figure 7.3. At first glance, there does not seem to be a meaningful difference. In fact, the same amount of participants reported having configured secrets and configs, however, there is a small difference between volumes and networks. To confirm that there is not a significant difference, we ran a McNemar test to test the disparity for volumes and networks.

	CG	EG	McNemar	
Artifact	%	%	H_1	ρ
Volumes	37.5	62.5	$\neq$	0.687
Networks	37.5	25.0	$\neq$	1.000

Table 7.3: Summary of the results of the McNemar test applied to the answers to artifact configuration. Contains the percentage of subjects who have configured the artifact for each group and the significance results of the McNemar test.

Considering the results displayed in Table 7.3 and the alternative hypothesis which states that the control group is different than the experimental group (CG $\neq$ EG), the results demonstrate that there is **not a significant difference** of experience and skills between both groups. These results further support the hypothesis that there was not a significant difference in experience with Docker Compose between both groups. These results also give us more confidence that we can disregard the previously identified discrepancy for BQ6 (Table 7.1).

To conclude the background analysis, taking into account all of the data collected and corresponding analysis, we believe that we can confidently argue that the subjects were balanced across both groups.

7.3.2 Task Performance

We will now focus on analyzing the results obtained from the tasks themselves. The following statistics were considered during the analysis: task completion, overall time, overall time by activity, time by task, the overall number of context switches, and the number of execution attempts.

Task Completion

First, we will analyze the effectiveness between groups. We consider effectiveness as the ratio between successfully completed tasks and the total number of tasks. A task is successfully complete only if the subject finished within the allotted time limit and the solution was correct.

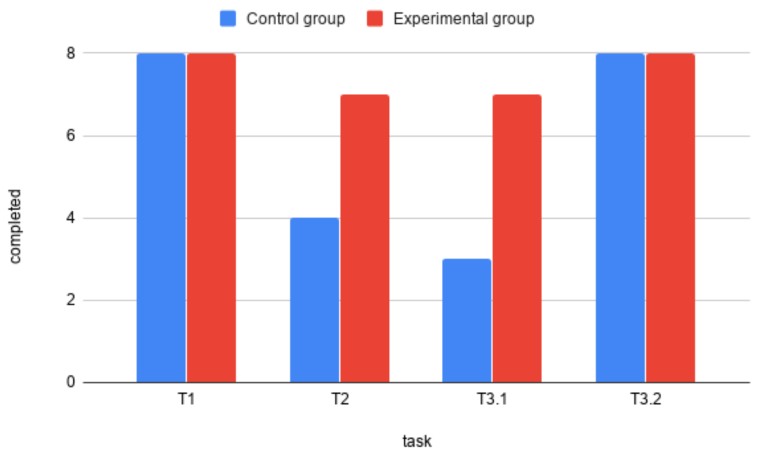


Figure 7.4: Distribution of completed tasks by group.

Figure 7.4 displays the distribution of completed tasks by group. While all subjects finished T1 and T3.2, there is a clear difference in T2 and T3.1. Almost all subjects in the EG finished the tasks while only approximately half of the participants in the CG were able to finish. Note that, for all instances, the participants were unable to complete some tasks due to time-out and no case was registered in which the solution was incorrect. Considering this data, we can conclude that fewer subjects in the CG finished the task T2 and T3.1. This in turn has an impact on the metrics considered for the remainder of this analysis since the registered values were capped up to the moment when the time limit was exceeded. If the time limit was not set the differences may have been even sharper. However, as previously stated, this was a necessary sacrifice to keep the overall time reasonable and manageable.

Context Times

As previously established, the times spent on each context were recorded for all tasks. These measurements are useful as they provide more detailed insight into the behavior of the participants while solving the tasks.

Figure 7.5 and Table 7.4 displays the global times spent on each context. To analyze this data, let us start by looking at the common contexts used by the participants, that is the contexts which were independent in both groups and do not intercept or replace each other in any way and can, therefore, be directly compared. These include the *Script*, *Documentation* (*Docs*) and *Browser*.

By looking at the data in Figure 7.5, we can identify a large discrepancy in the time spent on the *Docs* context for reading documentation. This is further supported by the discrepancy of the time spent on the *Browser* context which was also mostly dedicated to reading other non-official documentation resources. To confirm our suspicions we ran a Mann-Whitney U test for the independent contexts.

Context	CG			EG		
	$\sum$	$\overline{x}$	σ	$\sum$	$\overline{x}$	σ
Script	2:06:04	0:15:46	0:06:10	1:29:40	0:11:13	0:03:52
Composer*	-	-	-	3:50:08	0:28:46	0:10:29
Docs	1:51:36	0:13:57	0:06:25	0:18:59	0:02:22	0:02:15
Browser	0:41:44	0:05:13	0:04:02	0:11:36	0:01:27	0:01:53
Editor	2:52:17	0:21:32	0:03:57	0:04:51	0:00:36	0:00:29
Terminal	1:53:03	0:14:08	0:02:38	0:02:10	0:00:16	0:00:31
Stack Management	3:34:01	0:26:45	0:07:16	3:54:59	0:29:22	0:10:47

Table 7.4: Summary of the global time registered per activity for the sum of time taken in all tasks. Contains the mean and standard deviation for each group. *The Composer context does not contain data for the CG as this context was not available for this group and was exclusive to the EG.

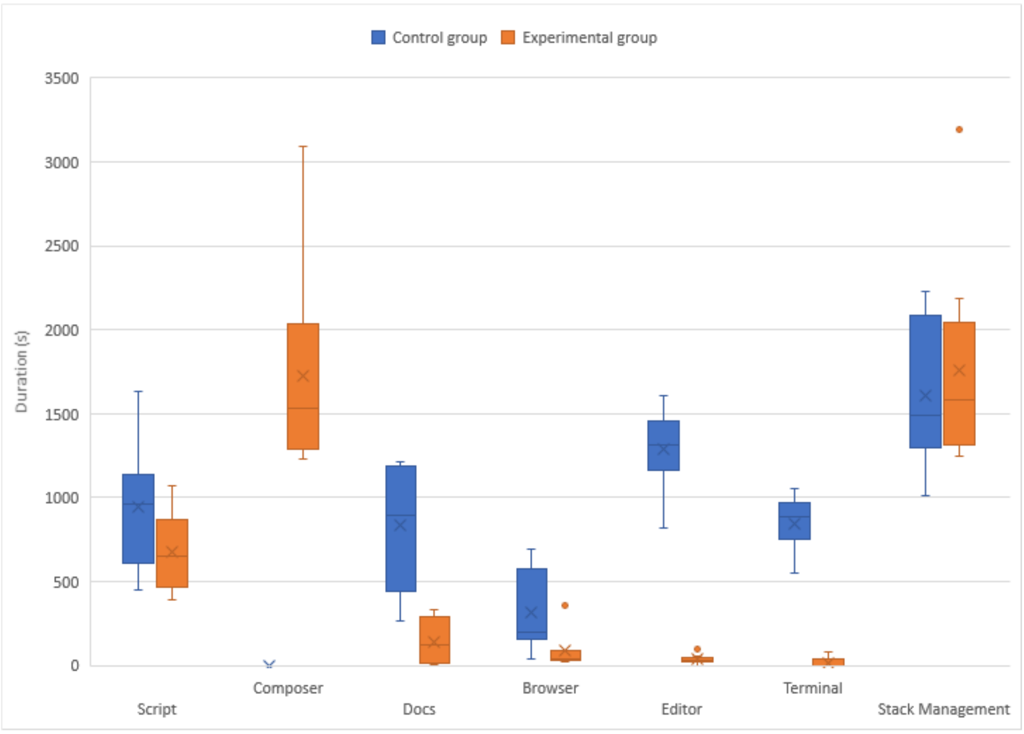


Figure 7.5: Distribution of the global times for each subject by context, by group. The Stack Management context refers to the sum of time spent on the Editor and Terminal contexts for the CG and the sum of time spent on the Editor and Composer contexts for the EG.

Considering the results in Table 7.5 and the alternative hypothesis which states that the time spent in the *Docs* and *Browser* contexts is higher for the CG, the results do indeed confirm that the CG spent **significantly longer** than the EG reading documentation. At the same time, we can conclude that there **is not a significant difference** between the groups in the time spent on the *Script* context for reading the script.

When considered individually, it is difficult to draw any other useful information from the

Context	H_1	u	ρ
Script	>	17	0.065
Docs	>	2	<0.001
Browser	>	9	0.007

Table 7.5: Result of the Mann-Whitney U equality test for the sum of time spent on three contexts. Contains the mean and standard deviation for each group.

remainder of the variables as these were either exclusive to some group (i.e. Composer for the EG) or partly replaced the purpose of one another across both groups. However, we can consider the sum of time spent on editor and terminal (ET) in the CG to be roughly equivalent to the sum of time spent on the solution prototype Composer and the textual editor (EC) — which mostly equates to the time spent accessing other textual materials such as configuration files which were used in tasks — for the EG. We can assume this to be true as no participant in the EG used the terminal to execute any other Docker or Docker Compose CLI commands besides those that were available in the prototype. We refer to this composite context as *Stack Management*. This approach provides a clearer comparison point between the activities across both groups.

Context	H_1	u	ρ
Stack Management	>	29	0.399

Table 7.6: Result of the Mann-Whitney U equality test for time differences in the Stack Management context.

Table 7.4 displays the results of the sums of the variables for both groups in the row named Stack Management. Upon closer inspection, the time difference does not appear to be very high. To test this hypothesis, a Mann-Whitney U test was performed. The results displayed on Table 7.6 do indeed confirm that the participants in the EG **did not significantly spend less time** managing the stack than those of the CG.

From this analysis, it seems reasonable to argue that the biggest impact in the overall times was the time spent looking at documentation resources. This reduction was expected as we believe that the solution promotes a more exploratory approach in which users converge to the solution by searching the available options provided by the prototype. In fact, this was the effect of a major goal of the approach, to streamline the orchestration process by presenting the underlying concepts more explicitly and making them more understandable and intuitive.

Task Times

While analyzing the times per context provides more detailed insight into the behavior of the participants, we will now look at the time spent globally (i.e. the total sum of time spent on each activity) to assess the overall speed. This metric directly answer **RQ1**.

	CG		EG		MW-U		
Task	$\overline{x}$	σ	$\overline{x}$	σ	H_1	u	ρ
T1	0:13:05	0:05:51	0:12:12	0:05:19	>	31	0.480
T2	0:22:59	0:04:55	0:14:41	0:05:59	>	11	0.014
T3.1	0:24:56	0:07:04	0:13:47	0:06:57	>	3	0.001
T3.2	0:09:35	0:04:26	0:04:01	0:01:56	>	6	0.002

Table 7.7: Summary of the times for each task across both groups. Contains the mean and standard deviation for each group and the results of the one-tailed MW-U test.

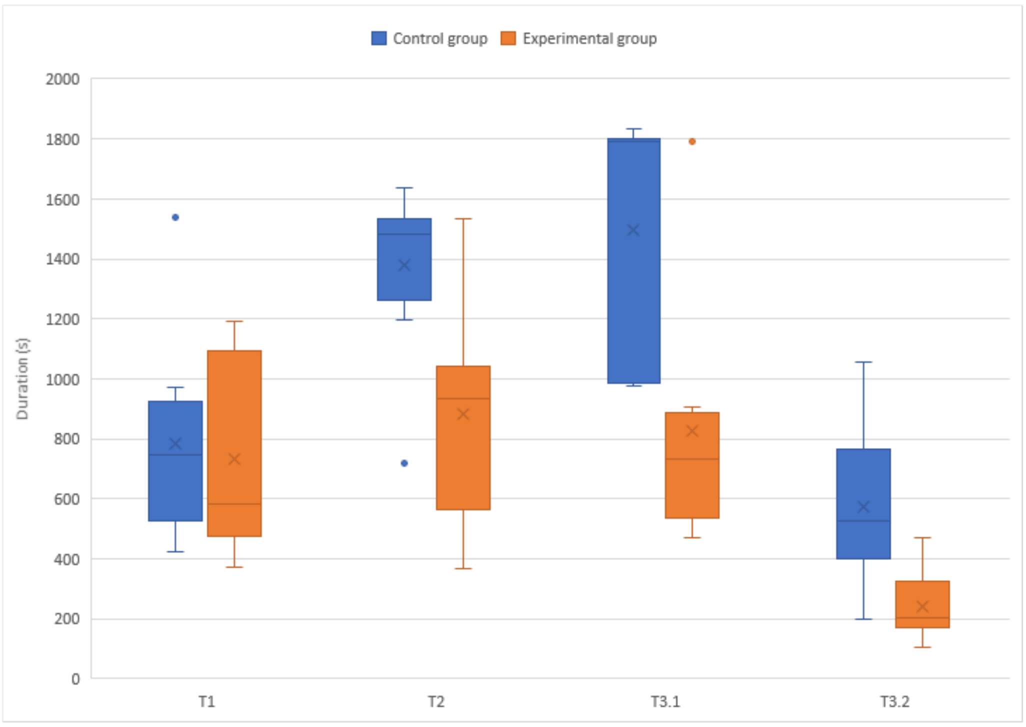


Figure 7.6: Distribution of times to completion for each subject by task, by group.

Figure 7.6 displays the distribution of times by task for each group. We can identify that the participants in the EG have generally finished tasks T2, T3.1, and T3.2 sooner than the participants in the CG. In contrast, for task T1, both groups are more balanced.

