

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Service Containerization with Docker

João Carlos Maduro



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Filipe Correia

Co-Supervisor: João Bispo

July 26, 2021

Automatic Service Containerization with Docker

João Carlos Maduro

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Pedro Souto

External Examiner: Prof. Nour Ali

Supervisor: Prof. Filipe Correia

July 26, 2021

Abstract

In recent years, many systems have been developed for the cloud due to all of its advantages like availability and flexibility, but it also carries new challenges, for example, those related to deploying, orchestrating, or isolating services from the underlying infrastructure. With containers provided by Docker, it is possible to remove the barriers to deploy in the cloud and adopt a service-oriented architecture.

To define the configuration of a Docker container, developers need to write a container specification—a *dockerfile*. To do so, the developer needs a good understanding of how the system is operated, but the dockerfiles are created often with a trial-error approach, which is not optimal. Furthermore, research shows that containers often include obsolete dependencies and vulnerabilities.

The objective of this master thesis is to propose an approach for automatically generating dockerfiles, implement it as a tool, and study how it performs with real-world projects. The prototype, which we name Hermit, dynamically tracks the system calls executed by the software and statically inspects its source code to infer the dependencies, the entrypoint of the application and the ports that must be exposed. Hermit’s approach is based on other works that can infer the each one of the different fields of a dockerfile.

Our study uses Hermit to generate dockerfiles for multiple open-source projects, and evaluates them using the original dockerfiles of the projects as baselines. This study aims to understand if the proposed approach can infer the dependencies without adding bloat, infer the exposed network ports without adding risks, and infer a suitable entrypoint for the service.

The results of our study showed that the containers, built with the dockerfiles generated by Hermit, tend to miss in average 22.3% of the dependencies while adding an average bloat of 43.2%, with an average balance of 5.5 MB. The field where we got more success was the detection of ports, due to its high accuracy (100%) and low risk (0%) of exposing unnecessary ports. Finally, the detected entrypoints got a similarity with their original counterparts of 72.6% .

Keywords: Cloud, Containers, Docker, Isolation, Automation, Dependencies, Bloat, Risk

Resumo

Nos últimos anos, vários sistemas têm sido desenvolvidos para a cloud, devido às vantagens que a mesma oferece como a disponibilidade e a flexibilidade, mas ao mesmo tempo leva a novos desafios, como por exemplo na questão do deploy, da orquestração e do isolamento de serviços da infraestrutura inferior. Com contentores, gerados pelo Docker, é possível remover as barreiras que impedem o desenvolvimento para a cloud e a integração desse software numa arquitectura de serviços.

Para definir a configuração de contentor de Docker, os desenvolvedores precisam de escrever manualmente um dockerfile. Para escrever um, os desenvolvedores necessitam de uma compreensão profunda do sistema, mas os contentores são criados com abordagem tentativa erro, o que por sua vez não é ótimo. Além disso, a falta de tempo e experiência podem resultar em contentores com dependências obsoletas e vulnerabilidades.

O objectivo desta tese de mestrado consiste em estudar e implementar uma ferramenta que gere dockerfiles adequados, sem comprometer a execução dos serviços. O protótipo, que nós chamamos de hermit, irá recolher dinamicamente as chamadas ao sistema efectuadas pelo software e inspecionar estaticamente o código-fonte do mesmo software, para inferir as dependências, o ponto de entrada da aplicação e as portas de rede que devem ser expostas. A abordagem do hermit é inspirado em diferentes trabalhos existentes, que inferem os diferentes campos de um dockerfile individualmente.

Um estudo empírico foi conduzido para avaliar a performance do hermit a gerar dockerfiles para múltiplos projects open-source, usando os dockerfiles originais como referência. Este estudo ambiciona compreender se a abordagem proposta consegue inferir as dependências sem adicionar bloat (dependências desnecessárias), inferir as portas de rede expostas sem adicionar riscos e infer um ponto de entrada adequado ao serviço.

Os resultados deste estudo mostraram que os contentores, construídos com dockerfiles gerados pelo Hermit, tendem a perder em média 22.3% das dependências, enquanto adicionam uma média de 43.2% de dependências desnecessárias, com um balanço médio de 5.5 MB. O campo onde tivemos mais sucesso foi na deteção de portas, devido à alta precisão (100%) e ao baixo risco (0%) de exposição de portas desnecessárias. Finalmente, os pontos de entrada detetados tiveram uma semelhança com os seus homólogos originais de 72.6%.

Keywords: Cloud, Contentores, Docker, Isolamento, Automação, Dependências, Bloat, Risco

Acknowledgements

In the first place, I would like my supervisors Filipe Correia and João Bispo, for all the guidance they gave me through the development of this master thesis. Without their help, I would not have overcome the different challenges faced during this last phase of my academic journey. I also would like to thank them for always being available to reunite with me, even with all the parallel responsibilities they had.

During these five years in *Faculdade de Engenharia da Universidade do Porto*, I met many people and made many friends, whom I would like to thank for all the good moments, memorable adventures, and things you taught me. It was because of you that I felt that college was my second home.

Finally, my gratitude goes to my family, not only due to the encouragement to fulfill my studies but also due to all the sacrifices they made to one day have a master's degree. And this was all possible thanks to them.

João Carlos Maduro

*“Software is a gas;
Software always expands to fit whatever container it is stored in.”*

Nathan Myhrvold

Contents

1	Introduction	1
1.1	Context	1
1.2	Objectives	2
1.3	Contributions	2
1.4	Document Structure	2
2	Background	5
2.1	Virtualization Technologies	5
2.1.1	Virtual Machines	5
2.1.2	Containers	6
2.2	Docker	6
2.2.1	Container Images	6
2.2.2	Services Architecture	7
3	State of the Art	9
3.1	Introduction	9
3.2	Goals and Methodology	9
3.3	Minimalist Containers	11
3.4	Dependency Inference	14
3.5	Ports Detection	16
3.6	Entrypoint detection	16
3.7	Empirical study about dockerfiles	17
3.8	Conclusions	20
4	Problem Statement and Research Strategy	23
4.1	Scope Definition	23
4.2	Hypothesis	24
4.3	Research Questions	25
4.4	Methodology	25
5	Approach and Design of Hermit	27
5.1	Approach	27
5.1.1	Project Analysis	28
5.1.2	Data Inference	29
5.1.3	Dockerfile Generation	32
5.2	Prototype - Hermit	34
5.2.1	Architecture	34
5.2.2	Design Decisions	35

5.2.3	Limitations	36
5.3	Hermit's Features	37
5.3.1	Dockerfile Generation	37
5.3.2	Container Execution	38
5.3.3	Two-Stage Dockerfile	39
6	Empirical Study	41
6.1	Study Goals	41
6.2	Design	41
6.2.1	Study Variables	42
6.2.2	Project Sampling	42
6.2.3	Dockerfile Comparison	43
6.2.4	Data Analysis	44
6.2.5	Replication Package	45
6.3	A toolset for running the study	45
6.3.1	Projects-fetcher	45
6.3.2	dockerfile-diff	47
6.4	Results analysis and discussion	48
6.4.1	Projects acceptance	48
6.4.2	Dockerfile Validation	49
6.4.3	Dockerfile Evaluation	49
6.5	Answering the Research Questions	56
6.5.1	RQ1 - To what extent it is possible to detect all dependencies without adding bloat?	57
6.5.2	RQ2 - To what extent it is possible to detect the ports to expose without adding unnecessary risks to the service?	57
6.5.3	RQ3 - To what extent it is possible to infer a suitable entrypoint?	57
6.6	Comparing against the state of the art	57
6.7	Validation's threats	58
7	Conclusions	61
7.1	Main Contributions	61
7.2	Future Work	62
7.2.1	Prototype Upgrades	63
7.2.2	Empirical Study Expansions	63
A	Empirical Study	65
A.1	Study Replication Steps	65
	References	67

List of Figures

2.1	Virtual Machine Architecture	5
2.2	Containers Architecture	6
3.1	Basic dockerfile for a NGINX Server and its 4 goals	10
3.2	Cimplifier [17] results	12
3.3	Confine's pipeline to select the authorized system calls [7]	12
3.4	Flowchart representing the dynamic analysis to build a container [28]	13
3.5	Comparison with a Minimal Container [8]	13
3.6	Example dependency tree	14
3.7	Pipeline of static analysis, used in "Automatic Container Definition" [13]	15
3.8	Dependency graph [11]	15
3.9	Distribution of revision sizes measured in lines changed [4]	17
3.10	Distribution of top 15 languages [4]	18
3.11	Percentage of Projects with Base Image Referenced in FROM Statements [4]	19
3.12	Percentage of base image types across top 25 images [4]	19
4.1	Complete containerization's hypothetical pipeline	24
5.1	Approach's activity diagram	27
5.2	Hermit's architecture diagram	34
5.3	Hermit doing the project analysis	37
5.4	Hermit Inferring the system packages	38
5.5	Hermit finishing and presenting all the dockerfile data	38
5.6	Example of a container execution	39
6.1	Study's activity diagram and artifacts	42
6.2	projects-fetcher activity digaram	46
6.3	dockerfile-diff activity digaram	47
6.4	Results from all of the selected projects	48
6.5	Results from all of the selected projects	49
6.6	Bloat Ratio boxplot	51
6.7	Missed Dependencies Ratio boxplot	52
6.8	Balance boxplot	53
6.9	Ports Detection Accuracy boxplot	55
6.10	Risk Percentage boxplot	55
6.11	Entrypoint Similarity boxplot	56

List of Tables

3.1	Works about minimalism.	11
3.2	Works about dependencies inference.	14
3.3	Works about network port detection.	16
3.4	Works about entrypoint detection.	17
3.5	All the analyzed works	20
5.1	Dockerfile fields inference map	30
6.1	Hermit's dockerfiles dependencies evaluation	50
6.2	Hermit's dockerfiles dependencies evaluation	51
6.3	Hermit's dockerfiles balance evaluation	53
6.4	Ports evaluation	54
6.5	Risk evaluation	54
6.6	Hermit's dockerfiles entrypoints evaluation	55

Abbreviations

VM	Virtual Machine
LXC	Linux Container Technology
OS	Operating System
SOA	Service-Oriented Architecture
AWS	Amazon Web Services
GCP	Google Cloud Platform
API	Application Programming Interface

Chapter 1

Introduction

This chapter introduces the themes and concepts to contextualize this dissertation. In the first section, Section 1.1, the **Context** of the thesis is introduced. After that, Section 1.2 explains the **Objectives**. The following section, Section 1.3 lists the **Contributions** of this work. Finally, Section 1.4 explains this **Document Structure**.

1.1 Context

Cloud computing had been registering considerable growth in recent years, resultant from the improvements in internet connection speed and hardware infrastructures. While in the past, software only run on the clients' machines, now it can be hosted on a server and be accessed by anyone through an internet connection. With the evolution of the cloud, many systems have been developed with it in mind due to the advantages it offers like bigger availability and flexibility.

However, taking full advantage of the benefits provided by the cloud carries new challenges. For example, those related to deploying, orchestrating, or isolating services from the underlying infrastructure. These problems usually do not represent simple tasks and require hard work to adapt the system to the hardware hosting the service.

Initially, a possible solution to this problem was the use of virtual machines due to their abstraction of the hardware through the operating system's virtualization. However, this comes with an overhead caused from the addition of a new layer in the architecture to emulate the operating system, adding a new cost to the processing speed. With containers, which isolate the software and its dependencies from the underlying operating system and hardware infrastructure, it is possible to remove the barriers to deploy in the cloud and integrate these software in a services-oriented architecture [22].

Docker is the leading technology in the field due to its robust and complex implementation of containers [1, 15].

1.2 Objectives

To define the configuration of a Docker container, developers need to write a dockerfile manually. However, to complete this task, developers need a high-level understanding of the system. Due to the lack of time and experience, containers are created mainly with a trial-error approach. Sometimes these containers present obsolete dependencies and vulnerabilities.

This master thesis aims to study and implement a tool to generate functional dockerfiles. With such automatic service, we could generate dockerfiles with optimizations in memory and security, usually not present in human-made dockerfiles, resulting in smaller and safer containers.

With this research we aim to ease the migration of existing monolithic and complex systems to the cloud.

1.3 Contributions

The contributions that result from this master thesis are:

- A literature review about dockerfiles' characteristics and what techniques exist to help us infer them.
- A novel approach to automatically generate dockerfiles, through the combination of existing heuristics for different individual goals.
- A prototype using the novel approach, to work as a reference implementation, so we can conduct an empirical study to evaluate the approach.
- The design of an empirical experiment to measure the improvements of the automatically generated containers, through comparison with containers defined by dockerfiles present in popular open-source projects. This includes the tools developed for this purpose
- A demonstration of the study referred in the previous bullet point and the respective results obtained.

1.4 Document Structure

This document is split into seven chapters, and the purpose of each one will be explained in this section.

Chapter 1 offers an introduction to its context and a general overview of this dissertation's objectives.

Chapter 2 explains in-depth the concepts and technologies crucial in this dissertation regarding virtualization techniques.

Chapter 3 presents the state of the art to answer the different literature research questions about the automatic generation of dockerfiles

Chapter 4 describes the problem statement, proposes a hypothesis to solve it, a methodology to validate the hypothesis, and a work plan.

Chapter 5 focus on the design of our approach and presents the details regarding the implementation of prototype based on said approach.

Chapter 6 is where we conduct an empirical study to evaluate our approach, and make all the conclusions.

In Chapter 7, there are the conclusions that resulted from our research.

Chapter 2

Background

This chapter presents all relevant concepts and technologies for this dissertation, including the most used **virtualization technologies** like containers, especially Docker containers. Their underlying technology, different types, and advantages will be explained in this chapter too.

2.1 Virtualization Technologies

Virtualization enables the abstraction of the underlying infrastructure hosting the service. That can be done at the hardware level through emulation of entire operating systems with **Virtual Machines** or at the level of the operating system with separated and isolated **Containers** packaging the software with libraries and other system resources.

2.1.1 Virtual Machines

Virtual Machines were the primary approach to virtualization in the last decade. They use an abstraction layer, known as a hypervisor, to run the software in a virtual set of resources that run directly in the hardware and are isolated from the guest operating system.

Virtual Machines can run other operating systems while the host OS is running, resulting in an intensive cost to the CPU and creates too much isolation between the different environments.

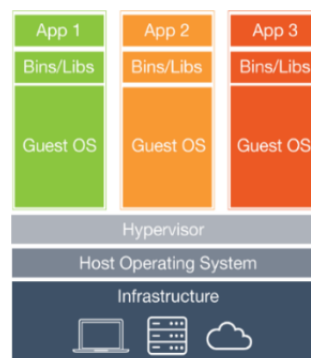


Figure 2.1: Virtual Machine Architecture

A representation of the integration of VMs in the system can be seen in Figure 2.1.

2.1.2 Containers

Containers are a lightweight alternative to Virtual Machines that have been gaining traction in recent years.

To achieve virtualization, the containers isolate the software and its dependencies from the operating system. However, instead of adding new layers to the architecture, they use the native kernel of the host operating system with an alternative set of libraries and system resources different than the ones from the operating system. Figure 2.2 shows how the containers are layered in the system.

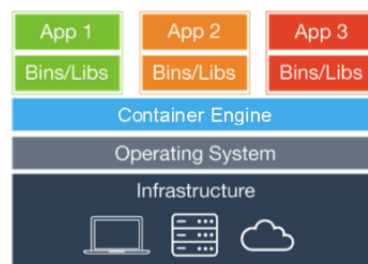


Figure 2.2: Containers Architecture

This approach reflects a lightweight alternative compared to virtual machines, with less processing overhead, easier integration and communication of different containers in a service-oriented architecture and lighter in memory.

Examples of Container Engines include LXC and Docker [21], with this thesis focusing on the last one.

2.2 Docker

Docker [15] is the leader container engine in the industry due to its robust and complete implementation. Implemented in Golang, Docker is a standard for web services implementations and many hosting providers like AWS and GCP have support the deployment of Docker containers [20].

A dockerfile is a file describing the definition and configuration of a container in Docker. This file, similar to a makefile, lists the building steps but for a container. Every line written in a dockerfile will be converted into an additional layer of the final container.

2.2.1 Container Images

Images are the lower layer of a container unless they are built from **scratch**. This specific layer is a read-only set of resources usually present in operating systems and necessary to execute software.

There are images popular due to their stability like **Ubuntu**, and other ones more focused in minimalist containers like **Apline** or Google **Distroless** Images.

2.2.2 Services Architecture

Despite their isolation, different containers can still share resources and memory and communicate with each other, making their integration in service-oriented architecture easier than with Virtual Machines. This middle term of isolation and integration between the different software benefited these architectures [12], which contributed to the massive popularity of Docker in software engineering.

With tools like **docker-compose**, **Kubernetes** or **Docker Swarm**, it is possible to spawn multiple containers and orchestrate them, which benefits the implementation of these architectures in the cloud.

Chapter 3

State of the Art

This chapter groups the related and relevant work regarding different topics necessary to make the generation of dockerfiles possible.

3.1 Introduction

First, it is necessary to understand what constitutes a dockerfile and how can we make good dockerfiles. There are already some empirical studies that analyze existing dockerfiles on GitHub [4, 25], which offer a brief analysis of the aspects that characterize dockerfiles and how they can be optimized. Such aspects are key for automatically containerizing services. This chapter will detail the necessary topics to achieve this goal and their respective related work.

3.2 Goals and Methodology

The first line of dockerfile consists of identifying the image where the container is based. The ideal image of any specific software is the one that only contains the minimum resources for it to work correctly. These resources include libraries of the operating system and other low-level components. So first topic to investigate is which approaches can be used to accomplish minimal containers. These approaches can be about fetching the right one from the preexistent images or building one from scratch to one specific scenario.

With the base image ready and the necessary resources fetched, the next step is to install the dependencies that the service requires to run. These dependencies usually are obtained from the used programming language's package manager. In some programming languages, there are files listing the dependencies (requirements.txt for python, package.json for Node.js) is not always the case. The next topic to study is about the different strategies to infer the dependencies. With the right heuristic to infer this data, the programming language package manager can deal with the subsequent installation.

The majority of Docker-based projects are built to be deployed in the cloud, where network operations are essential tasks, just like referred by the empirical study cited above. For this reason, it is required for Docker containers to expose specific ports or a range of them in order for the containerized software to work just like intended. The next logical topic in the state of the art should document the different methodologies to infer the exposed network ports.

The last essential aspect that a dockerfile must include is the entry-point to run the service. The last important topic is about the various approaches to detect the entry-point of the service.

The security of a container is a tangent problem that can be solved with the previous topics' solutions. However, some additional security measures can be beneficial too. For every topic, if there is any additional security measure that could be applied, it will be discussed too.

Considering all of topics listed, the four research questions that arise with them, and that are the focus of this literature review, are:

- **LRQ1** - What is the optimal approach to achieve a minimal container?
- **LRQ2** - What methodologies can infer the dependencies employed by a software?
- **LRQ3** - How can we detect all network ports possibly used by the service in its entire execution?
- **LRQ4** - What strategies can we use to infer the entry-point of the service?

The existing literature presents different approaches to answer these questions, by inspecting data from the software intended to be containerized. The existing works make this inspection through dynamic analysis and static analysis. The dynamic analysis is about tracking the service's execution and collecting the target data from its system calls. The static analysis is an inspection made to the service's source code and subsequently infers the answers from the parsed data.

In Figure 3.1, there is an example of a dockerfile, for a simple NGINX server, and its 4 important aspects.



Figure 3.1: Basic dockerfile for a NGINX Server and its 4 goals

The following sections will list the related works, separated by topics, and explain their heuristics in-depth to solve the enumerated questions.

3.3 Minimalist Containers

We looked into existing approaches that guarantee that the resultant containers only include the minimal and necessary resources, dependencies and base image. These approaches include the dynamic analysis of the containerized software's execution and the static analysis of the source code or the dockerfile in case of existence. The minimization can occur during or after the containerization. The minimization after a container was created is also known as **debloat**, because before that process, the container is considered to be bloated and the unnecessary resources are known as bloat.

Some of bloat that exists in containers may present some security issues, like vulnerable libraries or files. The removal of this bloat would also remove any potential vulnerability that could be exploited by an attacker, making the container safer.

The table 3.1 lists the works that deal with this problem, the type of analysis they use, when the minimization occurs and the type of the document.

Name	Analysis	Minimization	Type
Container Dockerfile and Container Mirror Image Quick Generation Methods and Systems [28]	Dynamic	During	Patent
Automatic Container Definition [13]	Static	During	Patent
A Novel Approach to Containerize Existing Applications[16]	Dynamic	During	Paper
Optimizing Service Delivery with Minimal Runtimes[8]	Both	After	Paper
FogDocker: Start Container Now, Fetch Image Later[5]	Static	After	Paper
Confine: Automated System Call Policy Generation for Container Attack [7]	Both	Both	Paper

Table 3.1: Works about minimalism.

The results obtained by both types of analysis are promising, which makes us question about the possibility of both in the same process. The minimal resources required for the container can be inferred from the static analysis and then from the dynamic analysis. Also, it would be interesting to investigate if the minimalism can be guaranteed during the container's construction and after applying a debloat.

However, if we compare both types of analysis, the dynamic analysis and the debloat after the container's build process are expected to be more effective than their alternatives. The results obtained by Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel [17], using dynamic analysis, can be seen in Figure 3.2.

Container	Size	Analysis time	Result size	Size reduction
nginx	133 MB	5.5 s	6 MB	95%
redis	151 MB	5.5 s	12 MB	92%
mongo	317 MB	14.0 s	46 MB	85%
python	119 MB	5.3 s	30 MB	75%
registry	33 MB	2.9 s	28 MB	15%
haproxy	137 MB	4.3 s	10 MB	93%
appcontainers/mediawiki	576 MB	16.8 s	244 MB	58%
eugeneware/docker-wordpress-nginx	602 MB	16.2 s	207 MB	66%
sebp/elk	985 MB	26.1 s	251 MB	75%

Figure 3.2: Table with the results obtained from Cimplifier [17]

The security of containers can be enforced, by the lack of obsolete dependencies, which sometimes present potential vulnerabilities to be exploited by attackers. Also, additional security measures can be implemented while building the base container, like a policy of authorized system calls. "Confine: Automated System Call Policy Generation for Container Attack" [7] is a proposal of a technique to enforce the container's security by statically analyze the source code of the software and subsequently infer the only authorized system calls required by the service. If any system call is not authorized, then it will be denied. Figure 3.3 shows how confine works.

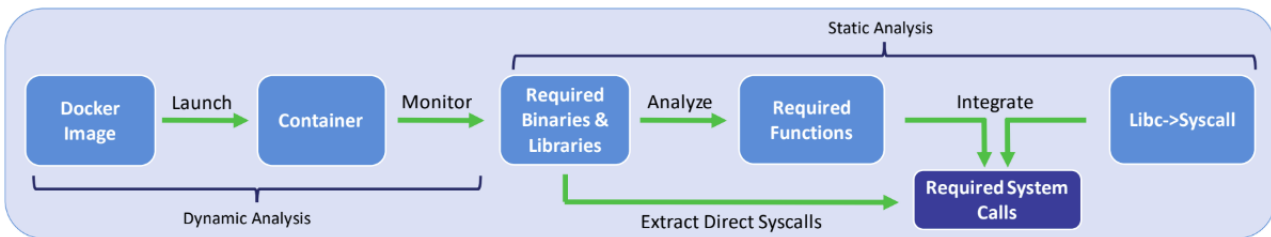


Figure 3.3: Confine's pipeline to select the authorized system calls [7]

The resultant containers won't only be minimalist, but safer too. With the automatic generation of dockerfiles it is possible to deal with issues that can be found in dockerfiles made by humans, like the unnecessary bloat and the vulnerabilities, which can be caused by trial and error approaches or lack of time and experience.

The bloat's existence in the container comes from the image used. These images are based in existing Linux distributions like Ubuntu or CentOS, for example. Despite being lighter than their desktop counterparts, they still have many OS resources in order to be able to run different kinds of software. However, when we are defining a docker container, we know exactly which software we want to run, and every unused resource, provided by the base image, will be unnecessary bloat. The removal of this bloat "would help Docker live up to its claim as a lightweight alternative to standard virtualization" [4], just like ponted by Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall.

Instead of debloating the container after being generated, minimalist images could be used from the start and the resources can be added as they are needed. Examples of minimalist images

include Alpine or Google's **Distroless** Images. In Figure 3.4, there is a flowchart representing the heuristic to dynamically add the resources to a docker container, using a minimalist image.

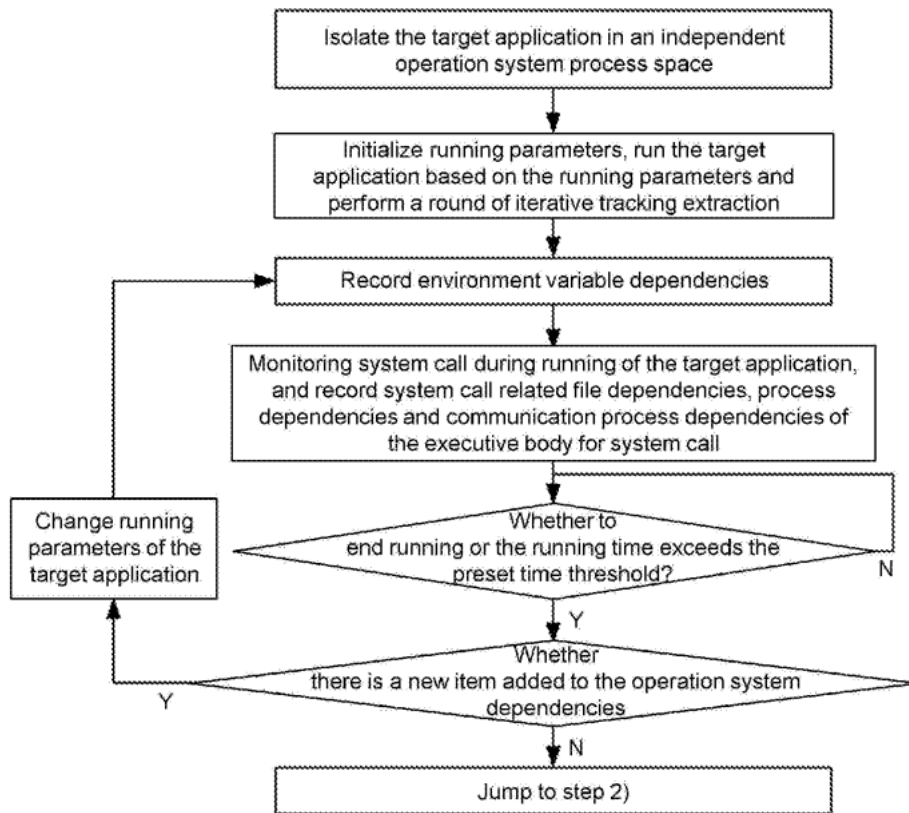


Figure 3.4: Flowchart representing the dynamic analysis to build a container [28]

By taking advantage of the heuristics presented in this section, developers would not only benefit from a decrease in time cost, but from optimized containers. In Figure 3.5, there is a comparison of a minimal container with a normal container and a VM.

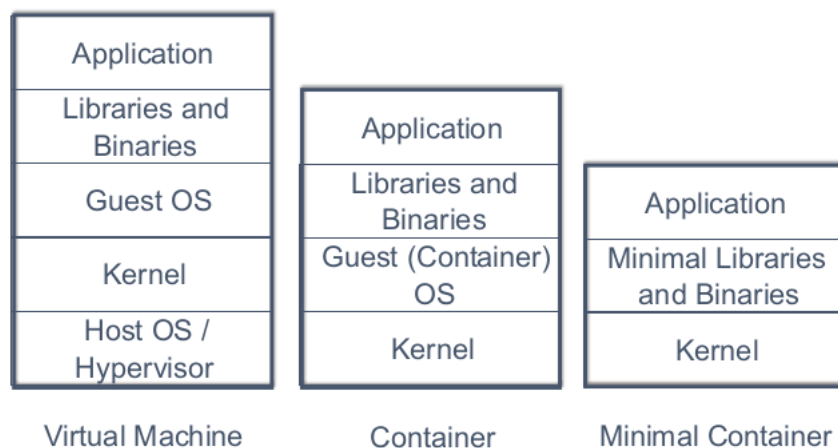


Figure 3.5: Comparison with a Minimal Container [8]

3.4 Dependency Inference

Modern tools can infer software's dependencies through a static analysis of the source code and the dynamic analysis of said software's execution.

The static analysis collects dependencies' data by inspecting the code snippets where files are imported.

The dynamic analysis detects the dependencies by tracking the system calls made by the process, particularly the system calls related to opened files.

Some dependencies depend on other dependencies, and in the majority of the package managers, the installation of the first ones automatically installs the last ones. An example of a dependency graph detailing this can be seen in Figure 3.6.

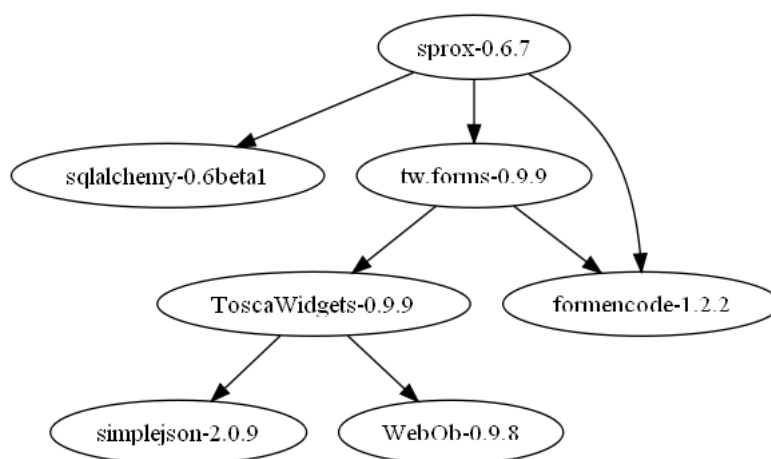


Figure 3.6: Example dependency tree

The comparison and discussion of these approaches are present in this section. The table 3.2 lists the works documenting those approaches.