Table 7.7 summarizes the results obtained for the times of each task along with the results of the MW-U significance test performed to compare both. By considering this data and the expected alternative hypothesis which states that the participants in the EG would finish tasks faster than those of the CG (i.e. CG > EG) for all tasks, the results demonstrate that EG did indeed **finish task T2, T3.1 and T3.2 significantly faster** than the CG. These tasks evaluated debugging, implementing, and updating a stack activities respectively. Particularly, it is interesting to note the significant difference in T3.2. The scenario in this task required the participants to use a somewhat uncommon feature of Docker Compose — secrets — with which most did in fact not have any prior experience. In practice, the workflow to utilize this feature in the prototype was very similar to that of other artifacts, such as volumes and networks. These results support that the prototype was sufficiently intuitive for participants to learn how to use this new feature, after having some experience with it, simply by following a similar rationale and without the need to consult additional documentation.

These results in conjunction with the overall times allow us to answer **RQ1**. The prototype was successful in reducing the overall time required to develop and debug stacks. While there was not a meaningful improvement for task T1 (in which participants had the goal of analyzing a stack), overall, the prototype managed to reduce the time for the remainder of tasks. Some of the questions in T1 required a deeper knowledge of concepts that were not immediately conveyed by the prototype. We suspect that, taking into account that although the participants in EG already had some hands-on experience with the prototype during the tutorial, T1 being the first task in conjunction with the more complex questions contributed towards the longer durations since the participants spent some time exploring the features of the prototype searching for answers.

Execution Attempts

In addition to the task times, the execution attempts were also registered for each task, that is, the number of times a participant tried to run the stack (in practice, executing the command *docker-compose up*). This metric is useful in measuring the error-proneness during the development process and directly answers **RQ2**.

	CG		EG		MW-U		
Task	$\bar{x}$	σ	$\bar{x}$	σ	H_1	u	ρ
T1	0.38	0.518	0.50	0.535	>	28.0	0.500
T2	7.00	3.928	5.63	2.560	>	21.5	0.134
T3.1	10.13	4.357	5.25	4.097	>	11.5	0.014
T3.2	3:50	1.690	1.75	0.463	>	13.0	0.016

Table 7.8: Summary of the execution attempts for each task across both groups. Contains the mean and standard deviation for each group and the results of the MW-U test.

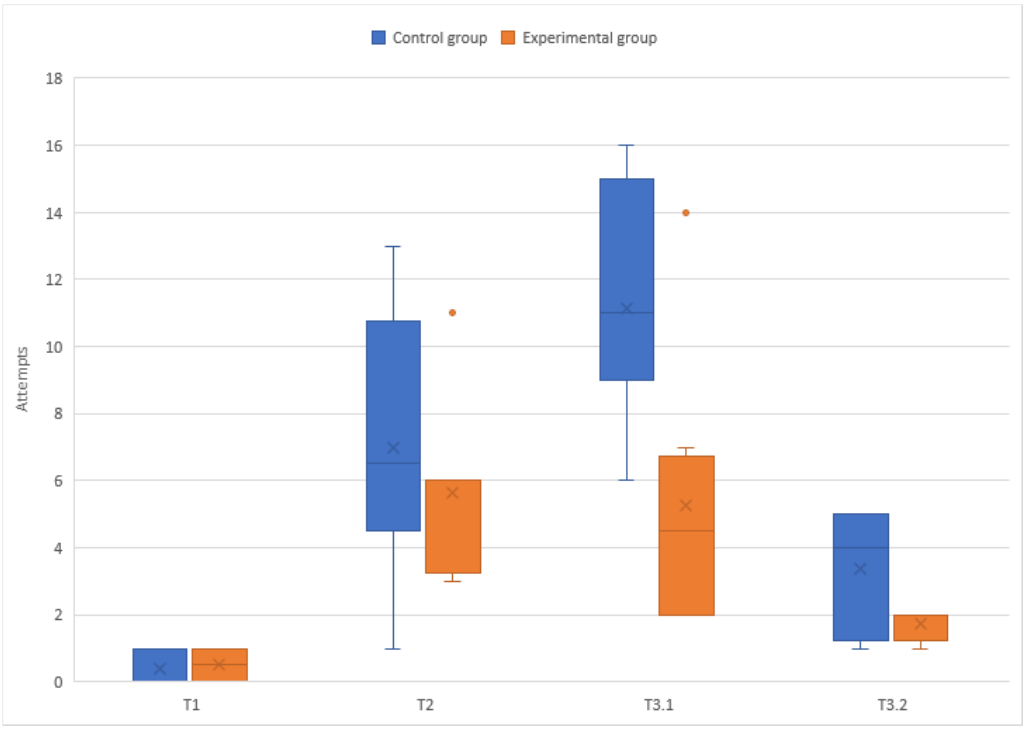


Figure 7.7: Distribution of execution attempts for each subject by task, by group.

Figure 7.7 displays the distribution of execution attempts by task for each group. We can identify that the participants in the EG have generally performed fewer execution attempts in tasks T2, T3.1, and T3.2 than the participants in the CG. In contrast, for task T1, both groups are more balanced. However, it is important to note that the results for T1 are not very revealing as the execution of the stack was completely optional for this task.

Table 7.8 displays the results for execution attempts. Considering this data and the expected alternative hypothesis which states that the EG would need fewer execution attempts than the CG (i.e. CG > EG) for all tasks, the results demonstrate that EG did indeed require **significantly fewer execution attempts for T3.1 and T3.2**.

These results are in line with the time difference established above. Overall, the participants in the EG were more efficient and did not spend as much time restarting the stacks. This behavior was also expected as in practice many execution attempts in the CG resulted from syntax errors. The prototype avoided most syntax errors simply due to the more strict form inputs (with stronger validation) and subsequent automatic code generation which was free of errors. We believe that this was the biggest factor to contribute to the non-significant difference in T2 since in this task a partially working stack was provided and few changes were required.

Context Switches

In addition to the context times, the **context switches** were also recorded, that is, the number of times the participant accessed each of the contexts. To keep this metric uniform across the participants we will consider the context switches per minute (s/m) as opposed to the total count of context switches. Although this metric does not provide a direct answer to any of the stated goals, it is useful in evaluating the degree of focus of participants which has a substantial impact on its usability. We argue that, in general, a higher number of context switches translates into a less optimized experience since a user has to shift their attention more frequently, therefore being more distracting.

We will analyze the global context switches during the full session, that is, the total sum of the switches between all contexts for all tasks.

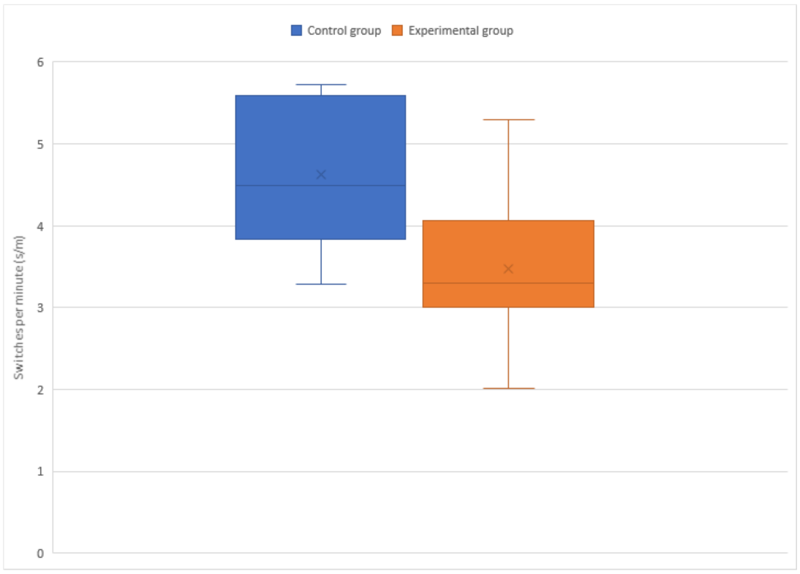


Figure 7.8: Distribution of global context switches for each subject by group.

	CG		EG		MW-U		
s/m	$\bar{x}$	σ	$\bar{x}$	σ	H_1	u	ρ
Global	4.628	0.912	3.479	0.929	<	10	0.010

Table 7.9: Results of MW-U test for global context changes. Contains the mean and standard deviation for each group and the results of the test.

By analyzing the data in Figure 7.8, it seems that the participants in the EG performed fewer context switches than those of the CG. To confirm our suspicions, we ran a MW-U test.

The results of the test are displayed in Table 7.9. Considering this data and the alternative hypothesis which states that subjects in the EG would execute fewer context switches than those in the CG (i.e. CG > EG) overall, the results demonstrate that the participants in the EG did, in fact, **execute significantly fewer context switches** than those in the CG. These results support that the process was more streamlined for participants in the EG and thus more optimized. This outcome is directly in line with the results of the task time analysis performed previously.

Task 1 Results

In task T1, subjects were asked to answer a set of questions to assess their interpretation of a Docker Compose stack which was provided for this end. In this section, we analyzed the score results for these questions. The questions were split between two exercises: (1) considered true or false statements and (2) short answers in a specific format. For the answers in exercise 2, a strict all-or-nothing correction criterion was adopted in which an answer would only be considered as correct if it fully matched the expected solution and incorrect otherwise.

	CG	EG	McNemar	
	%	%	H_1	ρ
Q2	50.0	75.0	>	0.313
Q7	25.0	37.5	>	0.500

Table 7.10: Summary of the results of the McNemar test applied to the answers of questions Q2 and Q7 of task T1. Contains the percentage of subjects who have answered correctly for each group and the significance results of the McNemar test.

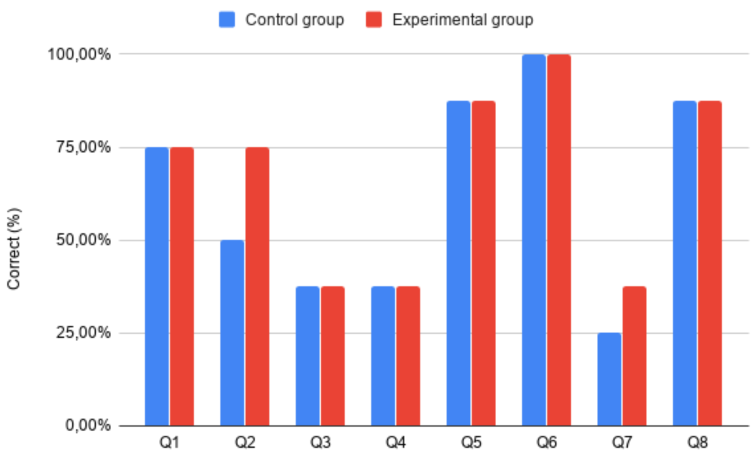


Figure 7.9: Percentage of subjects who answered the questions of task T1 correctly. **Q1.**

Figure 7.9 displays the percentage of participants which answered correctly for each of the 8 questions considered in both exercises. We can see that there is only a minor difference between Q2 and Q7, in which more subjects of the EG answered correctly. To test if this difference is significant we will run a Mcnemar test for both questions (Q2 and Q7).

Table 7.10 summarizes the result of the McNemar test. Considering this data and the expected alternative hypothesis which states that more participants in the EG would answer the questions correctly than those of the CG (CG < EG), the results demonstrate that **not significantly more subjects of the EG answered questions correctly** than those of the CG for any of the questions. Taking into account these results, we can conclude that the prototype was not successful in increasing the correctness of answers. We believe that these results are partly attributed to the design of the questions. Some required a deeper knowledge of Docker Compose including some concepts which were not directly conveyed by prototype while others required a higher level of indirectness such as searching for the answer on the image documentation in Docker Hub. Despite this, it is still interesting to note that the results were similar across groups even though the participants in the EG relied far less on documentation resources. These results may suggest that the knowledge gained from the information conveyed by the prototype was comparable to that of reading documentation resources, at least for the concepts addressed in the task.

7.3.3 Assessment Questionnaire

Similarly to the analysis performed for the background questionnaire, MW-U tests were used to assess potentially significant differences between both groups. As previously mentioned, the contents of this questionnaire differed between groups. It featured a set of identical questions to evaluate possible environmental deviations and procedure understandably along with a set of equivalent questions to assess PEOU (except for 1 question to the EG since it was not applicable to the CG). The questionnaire provided to EG featured an additional set of questions to evaluate the perceived usefulness of individual features and overall PU as well as ITU sentiment towards the prototype. The questions which focused on PU were formulated to compare the usefulness of prototype in relation to the participant's perception of the conventional method and toolchain.

Environment

The goal of this set of questions was to assess possible deviations resulting from the effect of external environmental factors. We had extra attention in this regard as the sessions were being conducted remotely.

	CG		EG		MW-U		
	$\bar{x}$	σ	$\bar{x}$	σ	H_1	u	ρ
ENV1	4.13	1.356	3.75	1.282	$\neq$	25.5	0.231
ENV2	1.88	1.356	2.13	1.458	$\neq$	29.0	0.367

ENV1. It was easy working in the remote machine.
ENV2. The environment was distracting.

Table 7.11: Summary of the the answers to the common Likert-scale questions in the assessment questionnaire. Contains the mean and standard deviation for each group and the results of the MW-U test.

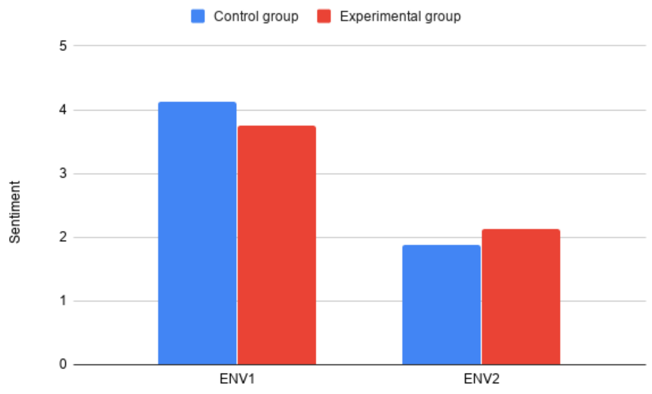


Figure 7.10: Mean of answers to the Likert scale questions related to environment factors for each subject, by question.

By analyzing the data in Table 7.11 and considering the alternative hypothesis which states that the perception of the CG of environment factors differs from the EG for all environment-related questions, the results demonstrate that there is **not a significant difference** between both groups. These results support the hypothesis that the influence of environmental factors on performance during tasks was balanced across both groups and therefore did not have a meaningful impact on the results.

Perceived process understandability

The goal of this set of questions was to assess possible bias between groups resulting from deviations in the understandability of the provided materials and instructions.

	CG		EG		MW-U		
	$\bar{x}$	σ	$\bar{x}$	σ	H_1	u	ρ
PPU1	2.25	1.282	1.25	0.463	$\neq$	17.0	0.080
PPU2	3.00	0.926	1.50	0.756	$\neq$	6.5	0.005

PPU1. I found the procedure instructions complex and difficult to follow.
PPU2. I found the task descriptions complex and difficult to follow.

Table 7.12: Summary of the results of the answers to the common Likert-scale questions in the assessment questionnaire. Contains the mean and standard deviation for each group and the results of the MW-U test.

It is important to note the discrepancy of the PPU between both groups. As demonstrated in Table 7.12, there is a **significant difference** for the PPU in regards to the task description across both groups. In particular, by analyzing the means (Figure 7.11) we can conclude that the participants in the EG reported finding the task descriptions more understandable than those of the CG. Although this discrepancy could potentially imply a threat to validity since it could entail that differences in performance could be related to the difficulty in understanding the instructions, we

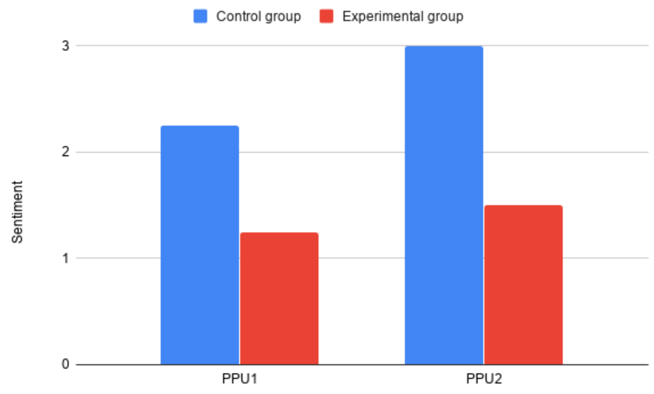


Figure 7.11: Mean of answers to the Likert scale questions related to perceived process understandability for each subject, by question.

argue that it may have been the result of the more pronounced difficulties felt by the participants of the CG, who generally struggled more to complete tasks, therefore, influencing their judgment. Another possibility is that the participants interpreted the questions wrong and mistook the intent of the statement with ease of implementation. In contrast, the prototype used by the participants in the EG provided a more streamlined and focused experience (as supported by the lower context switching) which helped participants to concentrate more clearly on the provided instructions and were as a result less fatigued by the end of the sessions.

Perceived Ease of Use

The goal of this set of questions was to assess the perceived ease of use (PEOU) and answer the question **PBQ1**.

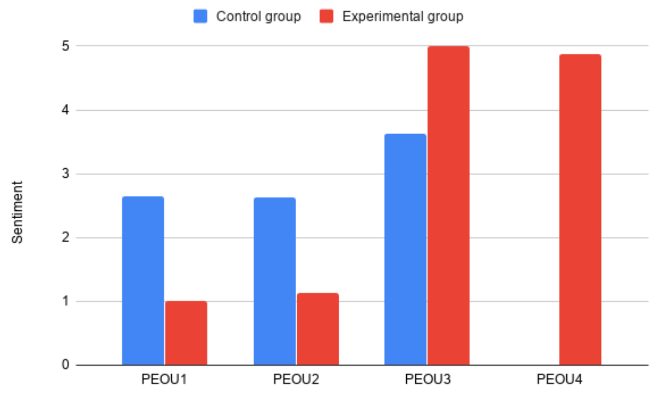


Figure 7.12: Mean of answers to the Likert scale questions related to perceived ease of use for each subject

	CG		EG		MW-U		
	$\overline{x}$	σ	$\overline{x}$	σ	H_1	u	ρ
PEOU1	2.64	1.188	1.00	0.000	>	8.0	0.002
PEOU2	2.63	0.916	1.13	0.354	>	5.5	0.001
PEOU3	3.63	0.744	5.00	0.000	<	4.0	0.001
PEOU4*	-	-	4.88	0.354	-	-	-

PEOU1. Overall, I found the tool difficult to use.
PEOU2. I found it difficult to understand stacks with the tool.
PEOU3. I found it easy to define stacks with the tool.
PEOU4. Overall, I found the tool easy to learn.

Table 7.13: Summary of the results of the answers to the Likert-scale questions related to perceived ease of use (PEOU) in the assessment questionnaire. Contains the mean and standard deviation for each group and the results of the MW-U test. *PEOU4 does not contain data for the CG as this question was exclusive to the EG.

By analyzing the data in Table 7.13 and considering the hypothesis which states that participants in the EG would find the prototype easier to use (i.e. CG > EG for PEOU1 and PEOU2 and CG < EG for PEOU3) for all equivalent PEOU questions, the results demonstrate that the EG did indeed find that it was **significantly easier to work with the prototype**. Therefore we can answer question **PBQ1** with some confidence that participants did in fact find the prototype easier to use than the conventional method. Additionally, the exclusive question PEOU4 also demonstrates that the participants in EG strongly agreed the tool easy to learn.

Features

The following sections focus on the questions which were exclusive to the questionnaire provided to the EG.

The questionnaire provided to the EG contained a section dedicated to evaluating the perceived usefulness of individual features, namely, the visual map of artifacts (VM), Docker Hub integration (DHI), visual feedback (VF), and executing commands on the UI (UIC). The goal of these questions was to assess the perceived usefulness with a higher degree of granularity so that we could better understand the impact of each feature in the overall perception. Table 7.14 summarizes the obtained results.

	VM		DHI		VF		UIC	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
ULE	4.88	0.354	4.00	1.195	4.50	0.756	-	-
UQ	4.75	0.463	4.00	1.195	4.75	0.707	-	-
DLE	4.88	0.354	4.00	1.195	-	-	4.75	0.707
DQ	4.88	0.354	4.00	1.195	-	-	4.75	0.707

ULE. Understand stacks with less effort.
UQ. Understand stacks more quickly.
DLE. Define stacks with less effort.
DQ. Define stacks more quickly.

Table 7.14: Summary of the results of the answers to the Likert-scale questions related to perceived usefulness of features in the assessment questionnaire. The cells marked with - signify that the questions on the corresponding row(s) was not part of the questionnaire.

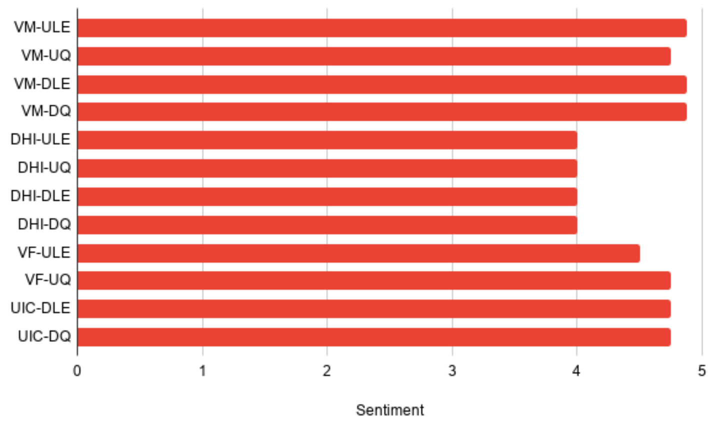


Figure 7.13: Mean of answers to the Likert scale questions related to perceived usefulness of features for each subject of the EG.

Considering the data displayed in Table 7.14, we can conclude that the feature considered most useful was the visual map of artifacts (VM) while the least was the Docker Hub integration (DHI). These results matched our expectations as the DHI feature was secondary and mostly added for ease-of-use and convenience. Moreover, the designed tasks did not take full advantage of this feature since subjects were able to copy and paste the image names and tags from the provided script without the need to locate them manually. In contrast, the VM feature was the direct result of the hypothesis of this dissertation and corresponded to the most novel and premeditated feature. Regardless, the response was positive for all features.

Perceived Usefulness

The goal of this set of questions was to assess perceived usefulness (PU) and answer question **PBQ2**.

	$\bar{x}$	σ
PU1	5.00	0.000
PU2	5.00	0.000
PU3	1.50	0.756
PU4	1.00	0.000
PU5	4.88	0.354

Table 7.15: Summary of the results of the answers to the Likert-scale questions related to the overall perceived usefulness in the assessment questionnaire.

PU1 I believe this tool would reduce the effort required to define Docker Compose stacks.
PU2 Overall, I found the tool useful.
PU3 A Docker Compose stack visualized with the tool would be more difficult to understand.
PU4 Overall, I think this tool does not provide an effective solution to define Docker Compose stacks.
PU5 Overall, I think this tool makes an improvement to the stack definition process.

List 7.1: Question identifiers for Table 7.15 and Figure 7.14.

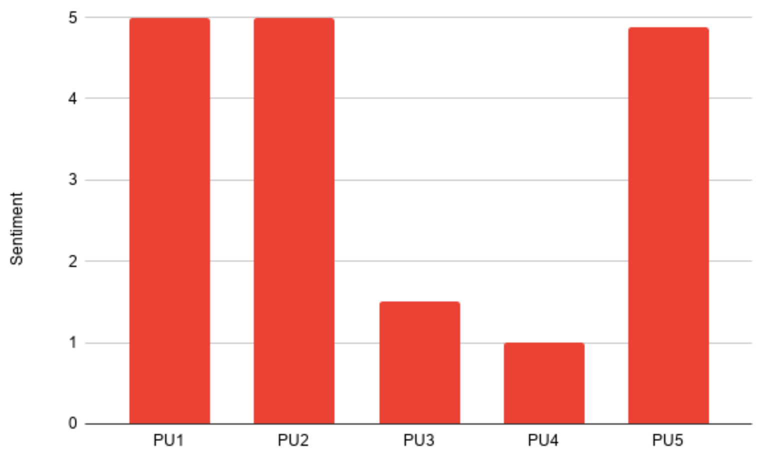


Figure 7.14: Mean of answers to the Likert scale questions related to Perceived Usefulness for each subject of the EG.

Taking into the account the combined feedback of the usefulness of individual features (Table 7.14 and Figure 7.13) and overall usefulness (Table 7.15 and Figure 7.14), we can answer the question **PBQ2** with some confidence, that subjects did indeed find the tool useful.

Intention to Use

The goal of this set of questions was to assess intention to use (ITU) and answer question **PBQ3**.

	$\bar{x}$	σ
ITU1	4.50	0.535
ITU2	4.75	0.463
ITU3	5.00	0.000
ITU4	4.50	0.756
ITU5	4.75	0.463

Table 7.16: Summary of the results of the answers to the Likert scale questions related to the overall perceived usefulness in the assessment questionnaire.

ITU1 This tool would make it easier for practitioners to define Docker Compose stacks.

ITU2 Using this tool would make it easier to communicate the stack architecture to other practitioners.

ITU3 I would recommend this tool to work with Docker Compose.

ITU4 I would like to use this tool in the future.

ITU5 It would be easy for me to become skillful in using this tool to work with Docker Compose.

List 7.2: Question identifiers for Table 7.16 and Figure 7.15.

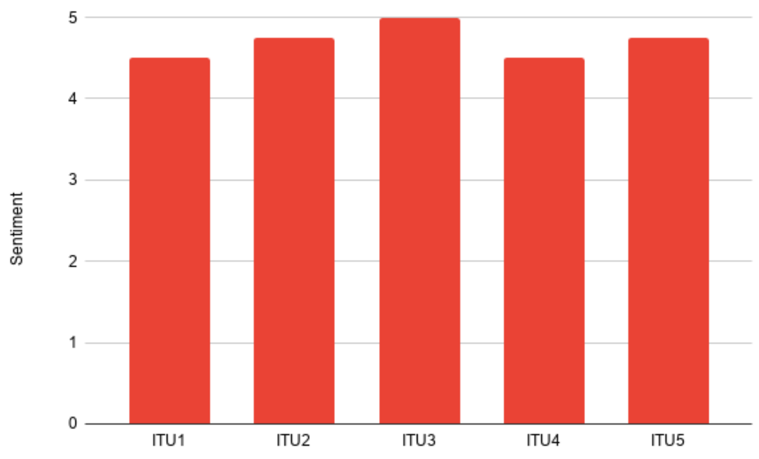


Figure 7.15: Mean of answers to the Likert scale questions related to Intention to Use for each subject of the EG

Taking into account the results displayed on Table 7.16 and Figure 7.15, we can answer the question **PBQ3** with some confidence, that subjects do indeed intend to use the tool in the future.

Overall, the results demonstrate that the response to the prototype was overwhelmingly positive and generally very consistent across participants. The participants found the approach easier to use than the conventional method, generally useful, and were interested in using it in the future.

7.4 Validation Threats

This section is dedicated to identifying and discussing threats which might hinder the soundness of the obtained results. The following threats were identified:

- **Psychological bias.** For the results to be unbiased, it is important to ensure that participants are unaware of what group they belong to. However, in practice, this was hard to achieve for all cases due to the design of the experiment which made it somewhat obvious to the participants in the experimental group. Nevertheless, all possible efforts were made to mitigate this threat, particularly, by preparing the materials as to omit any information and avoiding any verbal exchange during the experiments themselves which could allude to this fact.