Name	Analysis	Type
Container Dockerfile and Container Mirror Image Quick Generation Methods and Systems [28]	Dynamic	Patent
Automatic Container Definition [13]	Static	Patent
A Novel Approach to Containerize Existing Applications [16]	Dynamic	Thesis
Dockerizeme: Automatic inference of environment dependencies for python code snippets [11]	Static	Paper
Dependency Management in Software Component Deployment [2]	Static	Paper
RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis [9]	Static	Paper

Table 3.2: Works about dependencies inference.

Once again, the strategies presented in this topic are not mutually exclusive and can be applied in a combined methodology. The techniques used to achieve the inference of dependencies are very similar to those used in the previous topic but are in a more developed state.

With static analysis, it is possible to generate a dependency graph containing the data of the dependencies, just like suggested by Kumar [13]. This graph enables the elimination of duplicate dependencies. A pipeline detailing this strategy can be seen in Figure 3.7.

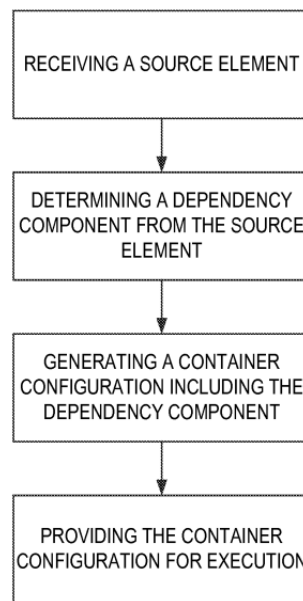


Figure 3.7: Pipeline of static analysis, used in "Automatic Container Definition" [13]

The dynamic analysis proved again to have more advantages than the static one, due to the higher probability of correct inferences, despite the possibility of missing specific dependencies.

The dependency graph, generated by the static analysis approaches, represents data with significant contributions, not just for the current topic but for all of them. Figure 3.8 represents how those graphs work.

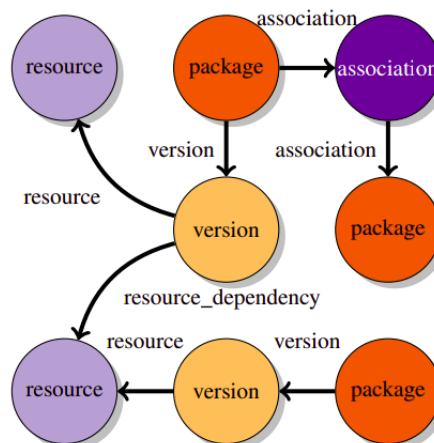


Figure 3.8: Dependency graph [11]

3.5 Ports Detection

There are many wellknown tools to detect the network ports assigned to specific processes like netstat for example. By inspecting a software with these tools, the the ports that must be exposed can be detected.

Regarding the static analysis of the source code, at the moment, there is not any experiemental work with reliable results.

The most promising works that deal with this problem can be seen in table 3.3

Name	Analysis	Type
Container Dockerfile and Container Mirror Image Quick Generation Methods and Systems[28]	Dynamic	Patent
A Novel Approach to Containerize Existing Applications[16]	Dynamic	Thesis
VM2Docker: Automating the Conversion from Virtual Machine to Docker Container[14]	Dynamic	Thesis

Table 3.3: Works about network port detection.

Due to the lack of robustness in static approaches, the dynamic ones are the only reliable approaches.

Taking advantage of a system calls tracer like strace, which is compatible with the other topics, the assigned ports can be dynamically inferred from the software's system calls to bind them. The data collected from the tracer will be translated to the list of exposed ports in the dockerfile. This approach has the advantage of inferring some of the ports that are bound in specific moments. However, if specific scenarios, where some network ports are bound in runtime, do not happen, then the said ports will be missed.

Another alternative to discover the ports bound to a specific process is using **Nmap** or **netstat** to get a map of running services and their respective ports bound. The analysis made by these tools is momentary and can miss certain ports because due to bad timings.

Considering all of these approaches, the first one seems more promising to infer more ports.

3.6 Entrypoint detection

This section lists different approaches to infer the entrypoint source code file to run the service, whatever it is a static analysis or dynamic analysis.

The only static approach is at the moment an hypothesis and experimental, while the dynamic approaches are more concrete and tested. These approaches are listed in table 3.3

The only static approach is a hypothesis and still experimental, while the dynamic approaches are more concrete and tested.

Through dynamic analysis, the entrypoint file is detected by inspecting which file is the first to be opened by the service. However, it is required to know how to run the software, which in certain cases include knowing the entrypoint, for the dynamic analysis to infer it, in first place.

Name	Analysis	Type
Container Dockerfile and Container Mirror Image Quick Generation Methods and Systems[28]	Dynamic	Patent
Automatic Container Definition[13]	Static	Patent
A Novel Approach to Containerize Existing Applications[16]	Dynamic	Thesis

Table 3.4: Works about entrypoint detection.

The static technique instead infers a graph that describes which files call each one of them. The first node of the graph is the entrypoint. However, it is not confident that the graph will present only one starting node.

In Conclusion, the dynamic approach is again more reliable than the static counterpart. This last methodology is not robust enough due to its status of being a hypothesis.

3.7 Empirical study about dockerfiles

For this dissertation, it is vital to understand the dockerfiles ecosystem and its most common issues. One particular research was an empirical study conducted by Cito et al. [4], just like referred to in the Introduction of this chapter.

In this study, the authors analyzed over 7000 dockerfiles and their respective containers to understand the most prominent quality issues. In the end, they concluded that the migration of those containers to minimalist images would be a significant improvement, just like we concluded in the section about minimalism.

One particular statistic measured in the study is the distributions of lines changed after the dockerfile is completed. Figure 3.9 shows a bar graph with those values.

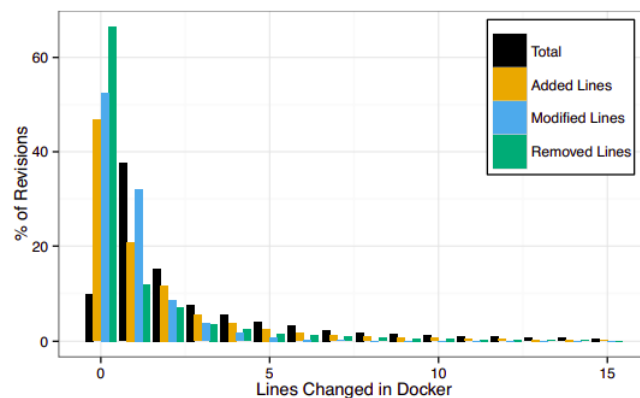


Figure 3.9: Distribution of revision sizes measured in lines changed [4]

As we can see, any modification after the dockerfile is completed is very scarce. These changes usually happen if the system receives a new extension, requiring the said change in the dockerfile.

To optimize the generation of the dockerfiles for specific programming languages, it is important to understand which languages are the main focus for docker containers. Cito et al. [4] tried to

understand this by analyzing the distribution of the languages used in GitHub projects employing docker. In Figure 3.10, there is a graph representing this distribution.

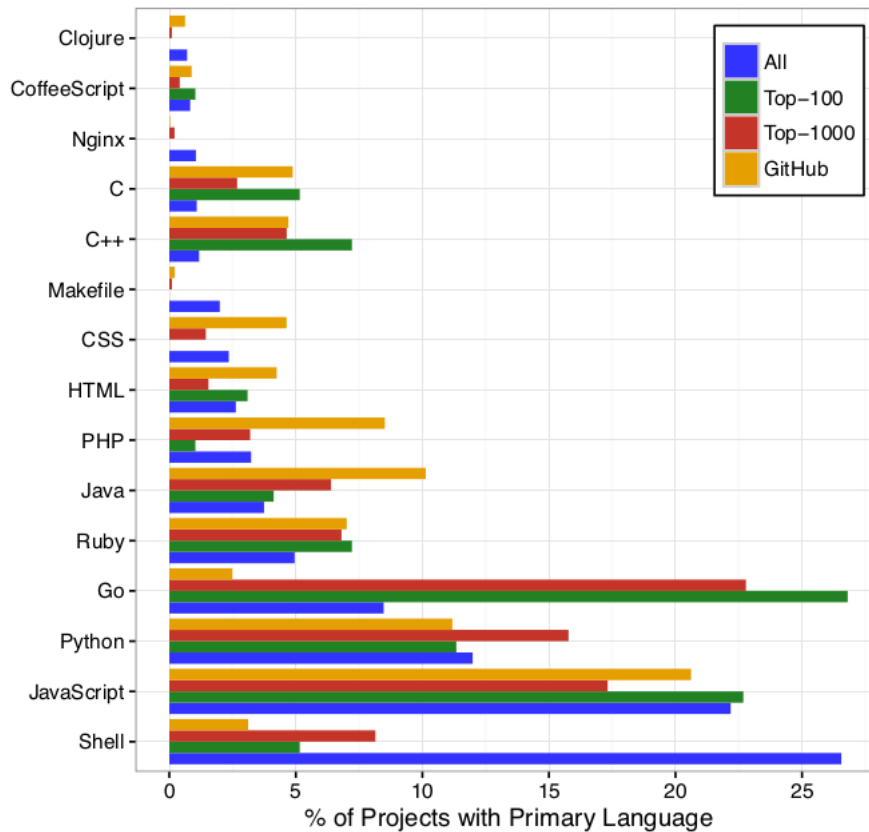


Figure 3.10: Distribution of top 15 languages [4]

From this data, we can conclude that most GitHub projects that target docker containers use scripting languages, where Go (also known as Golang) is the main exception. The top 4 programming languages targeting docker are Go, Shell Script, JavaScript and Python. Considering this information, our work will focus on that languages, and the containerization of projects using them will be prioritized. These languages also have the advantage of having robust package managers associated, which eases the installation of dependencies. Their package managers and their robust toolchain ease the process of "build and run." For this reason, they do not require complex dockerfiles to build them, but require all the necessary runtimes to execute. Simpler languages like C or C++ lack those toolchains and do not require additional runtimes, which requires complex dockerfiles to build the statically compiled binaries, but does not require heavy images.

Another useful statistic, registered from the analysis of the dockerfiles present in GitHub, is the distribution of the base images referenced in the FROM statements. A bar graph detailing this information can be seen in Figure 3.11.

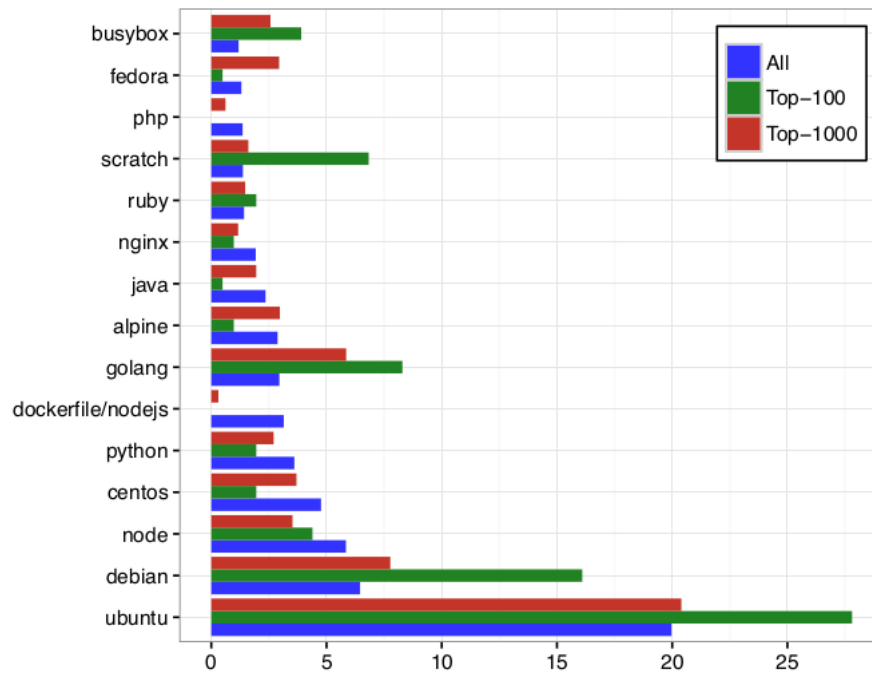


Figure 3.11: Percentage of Projects with Base Image Referenced in FROM Statements [4]

Ubuntu is easily the winner from the images listed, which is explained by the stability that this image offers. It is interesting to notice that the scratch image is mostly used in the top 100 projects, proving that minimalism is a big concern for many projects using statically compiled binaries. However, other minimalist images like the ones specific for the languages like java or python or a more generic one like Alpine are not that popular. However, Golang is a surprising exception. The second place of Debian can be explained for its middle-term between minimalism and stability. To better understand the focus of the images employed, Figure 3.12 shows a bar graph with the usage of different types of images.

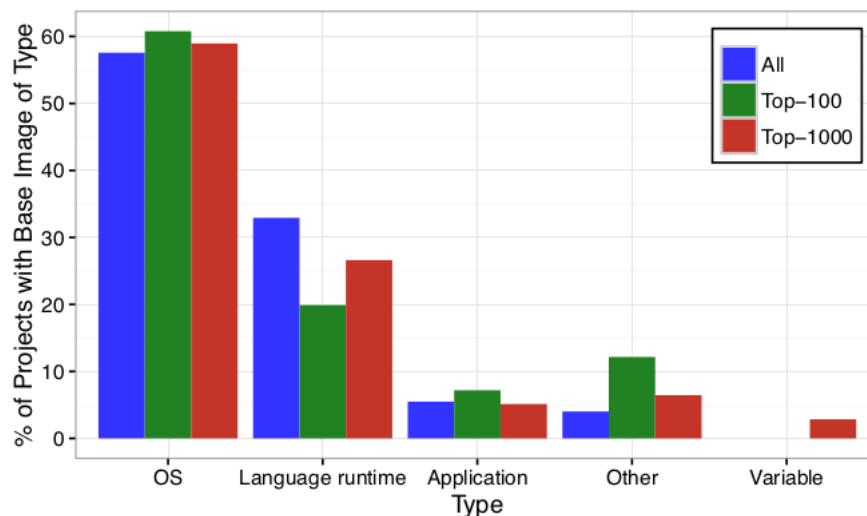


Figure 3.12: Percentage of base image types across top 25 images [4]

With this analysis, we conclude that the developers prefer to use OS-based images to prioritize stability and ease of deployment in the cloud but ignore the bloat or vulnerabilities that may be present.