- **Scarce sample size.** The somewhat small sample size was mostly attributed to time constraints in part because of the need to execute each session individually and lack of availability of some participants which hampered recruitment. To confirm the findings with more confidence, it is important to rerun the experiment with a larger sample size to mitigate this threat.

- **Experience differences.** It is crucial to ensure that the results are independent of possible skills and experience differences between groups. For this reason, the background questionnaire was part of the process and the data analysis supports that both groups were balanced. Therefore, we believe we can discard this threat.

- **Environment influences.** Performing the sessions remotely raised additional concerns in regards to possible deviations due to uncontrolled external factors. However, we believe that with the remote workstation approach we were able to virtually achieve a consistent environment for all participants. In addition, the researcher's constant observation during the sessions was also useful in identifying any unforeseen anomalies. The results in the assessment questionnaire further support that there was not any significant difference between groups. Thus, we believe we can discard this threat.

- **Sample bias.** The participants were all students with similar backgrounds. While this helped to ensure that there was not a significant experience disparity between groups, it may have caused a possible bias in the results and therefore fail to fully represent the more heterogeneous population. However, the research conducted by Host et al. [35] concluded that 5th students can be appropriate subjects, if properly trained, as they represent the future generations of developers. Thus, we believe the results are meaningful. In spite of this, further studies are required, featuring a more heterogeneous sample with more diverse backgrounds to achieve results with more confidence and mitigate this threat.

- **Auto-layout inefficiencies.** As described in Chapter 6, the prototype implemented an automatic layout algorithm to position artifacts when loading a stack from a YAML file. However, the results achieved were sub-optimal. To mitigate possible deviations resulting from this limitation, the stacks provided to the experimental group were prepared in advance to a more readable format and were loaded using the custom storage feature. In practice, this did not seem to influence the results as no participant suggested this improvement, therefore we can discard this threat.

We made an effort to mitigate validation threats throughout experimental planning and design phases. We argue that the overall balance is positive as we were able to offset most in this experimental design.

7.5 Summary

In this chapter, the design and results of the conducted user study were described and discussed.

To summarize, our findings support the hypothesis that a complete visual approach for orchestration does indeed generally reduce development time **and** error-proneness significantly. However, although the overall balance was positive, the improvements are not reflected in all activities. In particular, the biggest benefits were recorded during implementation-based tasks (T3.1 and T3.2). We can also conclude that in regard to time-related improvements, the results suggest that the observed gains were mostly due to reductions of time spent on reading documentation. Additionally, the analysis of the context switches suggests that the experience was overall more streamlined when using the prototype.

The findings related to perception-based metrics, although less objective, were positive as well. The participants overwhelmingly felt that the prototype was easier to use, was generally useful, and manifested strong intention of using it in the future. These results demonstrate that the prototype was successful in satisfying the established definition of *development experience* in the hypothesis.

While the obtained results are promising, additional research would be useful to further consolidate and confirm our findings with more confidence. In this sense, rerunning the study in other contexts (in which some of the most dangerous validity threats were offset) would be highly useful. In particular, we consider essential testing the approach with a bigger and more diverse sample, especially with members of the industry.

Chapter 8

Conclusions and Future Work

Cloud computing is an evolving paradigm and has become the *de facto* hosting solution. Simultaneously and partly as a reaction to the cloud revolution, microservices architectures are gradually becoming the preferred way to build software, replacing traditional monolithic systems, by providing low coupling and high cohesion. These architectures are heavily dependent on containers and these require comprehensive tools to manage them throughout their life-cycle.

The state of the art review demonstrated that the field surrounding cloud computing is a focal point of academic research and is already rich in technologies and tools. In fact, some progress has already been made towards solutions that allow the developer to manage systems at higher abstraction levels through model-driven and visual programming approaches. Working at high abstraction levels points to achieving faster development velocity, higher productivity, and improved maintainability.

In this dissertation, we aimed to contribute to a piece of the puzzle that comprises the current complex cloud ecosystem by expanding upon existing efforts in container orchestration technologies. This aim led us to explore the potential of a complete visual approach for the orchestration of technological stacks. In practice, we have limited the scope to Docker Compose and Docker technologies as the basis technologies for our research.

8.1 Hypothesis Revisited and Contributions

We have started with the following hypothesis:

H: *A complete visual programming approach for developing orchestration recipes improves the overall developer experience and reduces the error proneness and development time.*

We then designed and developed a prototype using a complete visual approach by combining techniques from model-driven engineering and visual programming. The prototype was afterward used to empirically validate the approach in a user study conducted among students and the results of this user study allowed us to test the hypothesis. We concluded that the results generally support the hypothesis, both in terms of performance improvements (development time and error-proneness) and user experience (as defined in the stated hypothesis).

To summarize, the contributions of this dissertation are as follows:

- **A study of the challenges of working with Docker Compose.** The study conducted with students to identify practical issues developers find when working with Docker technologies (Chapter 4). Although the findings of this study were mostly inconclusive for the purposes of this work, the results are still meaningful and we believe they may also be of interest to a wider scientific community.

- **A visual programming environment for orchestration with Docker Compose.** A prototype serving as a visual programming environment was developed leveraging the designed complete visual approach for orchestration with Docker Compose as described in Chapter 6.

- **An experimental design to validate the approach.** We have designed and conducted a user study to empirically validate the visual approach and test the hypothesis (Chapter 7). The resulting experimental design is fully documented and can be replicated. We believe this design may be used not only to validate this approach in other contexts but even to validate alternative approaches (visual or otherwise) in this field applied to Docker and Docker Compose technologies.

8.2 Future Work

As is common in research projects, new ideas are always on the horizon. We dedicate this section to documenting ideas for expansions and improvements which go beyond the immediate objective of this dissertation and propose them as future work to further consolidate and complete the work develop thus far. We will address three topics: the visual approach, the developed prototype and empirical validation.

Visual Approach

As previously stated, the scope was limited to Docker Compose. As a result, the designed visual approach is, as expected, highly tied to the underlying concepts of Docker Compose. However, we believe that similar visual approaches are applicable in a broader context, in particular, to other orchestration technologies. It would be interesting to explore domain-specific visual notations for other orchestration technologies (e.g. Kubernetes). Furthermore, one can even consider the possibility of a more generic and technology agnostic model-driven approach, useful for a wider set of use-cases. The positive results obtained in this work in conjunction with the swift paradigm shift in software engineering towards microservices architectures provide strong motivation to promote research in this field.

Prototype

We also propose evolving the prototype to a full-fledged application by expanding upon existing features as well as exploring other ideas which go beyond the immediate objective of this dissertation which we believe may further improve the orchestration process. Some of these ideas stem from the conceptual stages of the implementation but were ultimately not realized as they did not directly contribute towards our goal while others are the result of how we foresee the prototype could evolve.

- **Textual editor.** This idea explores the inclusion of a textual editor which would work in parallel with the graphical editor, similarly to the feature offered in Dockstation. However, unlike Dockstation, we propose the simultaneous view of both perspectives in a single screen. This would require real-time sync mechanisms to maintain both views consistent on change in either one and could be achieved through MDSE bidirectional transformation techniques as demonstrated in the state of the art review. We consider this as one of the most significant improvements since it could impact the potential target audience of the prototype, as many developers (especially more experienced) seem to prefer working with textual methods. This addition would mean that the solution would not substitute the conventional method and would instead complement it with more information and options. However, in the end, this feature was deemed as secondary since it did not directly impact the hypothesis and its exclusion contributed towards making the validation between the VPL and text-based method clearer. Additionally, some tools identified in the state of the art analysis (Chapter 3) already achieve a similar behavior (e.g. Micro:Bit), meaning there was reduced scientific value in replicating it. Regardless, we strongly believe this would have been a great addition, allowing a more complete experience which takes advantage of both edition methods simultaneously.

- **Automatic layout.** As previously noted the automatic layout feature was sub-optimal and requires further research to better distribute the artifacts. The approach should preferably consider the stack definition itself to optimize placement based on artifact type rather than using a pre-existing generic graph node distribution algorithm.

- **Visual feedback.** This is a broad subject which considers minor changes such as more detailed status LEDs to fully match the notation used by Docker to more substantial improvements, for instance, optimizing feedback for Docker Swarm and its multiple containers per service.

- **Static validation.** While the prototype considers validations for some property fields, there is the potential to further enrich this feature with even more. These include, among others, validations for ports across the complete stack which take into account the available host ports.
- **Exploring liveness.** While the previous two points already contribute towards a more live experience, we can see the possibility of exploring this concept more exhaustively. Particularly by addressing more substantial concerns such as the need to manually restart stacks on change. We believe that increasing the liveness level can further improve the orchestration process.

Empirical validation

As stated in the conclusions of Chapter 7, although the findings obtained in the conducted user study are promising, further research is needed to consolidate and increase the confidence level of these results. For this purpose, we believe it is essential to perform similar experiments in other contexts to offset outstanding threats to validity identified. In particular, we suggest industrial settings to complement the data obtained in a purely academic environment. With this in mind and to encourage other researchers, we have compiled a replication package as described in Section 7.2.9.

While controlled user studies are powerful in identifying isolated cause-effect relations, they fail to fully capture the intricacies of real-world scenarios. To complement the research we believe that case studies may serve as another invaluable source of insight into the actual behavior of the approach in more realistic scenarios.

Appendix A

Tools Listing

This appendix contains a complete list of the tools identified in section 3.2 of the state of the art review.

Name	Source
cAdvisor	https://github.com/google/cadvisor
Scout	https://scoutapm.com/
Data Dog	https://www.datadoghq.com/
Prometheus	https://prometheus.io/
Sysdig	https://sysdig.com/
Sensu monitoring framework	https://sensu.io/
Sematext	https://sematext.com/
Dynatrace	https://www.dynatrace.com/solutions/application-monitoring/
Broadcom AIOps	https://www.broadcom.com/info/aiops/application-monitoring
Site24x7	https://www.site24x7.com/
AppDynamics	https://www.appdynamics.com/
Weavescope	https://github.com/weaveworks/scope

Table A.1: Monitoring tools source

Name	Source
Kitematic	https://kitematic.com/
Dockeron	https://github.com/dockeron/dockeron
Seagull	https://github.com/tobegit3hub/seagull
Portainer	https://www.portainer.io/
Docker Compose UI	https://github.com/francescou/docker-compose-ui
Swirl	https://github.com/cuigh/swirl
Swarmpit	https://github.com/swarmpit/swarmpit
DockStation	https://dockstation.io/
Admiral	https://github.com/vmware/admiral
Rancher	https://rancher.com/
Kubernetes	https://kubernetes.io/
Mesos	http://mesos.apache.org/
Nomad	https://www.nomadproject.io/

Table A.2: Service tools sources

Name	Source
CodeHerent	https://codeherent.tech/home
Visual Composer	https://cloudsoft.io/software/cfn-composer/
Cloudcraft	https://cloudcraft.co/
CloudMap	[64]

Table A.3: Infrastructure tools sources

Appendix B

Preliminary Work Questionnaire

This appendix contains the questionnaire used during the user-study performed as described in Chapter 4.

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.

*Obrigatório

Personal context

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

Marcar apenas uma oval por linha.

	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... ***

Marcar apenas uma oval por linha.

	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? ***

Marcar apenas uma oval.

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

4. **... worked on that had a docker-compose.yml file?** *
Marcar apenas uma oval.

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

5. **... used Dockerfiles created by others (colleagues or third parties)?** *
Marcar apenas uma 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)?** *
Marcar apenas uma oval.

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

7. **... created/updated a Dockerfile?** *
Marcar apenas uma oval.

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

8. **... created/updated a docker-compose.yml file?** *
Marcar apenas uma oval.

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

9. **In the Dockerfiles I've developed, I've specified...**
Marcar tudo o que for aplicável.

- ☐ ... 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...**
Marcar tudo o que for aplicável.

- ☐ ... volumes.
- ☐ ... networks.

Working with Docker technologies

11. **When I write a Dockerfile, I spend a lot of time... ***
Marcar apenas uma oval por linha.

	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?** *

Marcar apenas uma oval.

- Yes
- No

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

15. **When I write a docker-compose.yml file, I spend a lot of time…** *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
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.					

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

____

____

____

____

____

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

Marcar apenas uma oval.

◯ Yes

◯ No

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

____

____

____

____

____

19. **When I read a docker-compose.yml file, I spend a lot of time trying to understand …** *

Marcar apenas uma oval por linha.

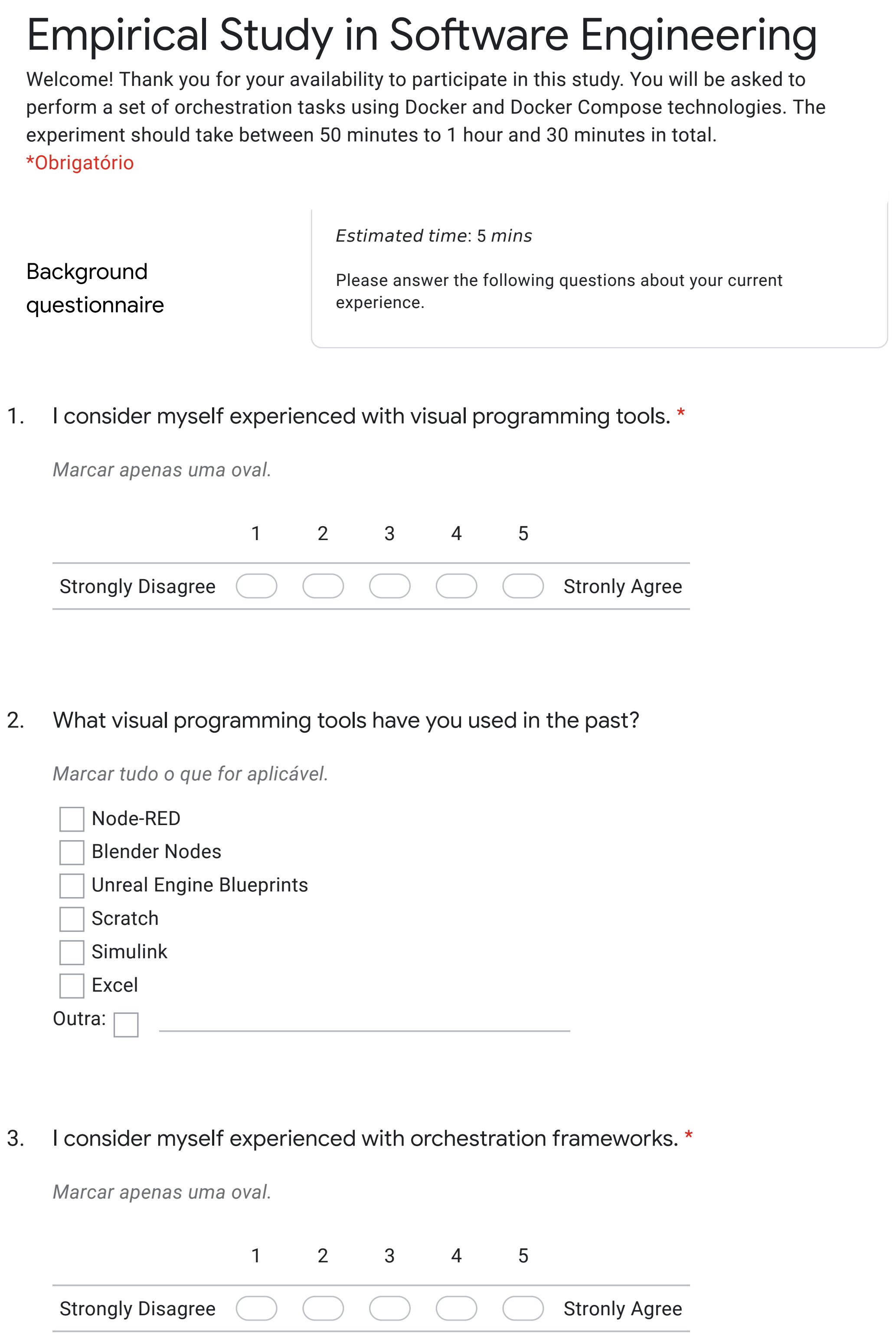
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
what the services are.	◯	◯	◯	◯	◯
the dependencies between services (e.g. depends_on).	◯	◯	◯	◯	◯
what volumes are used and how they are attached to the services.	◯	◯	◯	◯	◯
what networks are used and how they are connected to the services.	◯	◯	◯	◯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

Appendix C

User Study Materials

This appendix contains the materials used in the user study, namely, the Google Forms provided to the control and experimental groups.

C.1 Control



Empirical Study in Software Engineering

Welcome! Thank you for your availability to participate in this study. You will be asked to perform a set of orchestration tasks using Docker and Docker Compose technologies. The experiment should take between 50 minutes to 1 hour and 30 minutes in total.

*Obrigatório

Background questionnaire

Estimated time: 5 *mins*

Please answer the following questions about your current experience.

1. I consider myself experienced with visual programming tools. *

Marcar apenas uma oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Stronly Agree

2. What visual programming tools have you used in the past?

Marcar tudo o que for aplicável.

☐ Node-RED
☐ Blender Nodes
☐ Unreal Engine Blueprints
☐ Scratch
☐ Simulink
☐ Excel
Outra: ☐ ____________________

3. I consider myself experienced with orchestration frameworks. *

Marcar apenas uma oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Stronly Agree

4. What orchestration frameworks have you used in the past?

Marcar tudo o que for aplicável.

☐ Ansible
☐ Chef
☐ Puppet
☐ Salt
☐ Kubernetes
Outra: ☐ ______________________

5. I consider myself experienced with...

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Stronly Agree
...the Linux operating system.	⬭	⬭	⬭	⬭	⬭
...Docker.	⬭	⬭	⬭	⬭	⬭
...Docker Compose for development purposes.	⬭	⬭	⬭	⬭	⬭
...Docker Compose in production environments.	⬭	⬭	⬭	⬭	⬭

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

6. ...worked on which have used Docker Compose? *

Marcar apenas uma oval.

	0	1	2	3	4	5	6	7	8	9	10	
	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	>10

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

Marcar apenas uma oval.

	0	1	2	3	4	5	6	7	8	9	10	
	○	○	○	○	○	○	○	○	○	○	○	>10

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

Marcar apenas uma oval.

	0	1	2	3	4	5	6	7	8	9	10	
	○	○	○	○	○	○	○	○	○	○	○	>10

9. In the docker-compose files I've written, I've configured...

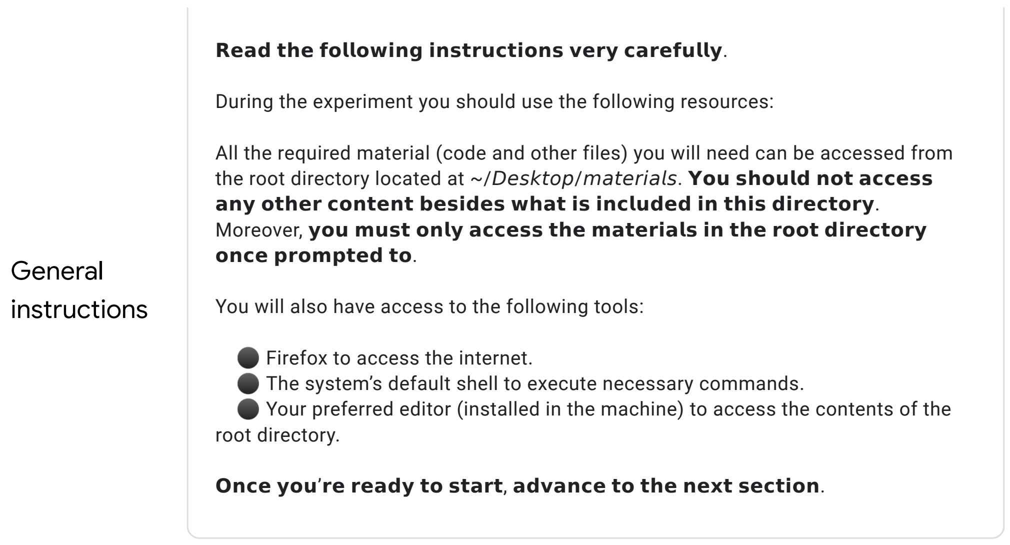
Marcar tudo o que for aplicável.

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

10. What software have you used to manage Docker or Docker Compose resources?

Marcar tudo o que for aplicável.

☐ Portainer
☐ Kitematic
☐ Admiral
☐ Dockstation
Outra: ☐ ____________________



General instructions

Read the following instructions very carefully.

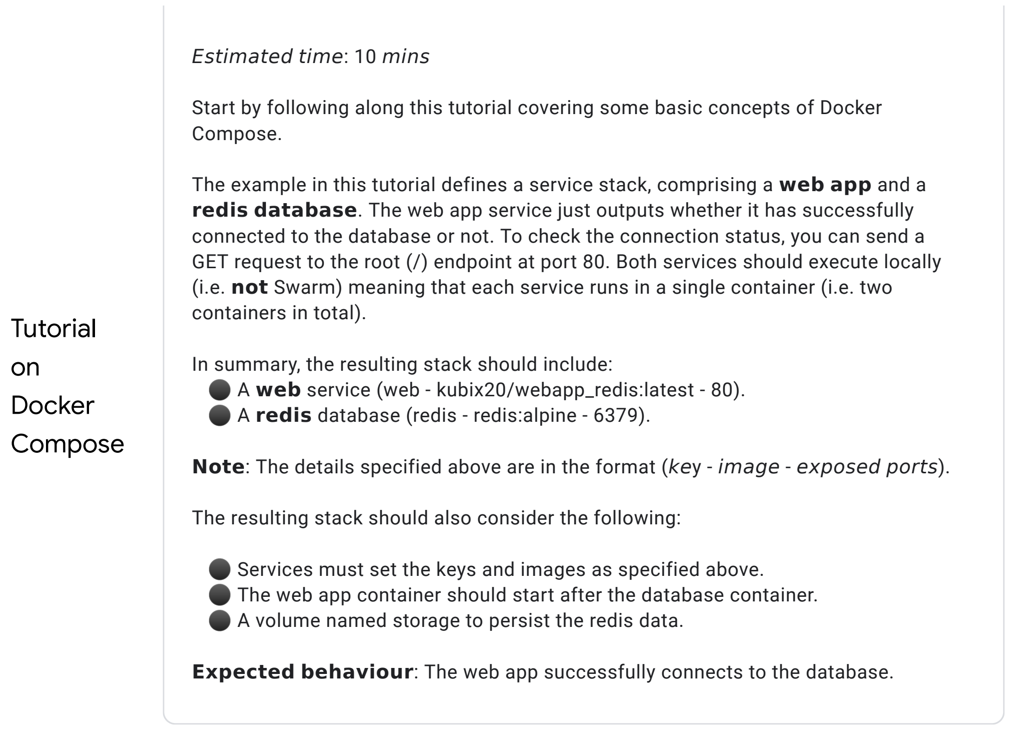
During the experiment you should use the following resources:

All the required material (code and other files) you will need can be accessed from the root directory located at ~/*Desktop*/*materials*. **You should not access any other content besides what is included in this directory**. Moreover, **you must only access the materials in the root directory once prompted to**.

You will also have access to the following tools:

- ⚫ Firefox to access the internet.
- ⚫ The system's default shell to execute necessary commands.
- ⚫ Your preferred editor (installed in the machine) to access the contents of the root directory.

Once you're ready to start, **advance to the next section**.



Tutorial on Docker Compose

Estimated time: 10 *mins*

Start by following along this tutorial covering some basic concepts of Docker Compose.

The example in this tutorial defines a service stack, comprising a **web app** and a **redis database**. The web app service just outputs whether it has successfully connected to the database or not. To check the connection status, you can send a GET request to the root (/) endpoint at port 80. Both services should execute locally (i.e. **not** Swarm) meaning that each service runs in a single container (i.e. two containers in total).

In summary, the resulting stack should include:
- ⚫ A **web** service (web - kubix20/webapp_redis:latest - 80).
- ⚫ A **redis** database (redis - redis:alpine - 6379).

Note: The details specified above are in the format (*key* - *image* - *exposed ports*).

The resulting stack should also consider the following:

- ⚫ Services must set the keys and images as specified above.
- ⚫ The web app container should start after the database container.
- ⚫ A volume named storage to persist the redis data.

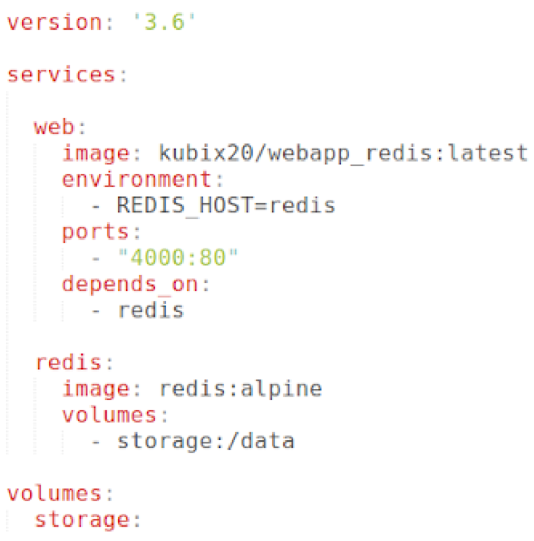
Expected behaviour: The web app successfully connects to the database.

11. Follow along these next steps. Tick each step as you complete it.

Marcar tudo o que for aplicável.

☐ 1. Create a file in the tutorial folder located in the root directory named *docker-compose.yml*.
☐ 2. Set the version to "3.6".
☐ 3. Add the top level “services” declaration.
☐ 4. Add a service with the key web.
☐ 5. Set the image to kubix20/webapp_redis:latest.
☐ 6. Open the image page on Docker Hub to learn more about the service.
☐ 7. Expose port 80 of the container to 4000 on the host.
☐ 8. Add a service with the key redis.
☐ 9. Set the image to redis:alpine.
☐ 10. Add a depends_on property from the redis to the web service.
☐ 11. Set the the environment variable REDIS_HOST = redis on the web service.
☐ 12. Add the top level "volumes" declaration.
☐ 13. Add a volume with the key storage.
☐ 14. Mount the storage volume on /data in the redis service.

The stack should now look like this:



```
version: '3.6'

services:

  web:
    image: kubix20/webapp_redis:latest
    environment:
      - REDIS_HOST=redis
    ports:
      - "4000:80"
    depends_on:
      - redis

  redis:
    image: redis:alpine
    volumes:
      - storage:/data

volumes:
  storage:
```

Next, it's time to test the stack.

12. Follow along these next steps. Tick each step as you complete it.

Marcar tudo o que for aplicável.