3.8 Conclusions

Looking for the analyzed works, we have enough approaches to answer the 4 literature research questions listed in above. However, regarding the detection of the entrypoint, there are still many challenges, due to the lack of robust static approaches and to the holes that still exist in the dynamic strategies. Despite that, we still have the necessary information to build our own approach.

In Table 3.5, there is a correlation between the analyzed works and which goals they try to solve.

Name	Minimalism	Dependencies	Ports	Entrypoint
Container Dockerfile and Container Mirror Image Quick Generation Methods and Systems[28]	●	●	●	●
Automatic Container Definition[13]	●	●	●	●
A Novel Approach to Containerize Existing Applications[16]	●	●	●	●
Confine: Automated System Call Policy Generation for Container Attack Surface Reduction [7]	●			
Cimplifier: Automatically Debloating Containers [17]	●			
New Directions for Container Debloating [18]	●			
Optimizing service delivery with minimal runtimes [8]	●			
FogDocker: Start Container Now, Fetch Image Later [5]	●			
Dockerizeme: Automatic Inference of environment dependencies for python code snippets [11]		●		
Dependency Management in Software Component Deployment [2]		●		
RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis [9]		●		
VM2Docker: Automating the Conversion from Virtual Machine to Docker Container [14]			●	●

Table 3.5: All the analyzed works

From this table, there are 3 works that try to accomplish the 4 goals. The first two presented in the table [13, 28] represent patents with interesting ideas supporting them, but no concrete implementations or solutions.

Vineet Palan, in his master thesis "A Novel Approach to Containerize Existing Applications" [16], presented an heuristic to complete the 4 goals, using an virtual machine as an auxiliary system. His research was based in Eric Lubin's master thesis "VM2Docker: automating the conversion from virtual machine to docker container", which proposed a conversion from existing virtual machines to docker containers. The contributions, made by Palan, adapted this

strategy for the generation of dockerfiles and filled some gaps regarding the minimalism of the container and the inference of the dependencies.

Despite being a big contribution to the automatic generation of dockerfiles, the use of the virtual machine still presents a bottleneck to this process. Running VMs is a task intensive to the machine in both processing and memory. The instance of a Virtual Machine has a cost in time, which may be counterproductive, considering that the automatic generation of dockerfiles was supposed to reduce this cost. For this reason, it is crucial to research some alternative strategies to infer all the necessary data, to fill the gaps occupied by virtual machines at the moment.

To evaluate the results obtained, Palan only measured the size of the containers obtained, but lacks empirical evaluation of other metrics. Not only him, but all the authors of the works presented did not offer complex benchmarks to measure their results. For example, we believe that the dockerfile generation success ratio and the accuracy of the inferred dependencies are important metrics to measure in automatic generations of dockerfiles. For that reason, it is important to expand the criteria of empirical studies used in this field.

To conclude, by taking advantage of the approaches listed in Table 3.5, we may discover a new solution to complete the 4 goals, fill the gaps that still exist and remove the current bottlenecks that prevent us from achieving an entirely automatic containerization process [6]. To measure the solution's performance, the current metrics are not enough, which will require the exploration of new approaches.

Chapter 4

Problem Statement and Research Strategy

This chapter describes the current challenges to automatically generate dockerfiles and formulates a hypothetical solution to them.

Firstly, we introduce the **Scope Definition** (*cf.* Section 4.1), which contextualizes the scope of the problem, according to the researched literature. Then, we present the **Hypothesis** (*cf.* Section 4.2) that underlies our work. That hypothesis is put in practice through a defined **Methodology** (*cf.* Section 4.4). After that, the **Planned Contributions** (*cf.* Section 4.3) will be discussed.

4.1 Scope Definition

Dockerfiles are usually made by human developers, lacking any robust tool to automatically generate them.

The development of dockerfiles requires good knowledge about docker and the system intended to be containerized. Writing a dockerfile is a task that is not always optimally practiced, but with a "trial and error" approach. Sometimes even with unnecessary bloat, and the optimizations to fix it require more effort. With an automatic process this additional effort can be avoided.

Despite the use of Docker containers to reduce the burden of deploying services in the cloud and orchestrating them in services-oriented architecture [12], the need to write a dockerfile still requires some effort from the developers [3]. This is one of the many tasks necessary to move systems to the cloud [23, 24] and becomes a bottleneck that prevents the automatic containerization of existing software.

Sometimes, the developers' lack of time and experience can result in unoptimized containers with obsolete dependencies or potential vulnerabilities, which doesn't take full advantage of the very concept of "container" as a lightweight alternative to standard virtualization [4].

The few existing heuristics that aim to containerize existing software still lack concrete implementations [28, 13] or demand heavy resources like virtual machines [16]. VMs are not ideal because this process will take more time than a human doing this task and are CPU intensive.

The automatic generation of dockerfiles would by extension make the containerization of existing software possible. Figure 4.1 represents a hypothetical pipeline of complete automatic containerization.

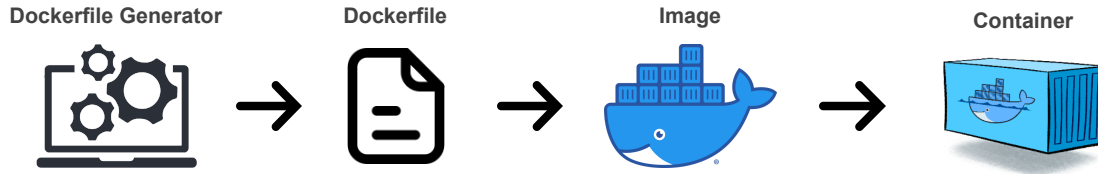


Figure 4.1: Complete containerization's hypothetical pipeline

Some interesting solutions have been proposed, including two promising patents, one using dynamic analysis [28] and the other with static analysis [13]. However, despite been registered and having interesting ideas supporting them, no concrete implementations of these patents are known. Palan [16] and Lubin [14], in their respective dissertations, presented strategies to automatically generate the containers, but by using virtual machines as an intermediary step, instead of dynamic analysis and static analysis exclusively.

Despite not having any concrete approach to generate dockerfiles, this can be accomplished using the different heuristics presented in the **State of the art**, both static and dynamic, to infer the necessary data to generate a dockerfile. While there are promising techniques to infer the majority of the goals, we are skeptic about the efficacy of the ones that detect the entryptpoint. For this reason, we think we could try an alternative authored by us.

4.2 Hypothesis

Considering the findings presented in the previous chapter, there is a lack of tools to automatically generate dockerfiles using dynamic analysis and static analysis and aiming to reduce the effort associated to the creation of dockerfiles. However, recent scientific contributions present different strategies to infer the multiple instructions of a dockerfile. It is not guaranteed that if by combining them we could fully generate working dockerfiles for any service, including scenarios that require more complex container configurations.

The hypothesis proposed in this work is that *"We can generate dockerfiles, through static and dynamic analysis, with suitable dependencies, network ports and entryptpoint"*. These dockerfiles must work properly and build stable containers. By "suitable", we mean that these automatically generated dockerfiles present all the necessary and appropriate resources that the service requires. By "dynamic and static analysis" we mean that the inferred data will be obtained exclusively

through the "dynamic" analysis of the system calls' logs and the "static" analysis of software's source code.

If this hypothesis is verified, developers will benefit from the automation of this task, instead of doing it manually.

4.3 Research Questions

With our hypothesis defined, we can proceed to the development of our approach and conduct a study to empirically evaluate it. This study will focus in certain topics, which are the basis of the formulated research questions:

- **RQ1** - To what extent it is possible to detect all dependencies without adding bloat? — Firstly, we want to study if our approach can infer all the dependencies that were present in the original dockerfile, without adding bloat not existent in the original container. These dependencies may come with the base image or may be installed by both system and languages package managers. So, this topic will be related to the inference of the base image and the inference of the dependencies that are installed by the package managers. The metrics of the evaluation relevant for this topic are the ratio of the bloat, the ratio of dependencies missed and the balance of different package versions.
- **RQ2** - To what extent it is possible to detect the ports to expose without adding unnecessary risks to the service? – The second topic of our study is about the accuracy of our prototype to detect all the exposed ports, while avoiding to expose of unused ports and all the risks associated to it. For this topic, we need to understand the accuracy of the ports detected and the percentage of risk.
- **RQ3** - To what extent it is possible to infer a suitable endpoint? — The last topic must focus in the inference of a suitable endpoint to run the service. This evaluation is based in on the endpoint similarity.

After conducting our methodology and discussing the obtained results, we will be able to present our answers for these questions.

4.4 Methodology

To test the hypothesis and answer our research questions, a methodology was formulated. The steps that describe it are the following ones:

- a. Design of an approach to generate dockerfiles, using the techniques studied as inspiration. To define this approach, we must map the information tracked from the system (dynamically or statically) to each respective inferred field of the dockerfile.

- b. Development of the tool to automatically generate dockerfiles to existing projects to empirically evaluate our hypothesis. This tool works as an implementation reference for other researchers. It will be open-source in order to be available to everyone to see and contribute. To make this possible, it will internally use different approaches from the literature review. The tool will incorporate the following strategies to accomplish the 4 goals:

- **Base Image** — Use of images more oriented to the language used in the project. For multi-stage dockerfiles, with minimalist containers, can also be used **minimal images** like **Alpine** or the **Distroless** images provided by Google.
- **Dependencies inference** — Infer the dependencies by cross-referencing the files dynamically opened with the **openat** syscall with the statically generated **dependency graph**.
- **Ports Detection** — Detect the **network ports** through the **dynamic analysis** of the **bind** syscall.
- **Entrypoint detection** — Infer the **entrypoint** from the **first command** executed by the system calls **exec** or **execve**.

- c. Empirically evaluate the tool by conducting some experiments with the **Hermit** and randomly selected open-source projects from Github. The dockerfiles present in these repositories will be compared with the dockerfiles generated by the tool. The containers built from the automatically generated will be compared with the container that results from the original dockerfile. This comparison will use the following metrics:

- The ratio of bloat in the new container that didn't exist in the original one.
- The ratio of the dependencies missed from the original dockerfile.
- The balance that results from the using different versions of the same packages in both containers.
- The accuracy of the ports detected.
- The percentage of risk added through the exposure of unnecessary ports.
- The similarity between the entrypoint of the automatically generated dockerfile and the original dockerfile.

Chapter 5

Approach and Design of Hermit

This chapter presents an approach to automatically generate dockerfiles, and the prototype's design, based on said approach, to work as a reference implementation.

Section 5.1 describes the approach to automatically generate dockerfiles, followed by an explanation of the prototype's implementation in Section 5.2. Finally, Section 5.3 lists the different features that are available in Hermit.

5.1 Approach

Our approach can be split into three steps: project analysis (*cf.* Section 5.1.1), data inference (*cf.* Section 5.1.2) and dockerfile generation (*cf.* Section 5.1.3). In Figure 5.1, we can see a diagram representing our approach.

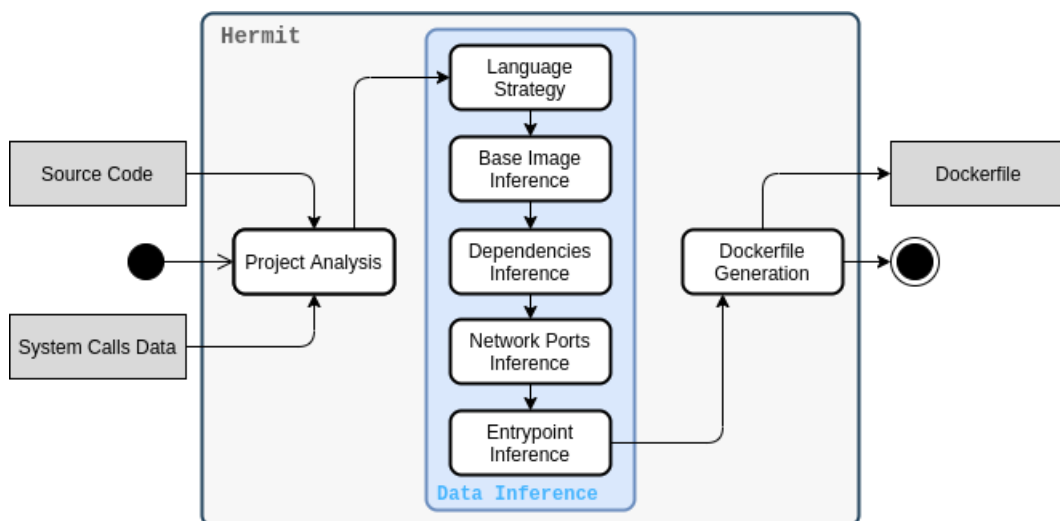


Figure 5.1: Approach's activity diagram

5.1.1 Project Analysis

In this first step, we want to collect all data to understand the service and what resources it requires. This process is done through dynamic analysis that collects the system calls logged from the service and static analysis of the source code. Just like stated previously, certain types of data can only be collected from one type of analysis, which justifies the use of both types. The following sections will explain how each of these analyses can be used to get that data, why we use them, and their limitations.

5.1.1.1 Static Analysis

With static analysis, we scan the repository and recursively inspect its files and folders. During this process, we count the number of occurrences of existing file extensions to determine what languages were employed for the project. The language most used will be assumed to be the principal language, which will influence the next steps of the approach, like the selection of a base image.

There are also static analysis strategies oriented for specific programming languages. At the moment, our approach tackles the JavaScript and Python languages with the following strategies:

- **Python** — Look for the existence of `requirements.txt` or a `Pipfile`, to find what dependencies pip must install and indirectly what system dependencies must be installed before.
- **JavaScript** — Look for the `package.json` and extract important scripts from it, for example the start script to discover the entrypoint.

The information outputted by this phase uses the following format:

```
SourceInfo {  
  files: number,  
  language: string,  
  web: boolean,  
  scripts: Map<string, string>,  
  langs: Map<string, number>  
}
```

5.1.1.2 Dynamic Analysis

By tracking all the system calls, or `syscall` for short, made by the service during its execution, we can infer resources that are only used in run time like the ports or shared libraries called dynamically. Some software can attach to processes and track its system calls, including system calls made by the process' children. The syscalls are stored in the following format:

The information outputted by this phase uses the following format:


```

Syscall {
  syscall: string,
  args: Array<any>,
  result: number,
  timing: number,
  pid: number,
  type: string
}

```

The different types of syscall that we track and the information that we can extract from them are:

- **open/openat** — The system calls `open` and `openat` are executed every time a process tries to open a file and the first argument received is the file name. Depending on the Linux distribution used, some operating systems use `open`, while others use `openat`. By tracking all the system calls of type `open/openat`, we can see the files opened by the service, such as language dependencies or dynamic shared libraries. If the system call returns a non-zero code, it means that the file does not exist and may be discarded from the dependencies.
- **bind** — When a process requests the allocation of a port, this process is made through the system call `bind`. The port requested can be found in after the field `sin_port` or `sin_port6`. With this syscall, we can track every port requested by the service and fill the dockerfile with it. Nevertheless, If the requested port is 0, it means that the service asked for a random port which means that all ports must be exposed.
- **exec/execve** — When a process executes another process, it requires the system call `exec` or the system call `execve`, depending on the distribution. The arguments of this system call include the process called, the arguments passed to said process and the environment variables. With this system call, we can track all the commands executed by the service, including sub-processes. Usually, the entrypoint is the first of the list. We can also discover some auxiliary programs that must be installed by the package manager of the system.