- ☐ 15. Ensure you've saved the file.
- ☐ 16. Run *docker-compose up* from the tutorial folder in the terminal.
- ☐ 17. Check the output in the terminal and verify that the message “Connected to DB” is printed by the web container, indicating that the web service has successfully connected to the database.
- ☐ 18. Make a GET request to localhost:4000 (using your preferred method, e.g. curl or browser) and verify the response “Connected to db” is received.
- ☐ 19. Ctrl+C in the terminal to stop the running stack.

If you've ticked all the boxes, advance to the next section.

Tasks

Estimated time: 30 *mins*

Read the following instructions very carefully.

You will be asked to perform a set of orchestration tasks with Docker Compose.

In the root directory, you will find folders containing the material for each task named t*, where * is the task number (e.g. t1 for task 1). **For each task**, **you must only access the contents of the corresponding folder while performing said task**. Moreover, **you must execute tasks** (**and subtasks**) **sequentially**, meaning that you cannot change your answers for previous tasks.

All tasks are preceded by a section labeled T* - Setup, where * is the task number (e.g. T1 - Setup), containing instructions you must follow before advancing to the actual task. These ensure that you abide by the guidelines described in this section.

When starting a task, carefully read the description once, in full. Once you’ve read the description and before you start solving the task, register the current time on the input labeled Start time. Likewise, once you finish the task register the current time on the input labeled Finish time. You can find both inputs below the description of the task.

Some tasks require the creation/edition of a Docker Compose stack. Keep in mind that the focus of the exercise is on the orchestration process and not in the development of the components that are part of the stack. In this sense, the requirements must be satisfied **strictly through the edition of the stack**.

Once you’re ready to start, **advance to the next section**.

T1 - Setup

Before proceeding, follow the next steps:

1. Navigate to the folder named **t1** from the root directory. You must exclusively access the contents of this folder while performing this task.
2. Open the stack in this folder named *docker-compose.yml*.

Once you’re ready to start, **advance to the next section**.

T1 - Analyzing a stack

In this task you will analyze a Docker Compose stack for a **voting app**.

Begin by carefully examining the contents of the stack. You can also start the app and access the services through their exposed ports to visualize it in action.

Important: You can learn more about each service in the corresponding image page on Docker Hub.

When you're ready, answer the following questions.

13. Start time *

Exemplo: 08:30

Exercise 1

14. Answer true or false to the following statements: *

Marcar apenas uma oval por linha.

	True	False
Some services use the default network.	◯	◯
The votes are stored in the redis service.	◯	◯
The named volume db-data is used to provide configurations to the postgres service at runtime.	◯	◯
The redis service always exposes port 6379 on the host.	◯	◯
The vote service uses a locally built image.	◯	◯

Exercise 2

15. What services depend on the redis service? (Answer in the format [services], e.g. ser1, ser2,...) *

16. What ports are exposed to the host by which services? (Answer in the format: service-[ports], e.g. container-123,124) *

17. What networks are used and what services are attached to each one? (Answer in the format: network-[services], e.g. net1-ser1, ser2,...; net2-ser1) *

18. Finish time *

Exemplo: 08:30

T2 - Setup

Before proceeding, follow the next steps:

1. Confirm that the stack is properly stopped.
2. Close all resources from previously used folders.
3. Navigate to the folder named **t2** from the root directory. You must exclusively access the contents of this folder while performing this task.
4. Open the stack in this folder named *docker-compose.yml*.

Once you're ready to start, **advance to the next section**.

T2 - Fixing a stack

Consider the stack for a simple app to register and view TODO notes.

The stack architecture is as follows:
- A mongoDB database to store the TODOs (mongo:4.2.0 - 27017)
- A backend server to expose an API to... (kubix20/todoapp_server - 3000)
- A client frontend for viewing and registering TODOs (kubix20/todoapp_client - 3000)

Note: The details specified above are in the format (*image - exposed ports*).

The backend server looks for the mongoDB service running on the host **mongo** and port 27017 without authentication required.

The frontend client proxies all API requests to the server service which is expected to be running on the host **server** and port 3000.

Unfortunately, the app isn't currently working as expected as users are unable to register new TODOs. Locate and fix the bugs so that the app behaves as expected.

Important: You must achieve the expected behaviour **without adding or removing any of the existing artifacts** (services, networks and volumes).

Note: The issue is unrelated to the stdin_open property on the client. It must be set to true for the client to execute properly.

Hint: Run the stack to observe the faulty behavior.

19. Start time *

____
Exemplo: 08:30

20. Finish time *

____
Exemplo: 08:30

T3.1 - Setup

Before proceeding, follow the next steps:

1. Confirm that the stack is properly stopped.
2. Close all resources from previously used folders.
3. Navigate to the directory named **t3.1** from the root directory. You must exclusively access the contents of this folder while performing this task.
4. Open the stack in this folder named *docker-compose.yml*.

Once you're ready to start, **advance to the next section**.

T3.1 - Creating a stack

Create a stack comprised of a **web app** and a **postgres database**. The web app outputs whether it has connected to the database or not. To check the connection status, you can send a GET request to the root endpoint (/) at port 80.

In summary, the stack should include:

- A **web app** service (web - kubix20/webapp_postgres:latest - 80)
- A **postgres database** (db - postgres:9.4 - 5432)

Note: The details specified above are in the format (*key* - *image* - *exposed ports*).

The web service accepts the following environment variables:

- DB_USER (default value ""), to specify the user when connecting to the postgres database.
- DB_PASSWORD (default value ""), to specify the password when connecting to the postgres database.

You must use the configuration files provided in the folder **t3.1**. More specifically:

- /postgres contains an environment file named credentials with variables to set the credentials in the **postgres database**. Note that you must set these variables by referencing the file and **not** by setting the individual environment variables within it.

The resulting stack should also consider the following:

- Services should set the keys and images as specified above.
- The web service should be exposed to the host on port 4000.
- The web container should start **after** the database container.
- A named volume called db-data to persist the database data mounted at /var/lib/postgresql/data.
- A custom network named my-net to which both services should be attached.

Expected behaviour: The web app successfully connects to the database.

The task is successfully complete **only if** the expected behavior is achieved **and** all other requirements satisfied.

21. Start time *

Exemplo: 08:30

22. Finish time *

Exemplo: 08:30

T3.2 - Setup

Before proceeding, follow the next steps:

1. Confirm that the stack is properly stopped.
2. Close all resources from previously used folders.
3. Navigate to the folder named **t3.2** from the root directory. You must exclusively access the contents of this folder while performing this task.
4. Open the stack in this folder named *docker-compose.yml*.

Once you're ready to start, **advance to the next section**.

T3.2 - With secrets

Alter the stack (as defined in T3.1) so that the database credentials are provided to the **web** service through secrets instead of environment variables.

You must use the configuration files provided in the folder **t3.2**. More specifically:

- /postgres/secrets contains two files (user and password) with matching credentials to the ones in the credentials file (used in the db service). The content of these files should be used as secrets, named db_user and db_password respectively.

Expected behaviour: The web app successfully connects to the database.

The task is successfully complete **only if** the expected behavior is achieved **and** all other requirements satisfied.

Note: The results service expects the credentials as secrets **or** environment variables. This means that the app will work as expected if the environment variables are correctly set, even if the secrets are not. Ensure that the environment variables are not set in the **web** service to test the stack.

Important: Ensure that you (re)start the stack with the --force-recreate option since docker-compose doesn't automatically detect changes to secret related properties to recreate the appropriate containers.

23. Start time *

Exemplo: 08:30

24. Finish time *

Exemplo: 08:30

Post-experiment questionnaire

Estimated time: 3 *mins*

You've completed all the tasks and have reached the last step of the experiment.

Please answer the following questions in regards to your experience.

25. Mark the answers that best reflect your opinions. *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
It was easy working in the remote machine.	⬭	⬭	⬭	⬭	⬭
The environment was distracting.	⬭	⬭	⬭	⬭	⬭
I found the procedure instructions complex and difficult to follow.	⬭	⬭	⬭	⬭	⬭
I found the task descriptions complex and difficult to follow.	⬭	⬭	⬭	⬭	⬭
Overall, I found the toolchain difficult to use.	⬭	⬭	⬭	⬭	⬭
I found it difficult to understand stacks with the toolchain.	⬭	⬭	⬭	⬭	⬭
I found it easy to define stacks with the toolchain.	⬭	⬭	⬭	⬭	⬭

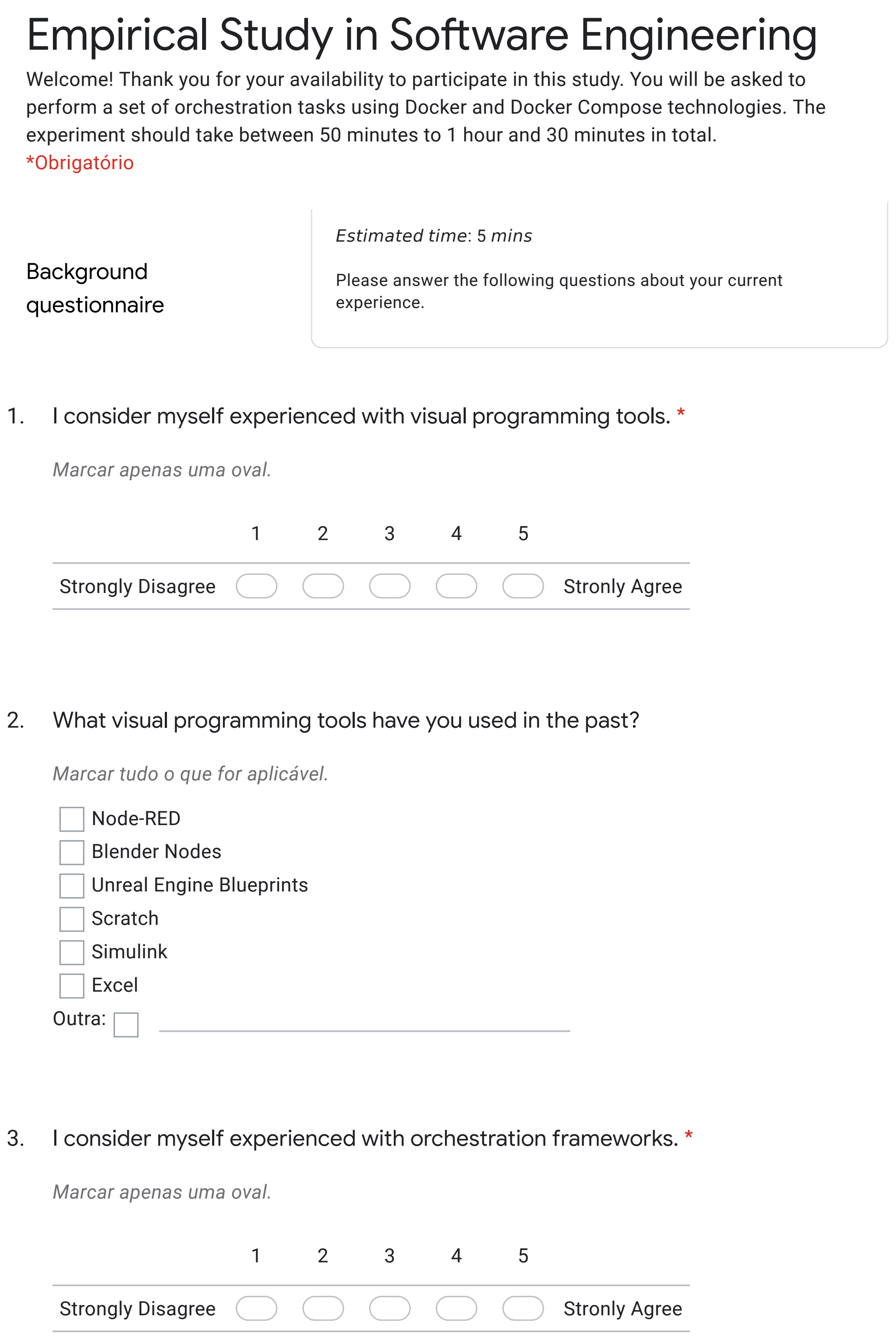
26. Any comments? (about your experience, the experiment process, ...)

End

Congratulations! You have completed the experiment.

Thank you for your participation!

C.2 Experimental



Empirical Study in Software Engineering

Welcome! Thank you for your availability to participate in this study. You will be asked to perform a set of orchestration tasks using Docker and Docker Compose technologies. The experiment should take between 50 minutes to 1 hour and 30 minutes in total.

*Obrigatório

Background questionnaire

Estimated time: 5 *mins*

Please answer the following questions about your current experience.

1. I consider myself experienced with visual programming tools. *

Marcar apenas uma oval.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Stronly Agree

2. What visual programming tools have you used in the past?

Marcar tudo o que for aplicável.

- ☐ Node-RED
- ☐ Blender Nodes
- ☐ Unreal Engine Blueprints
- ☐ Scratch
- ☐ Simulink
- ☐ Excel

Outra: ☐ ____________________

3. I consider myself experienced with orchestration frameworks. *

Marcar apenas uma oval.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Stronly Agree

4. What orchestration frameworks have you used in the past?

Marcar tudo o que for aplicável.

- ☐ Ansible
- ☐ Chef
- ☐ Puppet
- ☐ Salt
- ☐ Kubernetes

Outra: ☐ ______________________

5. I consider myself experienced with...

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Stronly Agree
...the Linux operating system.	○	○	○	○	○
...Docker.	○	○	○	○	○
...Docker Compose for development purposes.	○	○	○	○	○
...Docker Compose in production environments.	○	○	○	○	○

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

6. ...worked on which have used Docker Compose? *

Marcar apenas uma oval.

0	1	2	3	4	5	6	7	8	9	10	
○	○	○	○	○	○	○	○	○	○	○	>10

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

Marcar apenas uma oval.