The system calls collected are grouped in the following structure:

```

SystemInfo {
  openat: Array<Syscall>,
  bind: Array<Syscall>,
  execve: Array<Syscall>
}

```

5.1.2 Data Inference

After all of the project's data is collected, we can infer information essential to generate the dockerfile needed for the service to work correctly in a container. Table 5.1 shows which data can be used to infer each information.

Base Image	
• Project's main language	• Language oriented Image
Dependencies	
• System call open/openat • Project's source data	• System dependencies • Language Dependencies
Network Ports	
• system call bind	• Ports allocated during runtime
Entrypoint	
• system call exec/execve	• Processes called during execution

Table 5.1: Dockerfile fields inference map

This section of our approach can be split into 5 different phases, each one to infer specific parts of the dockerfile, which are the **language strategy**, the **base image inference**, the **dependencies inference**, the **network ports inference** and the **entrypoint inference**, just like seen in Figure 5.1. The description of each one of these approaches can be found in the following sections:

5.1.2.1 Base Image Inference

For every language, there are images more oriented to them. These dependencies contained in those images are usually employed in projects of that language, guaranteeing its stability without adding too much bloat. Our approach maps each language to two images: the default image and the first one's distroless counterpart aimed for minimalism. Both images are Debian-based to avoid the unexpected behaviors that alpine-based images sometimes exhibit. For example, for javascript the images would be **node:14** and **gcr.io/distroless/nodejs:14** and for python the images would be **python:3.8-slim** and **gcr.io/distroless/python3**

With the language strategy, we obtained two images, the default one and the distroless version. Our default approach will only choose the first one in the list, which is the default one. However, suppose we want a multi-stage dockerfile taking advantage of the second image's minimalism. In that case, both images will be selected, with the first one for a stable build and the second one only for execution in a minimalist environment.

To avoid the request to the package manager to install dependencies already installed, we must first identify what packages already come installed with the base image. With this information, we can remove them from the list of inferred packages.

5.1.2.2 Dependencies Inference

The inferred dependencies include both installed through the package manager of the operating system or the ones that can be installed with the language package manager.

The system packages can be discovered from the logged system calls `open` and `openat`. We can extract from these logs the name of the file that the system call tried to open, filter those that have extension `.so` (meaning shared object) and query the system package manager what package owns that file. These files are shared dynamic libraries called in runtime by the service, and if they are part of a dependency, we can ask the package manager what the name of the package is. For example, in debian-based operating systems the command is: `dpkg -S <filename>`. If any package does not own the file, it will not be included and if the detected package is referred to in the list of the image's installed packages, it will not be included in the final list of dependencies. When the static analysis detects the existence of specific files like `requirements.txt` or `Pipfile` in a python project, our approach includes additional packages in the dockerfile like `python-dev`, `cmake`, `build-essential` and `pkg-config`, in order for `pip` to have all the necessary tools to build the python dependencies.

Different languages have different packager managers, so depending on the language used in the project, the commands to install the dependencies will be chosen. The language dependencies can be detected using existing approaches, which differ slightly from language to language. Due to the limitation of our approach, to support only JavaScript and python, we only incorporated approaches to infer dependencies of these languages, which are:

- **Python** — If no `requirements.txt` and `Pipfile` exist, we can use existing tools like **pipreqs** or **pigar**. These tools analyze the project, tracing which file imports each one of them and query **pypi**, the repository of dependencies, to discover which ones exist. After discovering all the packages, the `requirements.txt` will be automatically generated. However, if there is a `Pipfile`, we can add another installation step to call **pipenv** to generate a `requirements.txt` from the `Pipfile`. If the `requirements.txt` exists, then the inference is skipped.
- **JavaScript** — JavaScript projects are always accompanied by a file, known as **package.json** that lists all the dependencies, automating their installation.

Both the system packages and the system libraries are outputted in the following format:

```
DependenciesData {  
  packages: Array<string>;  
  libraries: Array<string>;  
}
```

5.1.2.3 Network Ports Inference

All the ports used by the service are first requested to the operating system using the system call `bind`. The ports can be found in these system calls, specifically in the fields **sin6_port** or in the field **sin_port**. All of the detected ports are the same ones we want to expose in the dockerfile, except port 0, which is technically not a port but the argument used when a service requests a random port.

5.1.2.4 Entrypoint Inference

The entrypoint, in the majority of the cases, be inferred from data obtained through dynamic analysis. However, if a start script was statically detected (in a package.json), it will be assumed as the entrypoint.

Unfortunately, this phase of the data inference becomes irrelevant, at the moment, users are required to supply the start command to begin the dynamic analysis. The only situations where this is not necessary are when we can detect a start script or when the project already possesses a dockerfile, and we can run the service inside of a container (cf. Section 5.3.2). Only in these scenarios we can truly infer the entrypoint.

When no start script is found, we can look for the entrypoint in the system calls `exec` or `execve`. Iterating from the first system call to the last one, the first syscall executing a command that includes the language's runtime will be considered the service's entrypoint.

Different languages have different runtimes associated with them. The identification of these runtimes can help to filter the system calls that could have the entrypoint.

5.1.2.5 Other data

There are other type of data that we can infer, but the approach to achieve that differs from language to language. Our approaches takes the following data:

- **The environment variables** — Set the value of some environment variables that are important for a specific language. For example, **PYTHONPATH** for a python project to use local dependencies instead of the global ones, which eases the implementation of multi-stage dockerfiles.
- **The ignored files** — Certain languages tend to generate temporary files, like cache, during their installation or their execution. Considering that our approach containerizes projects after executing them, ideally, we generate a **.dockerignore** file filled with all the ignored files. Some examples include the **node_modules** in JavaScript projects and a **__pycache__** folder in python.

5.1.3 Dockerfile Generation

With the data obtained from the data inference, we can now proceed to generate the dockerfile, using a template of a dockerfile that will be filled with said data. This information is expected to be received in the following structure:

```
DockerfileData {
  images: Array<string>,
  systemPackages: DependenciesData,
  dependencies: Array<string>,
  ports: Array<number>,
  entrypoint: Array<string>,
  envVars: Array<string>,
```

```
filesIgnored: Array<string>
}
```

The construction of the dockerfile is based in the following sequence:

1. The first line of the dockerfile references the base image selected:

```
FROM <image name>
```

2. The source code of the project is copied to a folder named "/app", which will be defined as our workdir (working directory):

```
ADD . /app
WORKDIR /app
```

3. If the number of inferred system packages is bigger than zero, our dockerfile updates the package manager's package list and installs all the inferred packages. If not, it is skipped:

```
RUN apt-get update \
    apt-get install -y --no-install-recommends \
    <list of packages separated by whitespace>
```

4. After the system packages are installed, the dockerfile runs the language installation steps to install the language dependencies:

```
RUN <step 1>
...
RUN <step n>
```

5. If the dockerfile we are generating is a multi-stage one, the next inserted line must import the minimalist image and copy the "/app" folder from the first stage to the current stage, defining it as the workir too. Otherwise, this step is skipped:

```
FROM <minimalist base image>
COPY --from=build-env /app /app
WORKDIR /app
```

6. If the number of defined environment variables is bigger than zero, our dockerfile defines the environment variables. Otherwise, it is skipped:

```
ENV [<environment variable name>=<variable value>,,]
```

7. All the detected ports will be exposed, unless their count is null:

```
EXPOSE [<port>,,]
```

8. The entrypoint is attributed to the container's default command:

```
CMD <service entrypoint>
```

5.2 Prototype - Hermit

Using the approach we presented in the previous section, we developed a prototype of a tool that we called **Hermit** to automatically generate dockerfiles. Its name **Hermit** is a reference to the hermit crabs, which are crabs that try to grab the perfect shell to their body, without being neither too large or too small, just like we want docker containers to fit the software correctly. Hermit works as a reference implementation for other researchers trying to replicate our study as well as for industry practitioners wanting to use it in their contexts.

In this section, we explain some aspects of Hermit's implementation, including its **architecture** (*cf.* Section 5.2.1), the design decisions (*cf.* Section 5.2.2), and the current limitations (*cf.* Section 5.2.3).

5.2.1 Architecture

Hermit's architecture was designed to be highly modular, where every component is plug-able and represents independent aspects of our approach. Thus, other researchers can easily replace any of these modules with alternatives developed by them and retry the empirical study. Furthermore, all of these modules are agglomerated in a library that is imported by Hermit's command-line interface, but can also be imported by other tools or libraries.

Hermit's conceptual overall architecture is depicted in Figure 5.2.

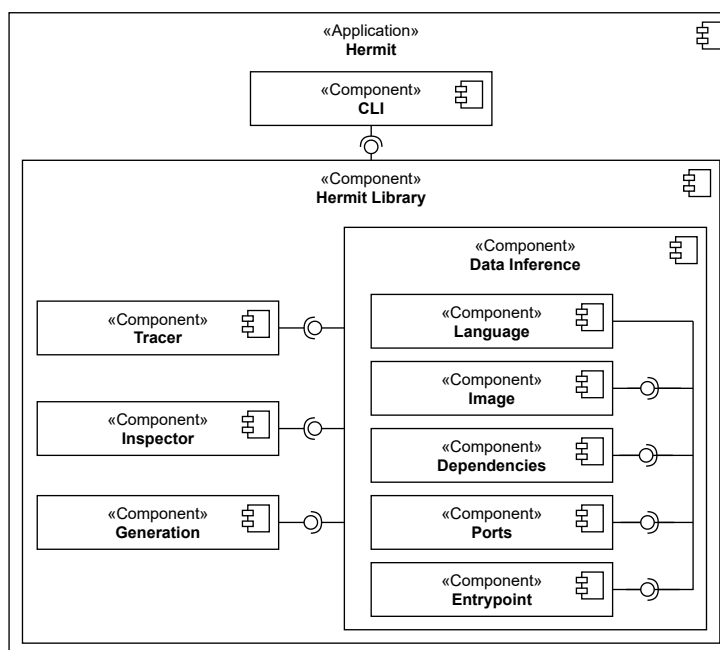


Figure 5.2: Hermit's architecture diagram

Hermit was architected to have its logic and functions grouped into what we call the **Hermit Library**. Hermit, as an application, presents a command-line interface that imports the functions from the library and establishes a communication bridge between users and the containerization logic. With this independence of the containerization functions from the main application, we enable the possibility of Hermit library being imported and integrated in other tools, just like the CLI does.

Inside of the library, there are the different modules that are part of the containerization approach. Each one of these modules are independent from each other and easily replaceable, enabling others researchers to try alternatives developed by them. Furthermore, each one of these modules represents different phases from the approach we proposed, which are:

- **Tracer Module** — Responsible for the dynamic analysis. This module requires the monitoring tool **strace** in order to track and log the system calls executed by the service. The data retrieved by this module is supplied to the modules of the Data Inference group.
- **Inspector Module** — This is the module that statically analyzes the project's repository. Like the previous module, this one sends the data it retrieves to the modules responsible for the data inference.
- **Language Module** — Every strategy specific to every language can be found in this module. This includes the other data we referred to above (cf. Section 5.1.2.5). This module uses the data retrieved by the previous modules to select the appropriate language strategies and supply them to the remaining modules in the data inference group. At the moment, Hermit only supports strategies for Python and JavaScript.
- **Image Module** — Module responsible to infer the base image. The result may include more than one image, in scenarios of multi-stage dockerfile.
- **Dependencies Module** — This module infers the dependencies and the instructions to install them. It is aided by **dpkg**, the tool used to manage system packages in Debian-based operating systems.
- **Ports Module** — The ports are inferred in these modules.
- **Entrypoint Module** — This module is responsible for the entrypoint inference. However, like previously stated, this module becomes useless when users are required to supply the start command.
- **Generation Module** — The data received from the data inference group is used by this module to generate the dockerfile.

5.2.2 Design Decisions

During the development of Hermit some design decisions were made, to overcome the challenges that appeared, and would like to highlight in this section:

1. Challenge: How can other researchers use Hermit but with new ideas of their own?

Decision: The strategies used to infer every piece of data are modularized, making Hermit more plug-able, where every module is easily replaceable. In that way, other researchers can replace only certain parts of the approach and retry the experiment.

2. Challenge: Is it possible to integrate Hermit's entire approach in other tools like Dockerlive [19]?

Decision: We confined the entire approach in a library, that agglomerates all the modules, which in turn is imported by Hermit's command-line interface. The objective behind this design decision is to support the integration of this library in other tools like Visual Studio Code extensions, including Dockerlive ¹.

3. Challenge: How many languages does Hermit support?

Decision: Due to time constraints, we could only support two languages. The chosen languages were JavaScript and Python due to being in the top 3 of the most used languages with docker and due to our previous experience with both languages.

4. Challenge: Base images are based on existent Linux distributions, and which ones should be prioritized?

Decision: We decided to prioritize Debian-based images due to their popularity, which gives more stability and more support from the developers.

5. Challenge: Different operating systems have different behaviors, and their system calls use different conventions. Which operating system is supported right now?

Decision: Hermit was developed for Debian-based operating systems, due to their proximity to the base images used in the dockerfiles.

5.2.3 Limitations

The current implementation of Hermit presents some limitations that prevent us from generalizing our approach to all languages and environments. Some of these limitations are:

- Only supports two languages: Javascript and Python.
- Only works in Debian-based operating systems
- Only generates dockerfiles with Debian-based images.
- Hermit cannot run the service if the user does not supply the correct start command. This makes the entrypoint inference irrelevant. However, if the project has a package.json with a start script or a dockerfile, Hermit can extract the entrypoint from those files and run the service without the user's input. Thus, in this last situation, there is a true entrypoint inference.

¹<https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive>

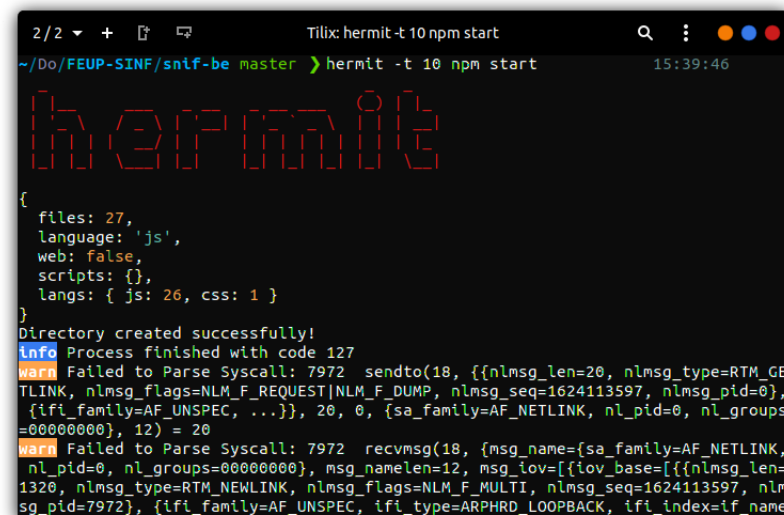
5.3 Hermit's Features

Throughout this section, we will describe Hermit's main features, which are the **dockerfile generation**, the **container execution** and the **two-stage dockerfile**.