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

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

Marcar apenas uma oval.

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

9. In the docker-compose files I've written, I've configured...

Marcar tudo o que for aplicável.

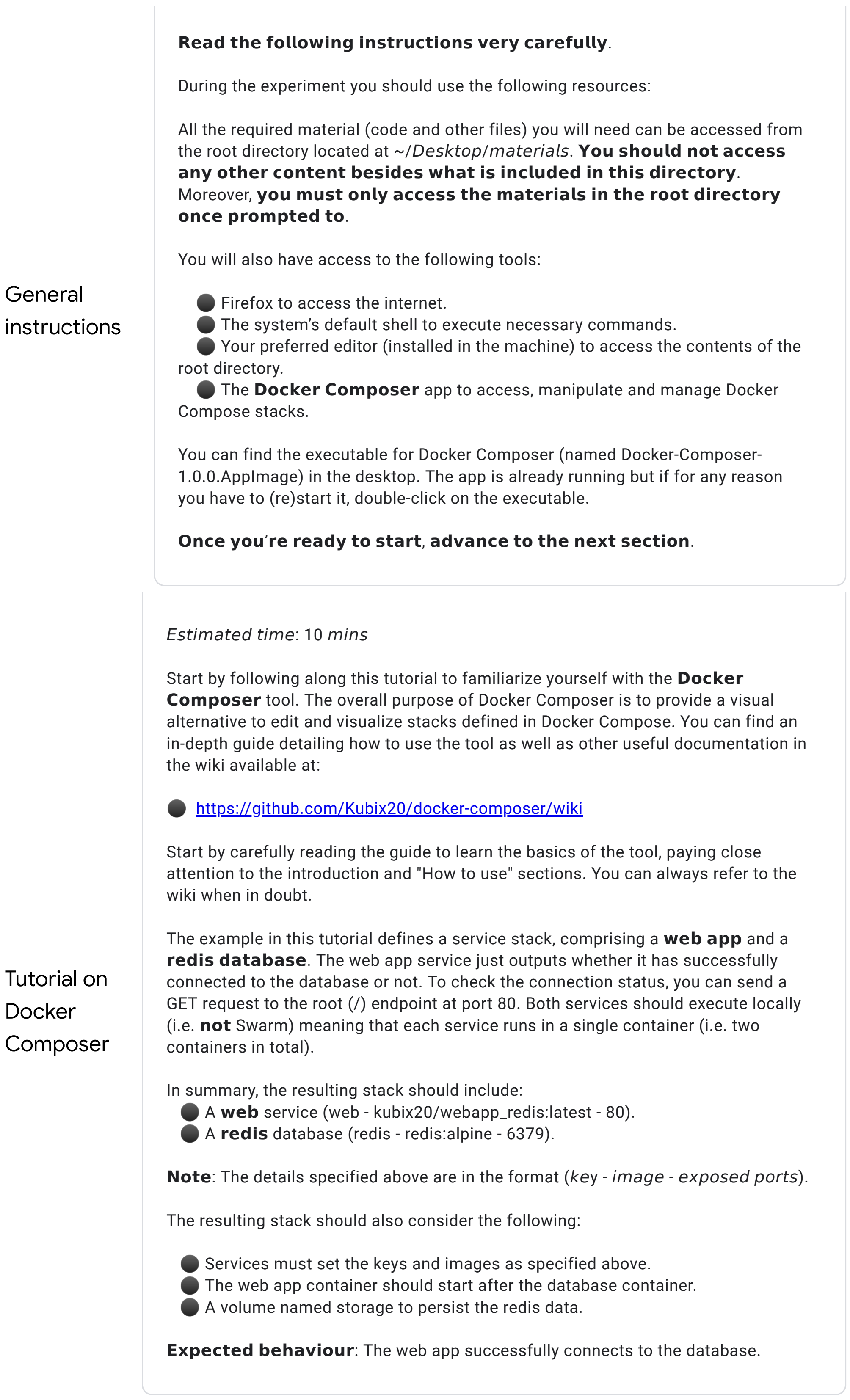
- ☐ ...volumes
- ☐ ...networks
- ☐ ...configs
- ☐ ...secrets

10. What software have you used to manage Docker or Docker Compose resources?

Marcar tudo o que for aplicável.

- ☐ Portainer
- ☐ Kitematic
- ☐ Admiral
- ☐ Dockstation

Outra: ☐ ____________________



General instructions

Read the following instructions very carefully.

During the experiment you should use the following resources:

All the required material (code and other files) you will need can be accessed from the root directory located at *~/Desktop/materials*. **You should not access any other content besides what is included in this directory**. Moreover, **you must only access the materials in the root directory once prompted to**.

You will also have access to the following tools:

- Firefox to access the internet.
- The system's default shell to execute necessary commands.
- Your preferred editor (installed in the machine) to access the contents of the root directory.
- The **Docker Composer** app to access, manipulate and manage Docker Compose stacks.

You can find the executable for Docker Composer (named Docker-Composer-1.0.0.AppImage) in the desktop. The app is already running but if for any reason you have to (re)start it, double-click on the executable.

Once you're ready to start, **advance to the next section**.

Tutorial on Docker Composer

Estimated time: 10 *mins*

Start by following along this tutorial to familiarize yourself with the **Docker Composer** tool. The overall purpose of Docker Composer is to provide a visual alternative to edit and visualize stacks defined in Docker Compose. You can find an in-depth guide detailing how to use the tool as well as other useful documentation in the wiki available at:

- https://github.com/Kubix20/docker-composer/wiki

Start by carefully reading the guide to learn the basics of the tool, paying close attention to the introduction and "How to use" sections. You can always refer to the wiki when in doubt.

The example in this tutorial defines a service stack, comprising a **web app** and a **redis database**. The web app service just outputs whether it has successfully connected to the database or not. To check the connection status, you can send a GET request to the root (/) endpoint at port 80. Both services should execute locally (i.e. **not** Swarm) meaning that each service runs in a single container (i.e. two containers in total).

In summary, the resulting stack should include:

- A **web** service (web - kubix20/webapp_redis:latest - 80).
- A **redis** database (redis - redis:alpine - 6379).

Note: The details specified above are in the format (*key* - *image* - *exposed ports*).

The resulting stack should also consider the following:

- Services must set the keys and images as specified above.
- The web app container should start after the database container.
- A volume named storage to persist the redis data.

Expected behaviour: The web app successfully connects to the database.

11. Follow along these next steps. Tick each step as you complete it.

Marcar tudo o que for aplicável.

☐ 1. Open the Docker Composer tool (already running on the machine).

☐ 2. Add a new service by right clicking in the graph area and selecting 'New service' in the context menu. This will be the **web** service.

☐ 3. Set the key field to web and image to kubix20/webapp_redis either in the artifact or in the properties editor.

☐ 4.Open the Docker Hub page for the image kubix20/webapp_redis by clicking on the icon above the image input on the properties editor to learn more about the service.

☐ 5. Add a port entry in the properties editor mapping port 80 on the container to 4000 on the host.

☐ 6. Using the image palette search for the official redis image and drag it onto the graph area. This will be the **redis** service.

☐ 7. Set the key to redis and image tag to alpine.

☐ 8. Add a depends_on dependency from the web service to the redis service by dragging an arrow from the depends_on anchor point on the web service.

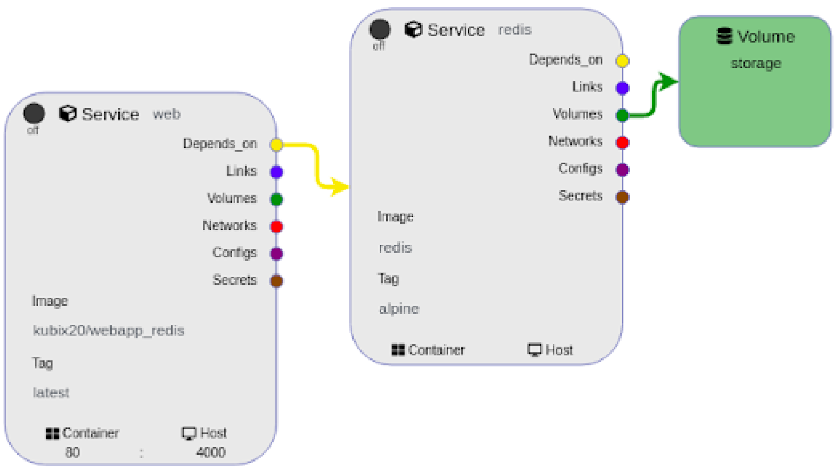
☐ 9. Add the environment variable in the properties editor REDIS_HOST=redis in the web service.

☐ 10. Add a volume via the context menu and set the key to storage.

☐ 11. Connect the redis service to the volume through the corresponding volume anchor point (green).

☐ 12. Select the connection and set the target to /data in properties editor. This is the path where the volume will be mounted in container.

The stack should now look like this:



Next, it's time to test the stack.

12. Follow along these next steps. Tick each step as you complete it.

Marcar tudo o que for aplicável.

- ☐ 13. Click on the Start button in the toolbar to run the app.
- ☐ 14. Check the output in the terminal and verify that the message "Connected to DB" is printed by the web container, indicating that the web service has successfully connected to the database.
- ☐ 15. Make a GET request to localhost:4000 (using your preferred method, e.g. curl or browser) and verify the response "Connected to db" is received.
- ☐ 16. Click on the Stop button to stop the running stack.
- ☐ 17. Export the stack and save it in the tutorial folder located in the root directory with the name *tutorial.yml*.

If you've ticked all the boxes, advance to the next section.

Tasks

Estimated time: 30 *mins*

Read the following instructions very carefully.

You will be asked to perform a set of orchestration tasks with Docker Compose.

In the root directory, you will find folders containing the material for each task named t*, where * is the task number (e.g. t1 for task 1). **For each task**, **you must only access the contents of the corresponding folder while performing said task**. Moreover, **you must execute tasks** (**and subtasks**) **sequentially**, meaning that you cannot change your answers for previous tasks.

All tasks are preceded by a section labeled T* - Setup, where * is the task number (e.g. T1 - Setup), containing instructions you must follow before advancing to the actual task. These ensure that you abide by the guidelines described in this section.

When starting a task, carefully read the description once, in full. Once you've read the description and before you start solving the task, register the current time on the input labeled Start time. Likewise, once you finish the task register the current time on the input labeled Finish time. You can find both inputs below the description of the task.

Some tasks require the creation/edition of a Docker Compose stack. Keep in mind that the focus of the exercise is on the orchestration process and not in the development of the components that are part of the stack. In this sense, the requirements must be satisfied **strictly through the edition of the stack**.

Once you're ready to start, **advance to the next section**.

T1 - Setup

Before proceeding, follow the next steps:

1. Navigate to the folder named **t1** from the root directory. You must exclusively access the contents of this folder while performing this task.
2. Load the stack in this folder with Docker Composer by clicking on the 'Open' button and selecting the folder t1.

Once you're ready to start, **advance to the next section**.

T1 - Analyzing a stack

In this task you will analyze a Docker Compose stack for a **voting app**.

Begin by carefully examining the contents of the stack. You can also start the app and access the services through their exposed ports to visualize it in action.

Important: You can learn more about each service in the corresponding image page on Docker Hub.

When you're ready, answer the following questions.

13. Start time *

Exemplo: 08:30

Exercise 1

14. Answer true or false to the following statements: *

Marcar apenas uma oval por linha.

	True	False
Some services use the default network.	⬭	⬭
The votes are stored in the redis service.	⬭	⬭
The named volume db-data is used to provide configurations to the postgres service at runtime.	⬭	⬭
The redis service always exposes port 6379 on the host.	⬭	⬭
The vote service uses a locally built image.	⬭	⬭

Exercise 2

15. What services depend on the redis service? (Answer in the format [services], e.g. ser1, ser2,...) *

16. What ports are exposed to the host by which services? (Answer in the format: service-[ports], e.g. container-123,124) *

17. What networks are used and what services are attached to each one? (Answer in the format: network-[services], e.g. net1-ser1, ser2,...; net2-ser1) *

18. Finish time *

Exemplo: 08:30

T2 - Setup

Before proceeding, follow the next steps:

1. Confirm that the stack is properly stopped.
2. Close all resources from previously used folders.
3. Navigate to the folder named **t2** from the root directory. You must exclusively access the contents of this folder while performing this task.
4. Load the stack in this folder with Docker Composer.

Once you're ready to start, advance to the next section.

T2 - Fixing a stack

Consider the stack for a simple app to register and view TODO notes.

The stack architecture is as follows:

- A mongoDB database to store the TODOs (mongo:4.2.0 - 27017)
- A backend server to expose an API to… (kubix20/todoapp_server - 3000)
- A client frontend for viewing and registering TODOs (kubix20/todoapp_client - 3000)

Note: The details specified above are in the format (*image - exposed ports*).

The backend server looks for the mongoDB service running on the host **mongo** and port 27017 without authentication required.

The frontend client proxies all API requests to the server service which is expected to be running on the host **server** and port 3000.

Unfortunately, the app isn't currently working as expected as users are unable to register new TODOs. Locate and fix the bugs so that the app behaves as expected.

Important: You must achieve the expected behaviour **without adding or removing any of the existing artifacts** (services, networks and volumes).

Note: The issue is unrelated to the stdin_open property on the client. It must be set to true for the client to execute properly.

Hint: Run the stack to observe the faulty behavior.

Important: Once you have finished the task, export the stack and save it in the t2 folder with the name docker-compose.yml.