5.3.1 Dockerfile Generation

This is Hermit's prominent feature and the basis of this master thesis. By calling Hermit in the root of the project's repository, the tool will start the containerization process. Therefore, it is required to pass the service's startup command as an argument, like this: "hermit <command>".

First, Hermit does a static analysis of the repository and prints the information obtained during this step in the console. After that, Hermit proceeds to the dynamic analysis.



```

2/2 + [?] [?] [?] Tilix: hermit -t 10 npm start 15:39:46
~/Do/FEUP-SINF/snif-be master > hermit -t 10 npm start

hermit

{
  files: 27,
  language: 'js',
  web: false,
  scripts: {},
  langs: { js: 26, css: 1 }
}
Directory created successfully!
Info Process finished with code 127
warn Failed to Parse Syscall: 7972 sendto(18, {{nlmsg_len=20, nlmsg_type=RTM_GE
TLINK, nlmsg_flags=NLM_F_REQUEST|NLM_F_DUMP, nlmsg_seq=1624113597, nlmsg_pid=0}},
{{ifi_family=AF_UNSPEC, ...}}, 20, 0, {{sa_family=AF_NETLINK, nl_pid=0, nl_groups
=00000000}}, 12) = 20
warn Failed to Parse Syscall: 7972 recvmsg(18, {{msg_name={{sa_family=AF_NETLINK,
nl_pid=0, nl_groups=00000000}}, msg_namelen=12, msg_iov={{iov_base={{nlmsg_len=
1320, nlmsg_type=RTM_NEWLINK, nlmsg_flags=NLM_F_MULTI, nlmsg_seq=1624113597, nlm
sg_pid=7972}}, {{ifi_family=AF_UNSPEC, ifi_type=ARPHRD_LOOPBACK, ifi_index=if_name

```

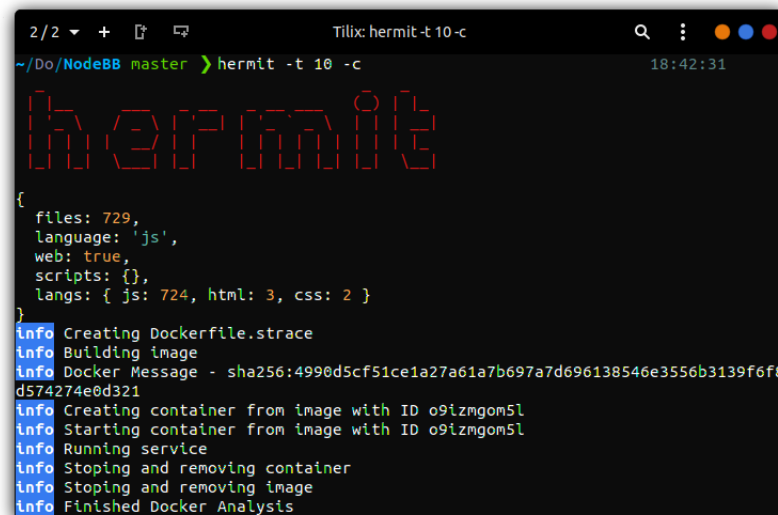
Figure 5.3: Hermit doing the project analysis

During the dynamic analysis step, the service runs until a timeout is exceeded. We are auxiliated by a software, known as **strace**, that attaches itself to the service and can track the system calls execution. The default value of the timeout is 60 seconds, but this value can be changed by passing the option "-t <number of seconds>" to Hermit. During the service's execution, the systems calls will all be printed in a file named **syscalls.log**. This file will be parsed and the system calls will be filtered to include only those of types open/openat, bind and exec/execve. If the parsing of system call fails, usually due to the lack of some data, a warning will be presented to the user. After this step is finished, the service and all the processes spawned by it are killed.

All the information collected in the previous step will be supplied to each of the data inference group modules. These modules will then produce each respective output. During the process of inferring the dependencies, after every package name is discovered, it is printed in the console.

those barriers, we can escalate our study and diversify the sample of projects, without losing time in the projects' setup. This feature can also be useful to suggest alternative dockerfiles to the original one.

To use this feature, it is required to activate the option "-c" and it is expected to exist a dockerfile in the root of the repository. In Figure 5.6, there is an example of this execution.



```
2/2 + [?] [?] Tilix: hermit -t 10 -c 18:42:31
~/Do/NodeBB master > hermit -t 10 -c

hermit

{
  files: 729,
  language: 'js',
  web: true,
  scripts: {},
  langs: { js: 724, html: 3, css: 2 }
}
Info Creating Dockerfile.strace
Info Building image
Info Docker Message - sha256:4990d5cf51ce1a27a61a7b697a7d696138546e3556b3139f6f8d574274e0d321
Info Creating container from image with ID o9izngom5l
Info Starting container from image with ID o9izngom5l
Info Running service
Info Stopping and removing container
Info Stopping and removing image
Info Finished Docker Analysis
```

Figure 5.6: Example of a container execution

In this scenario, the container already possesses an entrypoint, and for that reason, this argument is not necessary. However, if the container is expecting any argument, it can be passed through Hermit, and it will redirect the argument to the container.

To trace the system calls inside the container, we will generate a new dockerfile from the original one, but with some changes. The first change is the injection of a step to install **strace**, and after that, we attach the entrypoint to the tracing tool. This new dockerfile is named **dockerfile.strace** to avoid conflicts with the original one.

Now, through Docker API, we build the container from **dockerfile.strace** and run it until the timeout finishes. This time, the **syscalls.log** file is created inside of the container, so we bind the container's working directory to the local project's directory and it will appear in that directory. If the container does not have a defined working directory, we inject an instruction in **dockerfile.strace** to define one. After the timeout is exceeded, the container is destroyed.

With the system calls collected, the remaining steps of this feature are identical to those from the previous section. The only exception is that the new dockerfile is now named **dockerfile.hermit** to distinguish it from the original dockerfile.

5.3.3 Two-Stage Dockerfile

This last feature, just like the name says, generates multi-stage dockerfiles, with the first stage for building using a stable image and the second stage to run the service in a minimalist environment.

After the base image is specified in the second stage, there is a step that copies the source code and only the essential dependencies from the previous stage to the current one. However, it is still an experimental feature and still needs further investigation to complete it. To use, it requires the option "-m".

The following example shows how a multi-stage dockerfile for a javascript project would look like:

```
# Build Stage
FROM node:14 AS build-env
ADD . /app
WORKDIR /app

RUN npm ci --only=production

# Run Stage
FROM gcr.io/distroless/nodejs:14
COPY --from=build-env /app /app
WORKDIR /app

CMD ["src/index.js"]
```

Chapter 6

Empirical Study

In this chapter, we describe the experiment made to validate our strategy, using our prototype.

Different open-source projects with dockerfiles from GitHub were selected as test subjects for Hermit. Using our tool, we generated new dockerfiles for these projects and compared them with the original ones. For distinction, we will refer to the container of the original dockerfile as *original container* and the container of the automatically generated dockerfile as *hermit container*.

In the end, we made conclusions about the efficacy of our strategy using different metrics.

6.1 Study Goals

The goal of this study is to evaluate our approach and take conclusions about its feasibility and performance in real-world projects. This study is aligned with the research strategy we defined in Chapter 4 and its objective is to answer the research questions we previously defined on that chapter.

- **RQ1 - To what extent is it possible to detect all dependencies without adding bloat?** — The objective is to minimize the number of missed dependencies while avoiding as much bloat as possible, like unnecessary dependencies.
- **RQ2 - To what extent it is possible to detect the ports to expose without adding unnecessary risks to the service?** — We want to detect what ports must be exposed by the service, without creating new risks by adding unnecessary ports
- **RQ3 - To what extent it is possible to infer a suitable entrypoint?** — We want to infer an entrypoint identical or at least similar to the right one, maximizing the similarity.

6.2 Design

In Figure 6.1, there is an activity diagram representing the design of our empirical study and all of its stages. The first stage, **project sampling**, selects the projects to be used as test subjects to evaluate Hermit, the next stage, **dockerfile comparison**, executes those tests and the final stage,

data analysis processes the data that resulted from the tests, in order to evaluate how Hermit performs. An in-depth description of these stages' design can be found in the following sections. In Figure 6.1, there is a diagram representing the overall activities of our study and the artifacts that result from them.

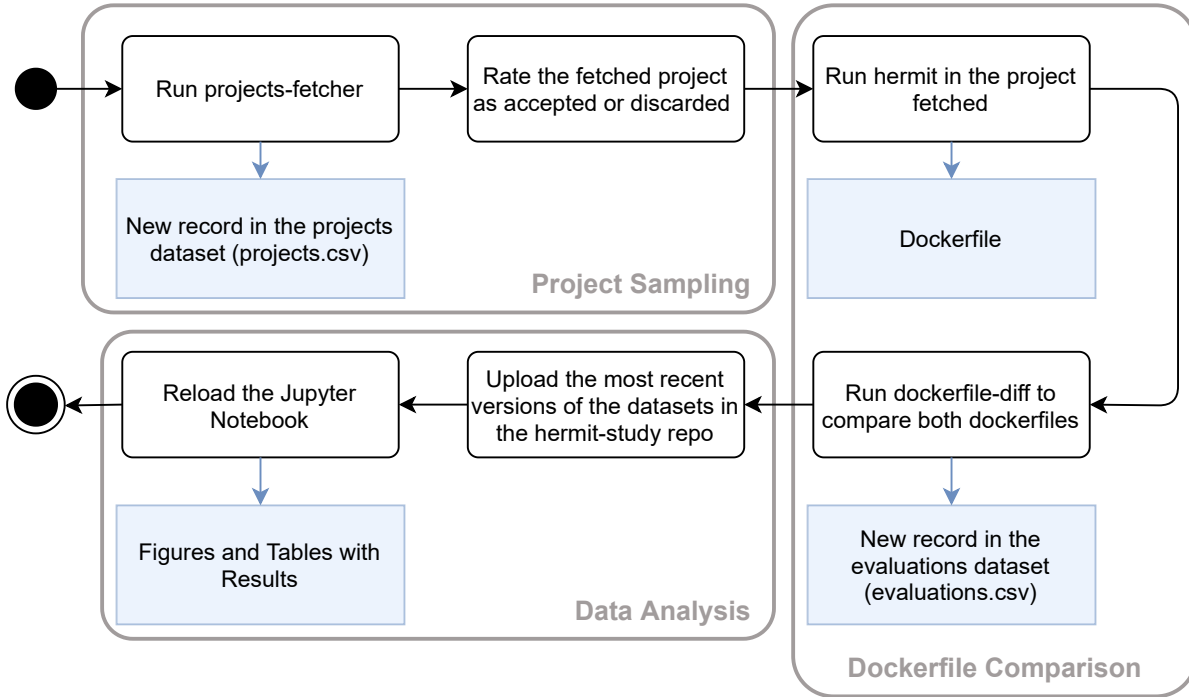


Figure 6.1: Study's activity diagram and artifacts

6.2.1 Study Variables

In this section, we identify the dependent variables.

The dependent variables are characteristics of the dockerfile generated by Hermit or in the container that results from said dockerfile. Characteristics like the **exposed ports**, the **installed dependencies** or the **entrypoint** and that we can compare with their equivalent ones present in the original dockerfile to calculate the metrics necessary to evaluate Hermit's performance during our study.

6.2.2 Project Sampling

This section describes the first stage of the study, which has the objective of randomly selecting from a population of open-source projects hosted on GitHub a sample of projects to apply our study. For this purpose, we developed a tool, which we name **projects-fetcher**, that creates a dataset of projects to be used as our test subjects, through queries to the GitHub API. In the query, we specify which programming languages we support, which at the moment are Python and JavaScript. Just like seen in the State of the Art (*cf.* Section 3.7), these two languages and Golang are the most used in projects with dockerfiles. We pseudo-randomly select one project from this

group of projects and check if any dockerfile exists in its repository. If there is a dockerfile, the project is selected and stored in a dataset which we name the **projects dataset**. If not, the process repeats until a project with a dockerfile is found.

The last task of this stage is to update the status of every entry of the projects dataset generated in the previous stage, with the information if they are *accepted* or *discarded*. A project is discarded when its dockerfile runs commands that install or/and compile dependencies through unorthodox methods unrelated to the package manager, otherwise, it is accepted. This task is not automated and must be done manually.

6.2.3 Dockerfile Comparison

In this stage, we evaluate the dockerfiles generated by Hermit, through comparison with the ones originally created by software developers for the same projects. This process is made using dockerfile-diff, which analyses the dockerfiles and the containers that result from them.

From the dockerfiles, the parsed data are the exposed ports (**EXPOSE**), the entrypoints (**ENTRYPOINT**) or commands for the execution of the service (**CMD**), their base images (**FROM**) and the installation steps that they run to configure the container (**RUN**).

From the containers, we can understand how the packages installed differ. These differences include the unnecessary packages that Hermit adds as bloat, the ones that failed to be detected, or what differs from the version of the packages used in the automatically generated dockerfile to the version used in the original one. This information cannot be extracted from the dockerfile directly, because it does not tell us what dependencies come with the base image, just like we cannot know the versions of all packages.

The data collected from these experiments are used to calculate different metrics to evaluate Hermit's dockerfiles. Then, they are stored in a second dataset, which we call the **evaluations dataset** to perform some statistical analysis in the following stage. These calculated metrics are:

- The **bloat ratio** (unnecessary dependencies) — Calculated by the division of the total sum of hermit container's exclusive packages by hermit container's total size.

$$bloat_ratio = \frac{\sum_{i=1}^{unnecessary_dependencies} dependency_i}{hermit_container}$$

- The **missed dependencies ratio** — Obtained by the division of the total sum of the original container's exclusive packages by its total size.

$$missed_dependencies_ratio = \frac{\sum_{i=1}^{missed_dependencies} dependency_i}{original_container}$$

- The **balance** from different versions of the same package — Difference of sizes (can be positive or negative) between packages used by both containers but in different versions.

$$balance = \sum_{i=1}^{hermit_dependencies \cap original_dependencies} (hermit_dependency_i - original_dependency_i)$$

- The **ports detection accuracy** rate ¹ — The percentage of ports exposed by hermit container that exist in the original container.

$$ports_detection_accuracy = \frac{\#detected_ports}{\#original_ports}$$

- The **risk percentage** — The ratio of exposed ports that don't exist in the original container.

$$risk_percentage = \frac{\#(hermit_ports - detected_ports)}{\#hermit_ports}$$

- The **entrypoint similarity** — Calculated the difference between the entrypoints supplied by both dockerfiles. The formula used to calculate the distance is the Sørensen–Dice coefficient.

In the end, we update the projects dataset and categorize the last entry as *successful*, *failed-build*, *require-extra-steps*. A project is rated as successful when the dockerfile generated by Hermit successfully builds the container, otherwise we rate it as failed-build. If we can do some steps of the project's installation manually, and leave the rest to the dockerfile, we can rate the project as require-extra-steps. One example of an extra step is compiling a TypeScript project to JavaScript.

6.2.4 Data Analysis

With the data collected in the projects dataset, we measure the overall performance of Hermit to generate dockerfiles. The stored data that resulted from the experiments, which are both the projects dataset and the evaluation dataset, are uploaded to the hermit-study repository and loaded in a code notebook, where the data is submitted to a statistical analysis.

First, we try to understand the distribution of the projects regarding their status, more specifically the **amount of projects discarded** and the **amount of projects accepted** for the experiment. Then considering only the accepted projects, we did another analysis, this time to compare the quantities of projects where Hermit succeeded in generating functional dockerfiles, with those where the dockerfiles failed, and with the projects that required additional manual steps to work.

The subsequent analysis is made to the evaluations database. In this analysis, we evaluate Hermit's results, using metrics like the bloat ratio, the missed dependencies ratio, the balance, the detected ports accuracy, the ports risk percentage and the entrypoint similarity. Then, we do a statistical analysis of every individual metric, which helps to understand Hermit's overall performance. These analyses include the mean values, the quartiles, the median, the standard

¹**Note:** This accuracy is only determined if the original container exposes any port.

deviation and the boxplot. With the values that result from this process, we can proceed to the discussion of our methodology. With these analyses, we can answer the research questions RQ1 (bloat, missed dependencies and balance), RQ2 (ports detection and risk) and RQ3 (entrypoint similarity).

6.2.5 Replication Package

In order to replicate all the experiments described in this chapter, we offer some instructions about the usage of the study's materials.

The materials developed to execute this study can be found in a general open-source repository named [Raidenkyu/automatic-service-containerization](https://github.com/Raidenkyu/automatic-service-containerization)². Inside this repository can be found the following repositories:

- [Raidenkyu/hermit](https://github.com/Raidenkyu/hermit)³ - The repository with the source-code of Hermit (*cf.* Chapter 5) and the instructions to install it.
- [Raidenkyu/dockerfile-diff](https://github.com/Raidenkyu/dockerfile-diff)⁴ - The repository with the source-code of dockerfile-diff (*cf.* Section 6.2.3) with the respective instructions for its installation and the current state of `evaluations.csv`.
- [Raidenkyu/projects-fetcher](https://github.com/Raidenkyu/projects-fetcher)⁵ - The repository with the source-code of projects-fetcher (*cf.* Section 6.2.2), the instructions to run it and the current version of `projects.csv`
- [Raidenkyu/hermit-study](https://github.com/Raidenkyu/hermit-study)⁶ - A repository with copies of the most updated versions or the datasets and a Jupyter Notebook analyzing them (*cf.* Section 6.2.4).

All the enumerated steps to replicate this study can be found in the Appendix A or in the general repository's README file.

6.3 A toolset for running the study

This section describes the design and implementation of the two tools developed to make this study possible. These tools are the **projects-fetcher**, which is vital for the projects sampling stage, and the **dockerfile-diff** to execute the dockerfile comparison.

6.3.1 Projects-fetcher

In order to have an accurate evaluation of Hermit's performance, we wanted a strategy that could guarantee that our sample of projects is the most diverse as possible. Also, We wanted a strategy

²<https://github.com/Raidenkyu/automatic-service-containerization>

³<https://github.com/Raidenkyu/hermit>

⁴<https://github.com/Raidenkyu/dockerfile-diff>

⁵<https://github.com/Raidenkyu/projects-fetcher>

⁶<https://github.com/Raidenkyu/hermit-study>

that could be fully reproducible by other researchers. That motivated us to develop a tool that could randomly obtain projects to fill our sample with projects from the most extensive open-source projects database, GitHub.

Our tool `projects-fetcher` takes advantage of the GitHub API to pick random projects made in Python or JavaScript and have dockerfiles. The fetched project's data is then stored in a file named **projects.csv**, which represents the projects dataset.

Due to the limit of requests imposed by GitHub for an hour, the tool only fetches one single project at every execution. Therefore, we run `projects-fetcher` every time we want to add another entry to the dataset, usually after we finished analyzing the previous entry. With this strategy, we are able to make good use of the limited requests, because the time spent between analyses is enough to "refill" our number of available requests.

GitHub API only allows us to search for repositories containing specific languages. However, if we want to filter only the projects that include dockerfiles, we must use another endpoint that searches inside a repository about specific files.

The diagram representing `projects-fetcher`'s activities can be found in Figure 6.2.

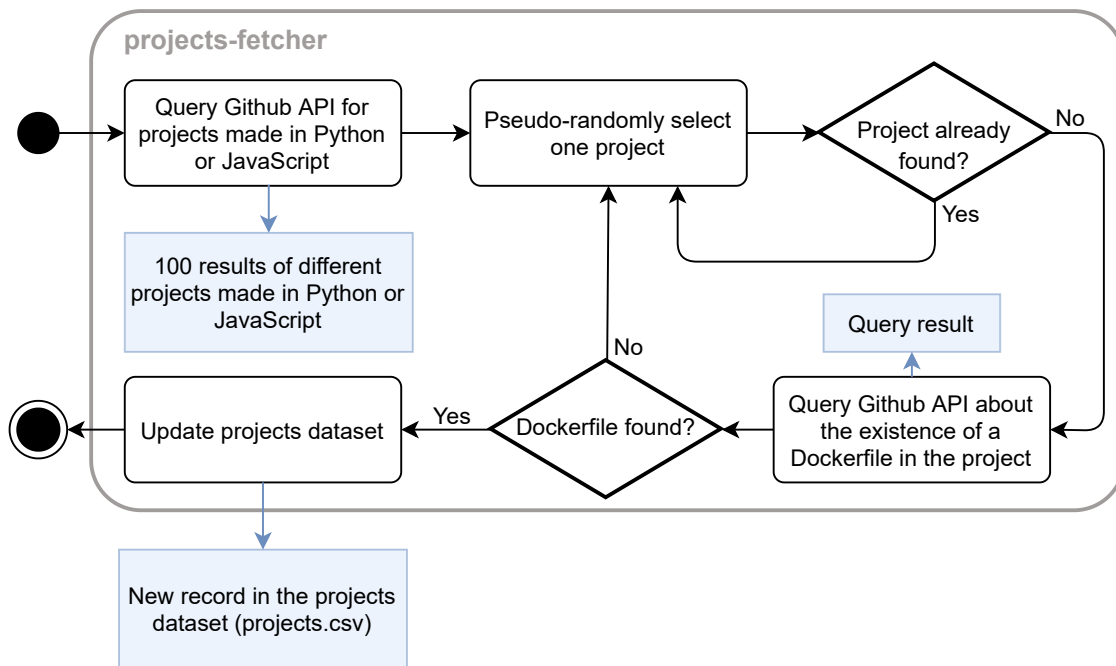


Figure 6.2: `projects-fetcher` activity diagram

First, we query the search API about all Python and JavaScript projects, which returns 8 million repositories. However, only 100 are randomly picked due to the limits of pagination. We will choose one project from this selection and query the search API if the said project includes a dockerfile. If one exists in the repository, the process finishes and the project is added to the projects dataset. If not, another project is randomly picked and the process repeats.

In order to make good use of the limited requests available during an hour, every analyzed project is stored in cache storage. Thus, before asking GitHub about the existence of a dockerfile,

projects-fetcher first checks if its data is included in the cache storage and will select another project until it finds a project that was not analyzed yet.

6.3.2 dockerfile-diff

For the comparison of dockerfiles, we sought a comparison strategy that could scale to a considerable amount of projects, as well as allow other researchers to reproduce our results. Therefore, we tried to automate the entire process as much as possible. This motivated us to develop a tool for this purpose, which is **dockerfile-diff**.

Figure 6.3 depicts the operation of the tool and other processes involved.

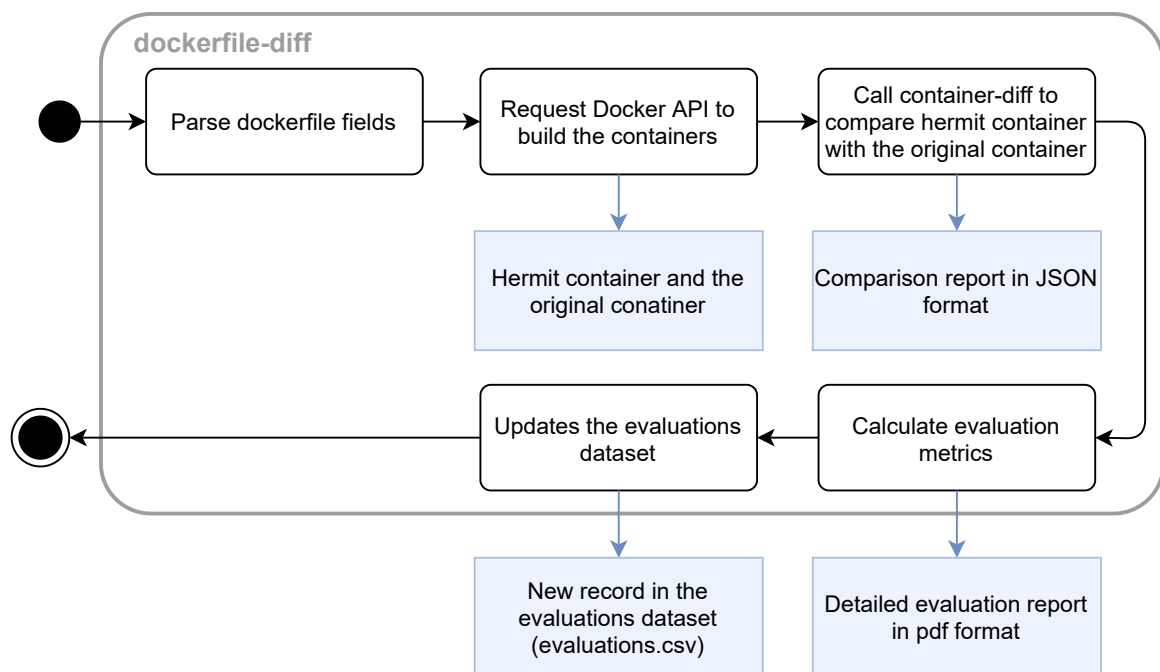


Figure 6.3: dockerfile-diff activity digaram

dockerfile-diff compares two dockerfiles and generates a report detailing their differences through a few metrics.

As explained in the previous section, the first data extracted from both dockerfiles comes from the parse of their fields like `EXPOSE`, `ENTRYPOINT`, `CMD`. With the first field, we can understand what ports the service exposes, and with the last two fields, we can discover the entrypoint to run the project.

However, to infer the packages contained in the containers generated by both dockerfiles, the best approach is to build the said containers and analyze them, because these dependencies came from both the base image or were installed using a package manager afterward. For those reasons, after finished the static analysis of the dockerfile, dockerfile-diff builds both containers, through the Docker API. Then, to compare both containers, our tool executes container-diff⁷, a tool that

⁷container-diff is a tool, developed by Google, to analyze and compare container images and examine them along with several different criteria like image size or the packages installed by apt, npm or pip

inspects containers and shows users the differences between them, what packages are exclusive to each container, their sizes and what versions of the same package both containers are using. After the comparison process finishes, the ancillary containers are destroyed. Thus, container-diff can compare the containers, while dockerfile-diff uses that comparison to calculate the evaluation metrics.

Finally, dockerfile-diff uses the collected data to calculate the metrics described in Section 6.2.3.

6.4 Results analysis and discussion

Using the methodology described in Section 6.2, we were able to obtain results that we will analyze and discuss, throughout this section. A total of 120 projects were sampled from GitHub, but only 77 projects had functional dockerfiles. From this group, 59 were accepted as test subjects for Hermit, and the tool was able to generate functional dockerfiles for 41 projects.

6.4.1 Projects acceptance

Due to Hermit's lack of support for all possible ways to install projects, we will only consider projects that depend on automated installation steps like commands to install dependencies, just like previously stated in Section 6.2.3. For that reason, we manually discard the projects collected by projects-fetcher that do not fill these criteria. The projects accepted were used by Hermit as test subjects. In Figure 6.4, we can see the distribution of projects that were accepted or discarded.

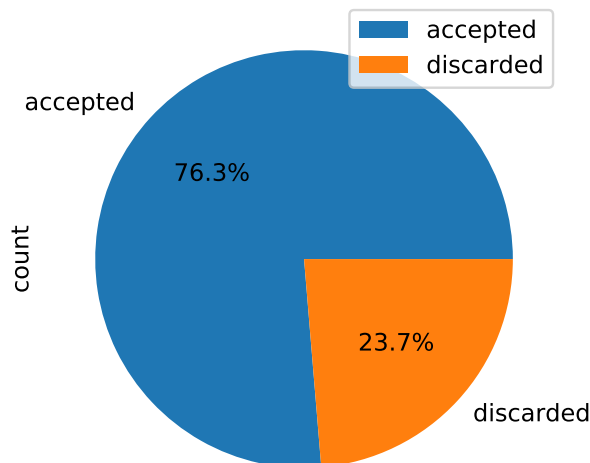


Figure 6.4: Results from all of the selected projects

As we can see, less than a quarter of the collected projects were discarded mainly due to the use of rare installation instructions. However, Hermit still shows limitations in its support of complex systems. For example, some of these systems presented intermediary steps to compile or

to download external dependencies that are not obtainable from the package manager, using git or curl.

6.4.2 Dockerfile Validation

After a project is accepted, it is submitted to the experiment, where we study if Hermit successfully generated a functional dockerfile (successful), generated a dockerfile that does not work properly (failed-build) or generated a dockerfile that only works after some additional tinkering in the project (require-extra-steps), but not in the dockerfile. We can see a distribution of those results in Figure 6.5.