19. Start time *

Exemplo: 08:30

20. Finish time *

Exemplo: 08:30

T3.1 - Setup

Before proceeding, follow the next steps:

1. Confirm that the stack is properly stopped.
2. Close all resources from previously used folders.
3. Navigate to the directory named **t3.1** from the root directory. You must exclusively access the contents of this folder while performing this task.
4. Load the stack in this folder with Docker Composer. Once loaded, the graph area should be empty.

Once you're ready to start, **advance to the next section**.

T3.1 - Creating a stack

Create a stack comprised of a **web app** and a **postgres database**. The web app outputs whether it has connected to the database or not. To check the connection status, you can send a GET request to the root endpoint (/) at port 80.

In summary, the stack should include:

- A **web app** service (web - kubix20/webapp_postgres:latest - 80)
- A **postgres database** (db - postgres:9.4 - 5432)

Note: The details specified above are in the format (*key* - *image* - *exposed ports*).

The web service accepts the following environment variables:

- DB_USER (default value ""), to specify the user when connecting to the postgres database.
- DB_PASSWORD (default value ""), to specify the password when connecting to the postgres database.

You must use the configuration files provided in the folder **t3.1**. More specifically:

- /postgres contains an environment file named credentials with variables to set the credentials in the **postgres database**. Note that you must set these variables by referencing the file and **not** by setting the individual environment variables within it.

The resulting stack should also consider the following:

- Services should set the keys and images as specified above.
- The web service should be exposed to the host on port 4000.
- The web container should start **after** the database container.
- A named volume called db-data to persist the database data mounted at /var/lib/postgresql/data.
- A custom network named my-net to which both services should be attached.

Expected behaviour: The web app successfully connects to the database.

The task is successfully complete **only if** the expected behavior is achieved **and** all other requirements satisfied.

Important: Once you have finished the task, export the stack and save it in the t3.1 folder with the name docker-compose.yml.

21. Start time *

Exemplo: 08:30

22. Finish time *

Exemplo: 08:30

T3.2 - Setup	Before proceeding, follow the next steps: 1. Confirm that the stack is properly stopped. 2. Close all resources from previously used folders. 3. Navigate to the folder named **t3.2** from the root directory. You must exclusively access the contents of this folder while performing this task. 4. Load the stack in this folder with Docker Composer. **Once you're ready to start**, **advance to the next section**.
T3.2 - With secrets	Alter the stack (as defined in T3.1) so that the database credentials are provided to the **web** service through secrets instead of environment variables. You must use the configuration files provided in the folder **t3.2**. More specifically: ● /postgres/secrets contains two files (user and password) with matching credentials to the ones in the credentials file (used in the db service). The content of these files should be used as secrets, named db_user and db_password respectively. **Expected behaviour**: The web app successfully connects to the database. The task is successfully complete **only if** the expected behavior is achieved **and** all other requirements satisfied. **Note**: The results service expects the credentials as secrets **or** environment variables. This means that the app will work as expected if the environment variables are correctly set, even if the secrets are not. Ensure that the environment variables are not set in the **web** service to test the stack. **Important**: Once you have finished the task, export the stack and save it in the t3.2 folder with the name docker-compose.yml.

23. Start time *

____

Exemplo: 08:30

24. Finish time *

____

Exemplo: 08:30

Post-experiment questionnaire	*Estimated time*: 8 *mins* You've completed all the tasks and have reached the last step of the experiment. Please answer the following questions in regards to your experience.

25. Mark the answers that best reflect your opinions. *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
It was easy working in the remote machine.	⬭	⬭	⬭	⬭	⬭
The environment was distracting.	⬭	⬭	⬭	⬭	⬭
I found the procedure instructions complex and difficult to follow.	⬭	⬭	⬭	⬭	⬭
I found the task descriptions complex and difficult to follow.	⬭	⬭	⬭	⬭	⬭
Overall, I found the tool easy to learn.	⬭	⬭	⬭	⬭	⬭
Overall, I found the tool difficult to use.	⬭	⬭	⬭	⬭	⬭
I found it difficult to understand stacks with the tool.	⬭	⬭	⬭	⬭	⬭
I found it easy to define stacks with the tool.	⬭	⬭	⬭	⬭	⬭

Mark the answers that best reflect your opinions.

26. I find the visual map of artifacts.... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...helpful to understand stacks with less effort.	⬭	⬭	⬭	⬭	⬭
...helpful to understand stacks more quickly.	⬭	⬭	⬭	⬭	⬭
...helpful to define stacks with less effort.	⬭	⬭	⬭	⬭	⬭
...helpful to define stacks more quickly.	⬭	⬭	⬭	⬭	⬭

27. I find the integration with Docker Hub (image palette and links to the docs)... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...helpful to understand stacks with less effort.	⬭	⬭	⬭	⬭	⬭
...helpful to understand stacks more quickly.	⬭	⬭	⬭	⬭	⬭
...helpful to define stacks with less effort.	⬭	⬭	⬭	⬭	⬭
...helpful to define stacks more quickly.	⬭	⬭	⬭	⬭	⬭

28. I find the visual feedback of running stacks (service and stack LEDs)... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...helpful to understand the state of a stack with less effort.	◯	◯	◯	◯	◯
...helpful to understand the state of a stacks more quickly.	◯	◯	◯	◯	◯

29. I find executing commands in the UI (start and stop)... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...helpful to define stacks with less effort.	◯	◯	◯	◯	◯
...helpful to define stacks more quickly.	◯	◯	◯	◯	◯

30. In comparison to the conventional procedure (editing a docker-compose.yml file and docker cli)... *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
...I believe this tool would reduce the effort required to define Docker Compose stacks.	◯	◯	◯	◯	◯
...overall, I found the tool useful.	◯	◯	◯	◯	◯
...a Docker Compose stack visualized with the tool would be more difficult to understand.	◯	◯	◯	◯	◯
...overall, I think this tool does not provide an effective solution to define Docker Compose stacks.	◯	◯	◯	◯	◯
...overall, I think this tool makes an improvement to the stack definition process.	◯	◯	◯	◯	◯

31. Mark the answers that best reflect your opinions *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
This tool would make it easier for practitioners to define Docker Compose stacks.	⬭	⬭	⬭	⬭	⬭
Using this tool would make it easier to communicate the stack architecture to other practitioners.	⬭	⬭	⬭	⬭	⬭
I would recommend this tool to work with Docker Compose.	⬭	⬭	⬭	⬭	⬭
I would like to use this tool in the future.	⬭	⬭	⬭	⬭	⬭
It would be easy for me to become skillful in using this tool to work with Docker Compose.	⬭	⬭	⬭	⬭	⬭

32. What do you think can be improved upon in the tool?

33. Any comments? (about your experience, the experiment process, the tool itself, ...)

End

Congratulations! You have completed the experiment.

Thank you for your participation!

References

[1] Babak Abbasov. Cloud computing: State of the art reseach issues. *8th IEEE International Conference on Application of Information and Communication Technologies, AICT 2014 - Conference Proceedings*, 2014.

[2] Ademar Aguiar, André Restivo, Filipe F. Correia, Hugo Sereno Ferreira, and João Pedro Dias. Software Development. *Programming '19: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, pages 1–6, 2019.

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016*, pages 44–51, 2016.

[4] Diogo Amaral, Gil Domingues, João Pedro Dias, Hugo Sereno Ferreira, Ademar Aguiar, Rui Nóbrega, and Filipe Figueiredo Correia. Live software development environment using virtual reality: A prototype and experiment. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 83–107, Cham, 2020. Springer International Publishing.

[5] László Angyal, László Lengyel, and Hassan Charaf. A synchronizing technique for syntactic model-code round-trip engineering. *Proceedings - Fifteenth IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2008*, pages 463–472, 2008.

[6] Danilo Ardagna, Elisabetta Di Nitto, Parastoo Mohagheghi, Sébastien Mosser, Cyril Ballagny, Francesco D'Andria, Giuliano Casale, Peter Matthews, Cosmin Septimiu Nechifor, Dana Petcu, Anke Gericke, and Craig Sheridan. MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds. *2012 4th International Workshop on Modeling in Software Engineering, MiSE 2012 - Proceedings*, pages 50–56, 2012.

[7] Michael Armbrust, Anthony D Joseph, Randy H Katz, and David A Patterson. Above the Clouds: A Berkeley View of Cloud Computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):25, 2009.

[8] V. L. Averbukh. Visualization metaphors. *Programming and Computer Software*, 27(5):227–237, 2001.

[9] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based cloud application modeling with libraries, profiles, and templates. *CEUR Workshop Proceedings*, 1242(317859):56–65, 2014.

[10] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1:81–84, 09 2014.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*, volume 10. Addison-Wesley Professional, 05 2005.

[12] Marat Boshernitsan and Michael Downes. Visual Programming Languages: A Survey. *Computer Science Division (EECS)*, 2004.

[13] M Brambilla, Jordi Cabot, and M Wimmer. *Model-driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[14] Margaret Burnett. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 32(1-3):275–283, 1999.

[15] Margaret Burnett. Software Engineering for Visual Programming Languages. *Handbook of Software Engineering & Knowledge Engineering, Volume 2*, 2:77–92, 2002.

[16] Margaret M. Burnett and Marla J. Baker. A Classification System for Visual Programming Languages. *Journal of Visual Languages & Computing*, 5(3):287–300, 1994.

[17] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes Up & Running*. O'Reilly Media, Inc., 2019.

[18] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[19] Emiliano Casalicchio. Container Orchestration: A Survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[20] Shi Kuo Chang. Visual Languages: A Tutorial and Survey. *IEEE Software*, 4(1):29–39, 1987.

[21] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5563 LNCS:260–283, 2009.

[22] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 45(3):1–17, 2003.

[23] J. P. Dias, J. P. Faria, and H. S. Ferreira. A reactive and model-based approach for developing internet-of-things systems. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 276–281, 2018.

[24] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag Berlin Heidelberg, 2007.

[25] M H Ebell. Visual programming languages. *M.D. computing : computers in medical practice*, 10(5):305–311, 1993.

[26] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, 2016.

[27] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. CloudMF: Model-driven management of multi-cloud applications. *ACM Transactions on Internet Technology*, 18(2):1–24, 2018.

[28] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2505:90–105, 2002.

[29] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, pages 480–483, 2011.

[30] Soichiro Hidaka, Kazuhiro Inaba, Zhenjiang Hu, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. *ACM SIGPLAN Notices*, 45(9):205–216, 2010.

[31] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992.

[32] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[33] Bernhard Hoisl, Zhenjiang Hu, and Soichiro Hidaka. Towards Bidirectional Higher-Order Transformation for Model-Driven Co-evolution. *Communications in Computer and Information Science*, page 15, 2015.

[34] Jezz Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[35] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 11 2000.

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

[37] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[38] Ann Mary Joy. Performance comparison between Linux containers and virtual machines. *Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015*, pages 342–346, 2015.

[39] Nafiseh Kahani and James R Cordy. Comparison and Evaluation of Model Transformation Tools. *Software and Systems Modeling*, 24(3):1–42, 2015.

[40] Babu Kavitha and Perumal Varalakshmi. Performance Analysis of Virtual Machines and Docker Containers. *Communications in Computer and Information Science*, 804:99–113, 2018.

[41] Asif Khan. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing*, 4(5):42–48, 2017.

[42] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained; The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[43] Pedro Lourenço, João Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Cloudcity: A live environment for the management of cloud infrastructures. *ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 27–36, 2019.

[44] Pedro Lourenço, João Pedro Dias, Ademar Aguiar, Hugo Sereno Ferreira, and André Restivo. Experimenting with liveness in cloud infrastructure management. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 58–82, Cham, 2020. Springer International Publishing.

[45] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010.

[46] Russ McKendrick and Scott Gallagher. *Mastering Docker - Second Edition.* Packt Publishing, 2017.

[47] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, 2006.

[48] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[49] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, Scotts Valley, CA, 2010.

[50] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[51] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 7161(c):1–14, 2017.

[52] Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-driven management of docker containers. *IEEE International Conference on Cloud Computing, CLOUD*, pages 718–725, 2017.

[53] Akond Rahman, North Carolina, Chris Parnin, North Carolina, Laurie Williams, and North Carolina. Gang of Eight : A Defect Taxonomy for Infrastructure as Code Scripts. Accepted submission for the International Conference on Software Engineering (ICSE) 2020.

[54] Akond Rahman, Rezvan Mahdavi-hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108, 12 2018.

[55] Alessandro Rossini. Cloud application modelling and execution language (CAMEL) and the PaaSage workflow. *Communications in Computer and Information Science*, 567:437–439, 2016.

[56] Julio Sandobalin, Emilio Insfran, and Silvia Abrahao. ARGON: A Tool for Modeling Cloud Resources. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10797 LNCS(November):393–397, 2018.

[57] Julio Sandobalin, Emilio Insfran, and Silvia Abrahao. On the Effectiveness of Tools to Support Infrastructure as Code : Model-Driven versus Code-Centric. *IEEE Access*, 8, 2020.

[58] Nan C. Shu. Visual Programming Languages: A Perspective and a Dimensional Analysis. *Visual Languages*, pages 11–34, 1986.

[59] 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.

[60] Joel Spolsky. The Law of Leaky Abstractions. *Joel on Software*, pages 197–202, 2004.

[61] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4735 LNCS:1–15, 2007.

[62] Steven Tanimoto. A perspective on the evolution of live programming. *International Conference on Software Engineering*, 41(10):31–34, 2013.

[63] James Turnbull. *The Docker Book*. James Turnbull, 2016.

[64] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, and Cao Jian. CloudMap: A Visual Notation for Representing and Managing Cloud Resources. *28th International Conference, CAiSE 2016*, pages 427–443, 2016.

[65] Marvin Zelkowitz and Dolores Wallace. Experimental models for validating technology. *Computer*, 31:23 – 31, 06 1998.