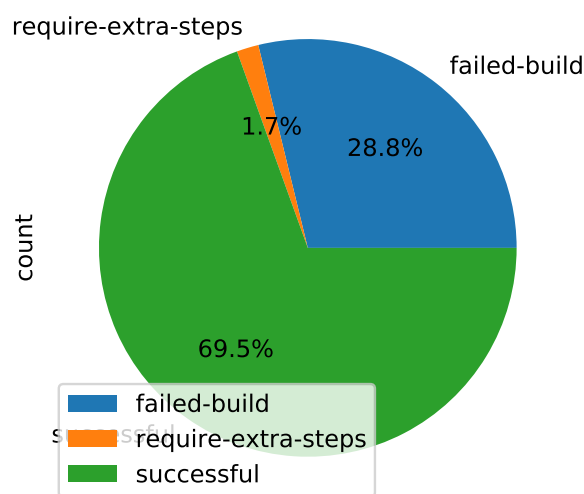


Figure 6.5: Results from all of the selected projects

With those results, we can confirm a success of our approach for 69,5% of the projects for which we have run Hermit

The failure of some dockerfiles, to build the respective container (28.8%), can be explained by the lack of the specific dependencies that are crucial to compile or build other dependencies installed by the package manager. Hermit does not detect these dependencies because they are used at build time and not at run time. As we know, Hermit's dynamic analysis only operates during run time.

The remaining 1.7% require extra steps to work, which means that with additional actions done manually in the repository, the dockerfile generated by Hermit will work just like any successful evaluation. This only projects, in our experiments, were typescript projects, that we manually compiled into JavaScript code, which Hermit's dockerfile can now handle on its own.

6.4.3 Dockerfile Evaluation

Now considering only the projects where dockerfiles generated by Hermit build successfully, we do a statistical analysis and take conclusions of Hermit's current ability to automatically generate

dockerfiles. The projects submitted for evaluation correspond to 54.3% of the original sample (69.5% successful and 1.7% require-extra-steps of the 76.3% accepted)

We split our analysis into three sections, where each one groups the relevant analysis to answer one particular research question. In the first section, we evaluate the detected dependencies, the ones that were missed, how they differ and what bloat was added. The second section is focused on the network ports detected, including which ones were missed and what risk was added. The last section evaluates the entrypoints inferred.

6.4.3.1 Dependencies Evaluation

To study Hermit's capacity to infer the dependencies, the metrics that we want are the bloat ratio, the missed dependencies ratio and the balance.

By doing a statistical analysis of the different evaluations got from the bloat ratio, we obtained the values listed in Table 6.1.

Metric	Value
Mean Bloat Ratio	43.2%
Minimum Value	0
Lower Fence	0
Quartile 1	0
Quartile 2 \ Median	43.4
Quartile 3	79.6
Upper Fence	97.4
Maximum Value	97.4
Interquartile Range	79.6
Standard Deviation	37.8

Table 6.1: Hermit's dockerfiles dependencies evaluation

We find that these dockerfiles have an average bloat ratio of 43.2%. This means that almost half of the average hermit container are unnecessary dependencies not present in the original dockerfiles. In the best scenarios, the containers are free of bloat, but in the worst cases, more than 90% of the container is bloat. A considerable amount of dockerfiles (from the minimum value to quartile 1) could present no bloat, showing that it is possible to avoid bloated containers. At the same time, there are some containers heavily bloated (From quartile 3 to the maximum value).

To understand the variance of bloat ratios obtained, we can see in Figure 6.6, the boxplot of the collected bloat ratios.

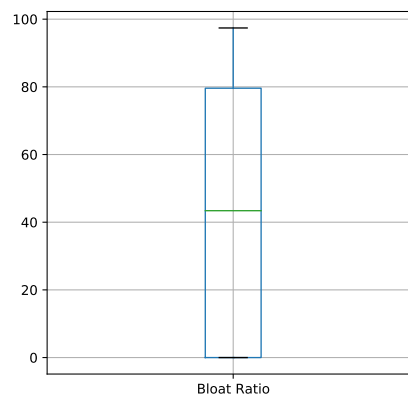


Figure 6.6: Bloat Ratio boxplot

Looking at the bloat ratio boxplot, we can see that many Hermit containers add a massive bloat compared to their original counterparts, and some have more bloat than necessary dependencies (bloat ratio higher than 50%). Just like we saw previously, in the worst case, almost 100% of the container is bloat. The median (43.4%) is close to the mean value, showing a symmetrical distribution around that value.

The majority of the unnecessary dependencies come with the base image, which is a consequence of choosing Debian-based images due to their stability. The original dockerfiles, in some cases, use Alpine-based images, which are very popular for minimal containers and come with minimal dependencies. However, there is a particular risk regarding the container's functionality and behavior. This risk results from the differences regarding some libraries' implementation, like `libc`. We can only conclude that the distroless images could be the ideal middle-term, because they are Debian-based images stripped from the majority of their dependencies.

After we studied the bloated dependencies, we wanted to know how many dependencies Hermit failed to detect from the original container. So, just like with the bloat ratio, we did a statistical analysis of the different missed dependencies ratios. The results obtained can be seen in Table 6.2.

Metric	Value
Mean Missed Dependencies Ratio	22.3%
Minimum Value	0
Lower Fence	0
Quartile 1	0
Quartile 2 \ Median	1.2
Quartile 3	43.9
Upper Fence	84.7
Maximum Value	84.7
Interquartile Range	44.0
Standard Deviation	28.8

Table 6.2: Hermit's dockerfiles dependencies evaluation

We can see that Hermit also misses 22.3% of the dependencies, which we believe that can be explained by the limited workflows simulated during the dynamic analysis, resulting in some dependencies not being called during the service's execution. The lower value of the median shows us that there is a considerable amount of projects where Hermit did not miss a single dependency.

We need to consider that our baseline to evaluate the missing dependencies is the original dockerfile. If that one is already bloated, then those dependencies are not missing but cleaned from the original container. However, in this master thesis, we will assume that every dependency used by the original container is indeed useful.

The analysis of the missed dependencies ratios' distribution can be made through the boxplot in Figure 6.7.

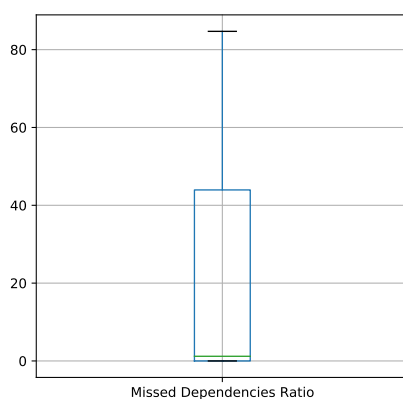


Figure 6.7: Missed Dependencies Ratio boxplot

That distribution confirms once more that the majority of the dockerfiles generated by Hermit miss at most 10% of the dependencies. However, many dockerfiles miss more than 40% of the dependencies, and in the worst case, it missed more than 80%. In these last cases, despite missing many dependencies, the containers were still appropriately built, meaning that the original containers were bloated.

Finally, we studied the balance between the versions of the packages used in the original containers to the versions used in Hermit's containers. The results from the statistical values made to these values of the balance are in Table 6.3.

Metric	Value
Mean Balance	5.5 MB
Minimum Value	-9.5
Lower Fence	-1.2
Quartile 1	0
Quartile 2 \ Median	0
Quartile 3	0.8
Upper Fence	2.0
Maximum Value	56.1
Interquartile Range	0.8
Standard Deviation	14.1

Table 6.3: Hermit's dockerfiles balance evaluation

With these results, we can conclude that most of the dockerfiles generated by Hermit have dependencies with a size comparable to those of the original dockerfiles. When a difference does occur, it is usually at the scale of megabytes. The balance can be positive or negative, but the first situation is much more common than the latter. A positive balance results from the versions of the packages used by Hermit, which usually are the latest ones. This analysis adds another important conclusion to answer the first research question.

For more details regarding the distribution of the different balances, we can look at Figure 6.6.

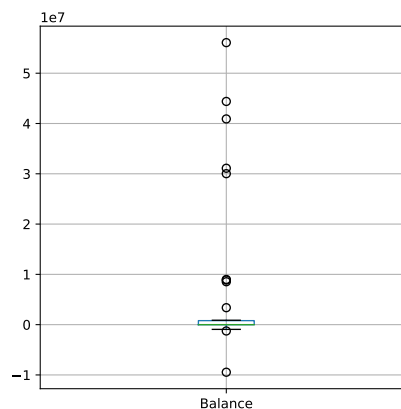


Figure 6.8: Balance boxplot

The balance's boxplot is the most peculiar one, due to presenting more outliers than the previous ones. The values represented in the figure, confirm what we saw in Table 6.3.

The majority of the dockerfiles have a value of balance near of zero, showing that the difference of versions is not a significant issue. However, there are some exceptions represented by the outliers, and in the worst case, the balance can be more than 50 MB.

With the analysis we made throughout this section, we think that we have enough data and conclusions to answer the research question "RQ1 - To what extent it is possible to detect all

dependencies without adding bloat?". To answer this question, it is essential to understand what bloat was added to the containers, what dependencies were missed, and how the dependencies differ from the original container to the container generated by Hermit, which was the basis of the analysis made in this chapter. Therefore, in Section 6.5.1, we will give our answer to the referred research question and discuss it.

6.4.3.2 Ports Evaluation

In this section, we focus our discussion on Hermit's ability to detect the ports that must be exposed by evaluating its ability to detect all the effectively needed ports, no more and no less.

The statistical analysis of both ports detection accuracy and risk percentage presented promising results, which can be seen respectively in Table 6.4 and Table 6.5

Metric	Value
Mean Ports Detection Accuracy	100%
Minimum Value	100
Lower Fence	100
Quartile 1	100
Quartile 2 \ Median	100
Quartile 3	100
Upper Fence	100
Maximum Value	100
Interquartile Range	0
Standard Deviation	0

Table 6.4: Ports evaluation

Metric	Value
Mean Risk Percentage	0%
Minimum Value	0
Lower Fence	0
Quartile 1	0
Quartile 2 \ Median	0
Quartile 3	0
Upper Fence	0
Maximum Value	0
Interquartile Range	0
Standard Deviation	0

Table 6.5: Risk evaluation

The detection of ports and the potential exposition of unnecessary ports got the best evaluation possible, confirming the success of our approach to extract the used ports from the system call **bind**, which is the system call used by a process when requesting a port. Both have uniform results with a null standard deviation.

To confirm this good results and that there were not outliers, we can look at their boxplots. Figure 6.9 represents the ports detection accuracy boxplot and Figure 6.10 shows us the risk percentage boxplot.

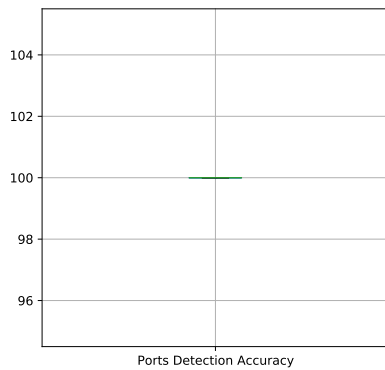


Figure 6.9: Ports Detection Accuracy boxplot

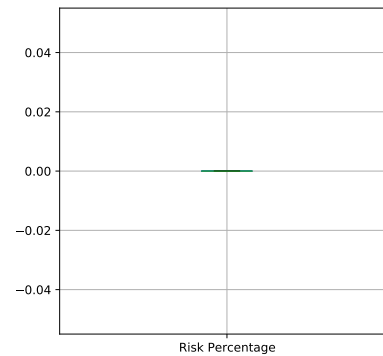


Figure 6.10: Risk Percentage boxplot

The uniform distribution of the two metrics about the ports proved, once again, the success of tracking the ports from the system calls. The first boxplot confirms a fixed value of 100% in the ports detection accuracy and a fixed value of 0% in the risk percentage, with no outliers detected in both cases.

Those promising results are essential to formulate our answer to the research question "RQ2 - o what extent it is possible to detect the ports to expose without adding unnecessary risks to the service?". In Section 6.5.2, this answer will be discussed.

6.4.3.3 Entrypoint Evaluation

Finally to evaluate the entrypoint, the metric used was the entrypoint similarity. The results obtained from the statistical analysis can be found in Table 6.6.

Metric	Value
Mean Entrypoint Similarity	72.6%
Minimum Value	0
Lower Fence	70.6
Quartile 1	70.6
Quartile 2 \ Median	100
Quartile 3	100
Upper Fence	100
Maximum Value	100
Interquartile Range	29.4
Standard Deviation	39.2

Table 6.6: Hermit's dockerfiles entrypoints evaluation

The last metric, entrypoint similarity, shows a promising mean value of 72.6% and a median of 100, proving the efficacy of Hermit to detect the entrypoints.

However, we think that the evaluation method needs to be improved. In specific cases where both dockerfiles presented different entrypoints, but had the same effect in the service, the metric

would give an unjust low evaluation. For example, in certain Node.js projects, Hermit inferred that the entrypoint was `npm start`, but the original dockerfile's entrypoint was the command defined in the start script. Moreover, we concluded that those scenarios interfered with the validation of the approach.

To understand better the quantity of these specific scenarios, we can analyze the boxplot in Figure 6.11 and look for outliers.

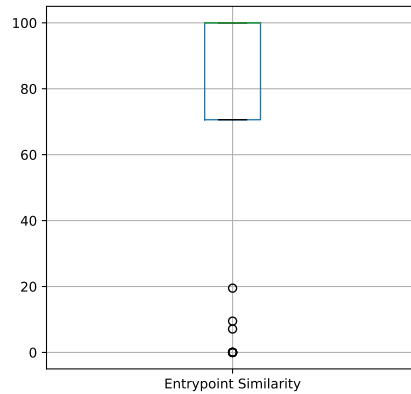


Figure 6.11: Entrypoint Similarity boxplot

Looking at the entrypoint's boxplot, we can confirm Hermit's capability to infer said data. However, we can also notice some outliers representing rare scenarios where different entrypoints have the same functionality.

The conclusions discussed in this section help us to answer the research question "RQ3 - To what extent it is possible to infer a suitable entrypoint?", which can be found in section 6.5.3.

6.5 Answering the Research Questions

After analyzing and discussing the results obtained from this study, we are now able to answer the questions we referenced in Section 6.1.

Due to the random sampling, our final answers end up being the most generic as possible to handle different kinds of projects, whatever their complexity is. Thus, the projects collected can vary from simple micro-services to complex monolithic systems. It is crucial to notice that the topics we focused on in our study can only handle projects from low complexity to medium complexity. In contrast, projects with higher complexity require dockerfiles less generic and more specific.

Returning to the questions, we formulated, our proposals to answer them can be found in this section.

6.5.1 RQ1 - To what extent it is possible to detect all dependencies without adding bloat?

We can detect all the dependencies used during run time (and only those), but some bloat is inevitable to ensure the container's stability.

We found that the majority of the dependencies used by the service in run time were not missed. This makes us believe that an effective approach for finding dependencies can indeed be achieved by detecting which files are used by the service (through the system calls **open** or **openat**) and then querying the system's package manager to find what are the relevant packages. However, we could not detect dependencies used for build, because the service did not call them during run time. Due to current support of two scripting languages, the majority of the dependencies are scripts or precompiled binaries. However, for the minority that is compiled locally, we include in the dependencies list some build tools like gcc, python-dev or cmake. These tools are the most used to compile dependencies, but this is not enough in some cases.

Regarding the bloat, it comes with the base image, so we believe that smaller images should reduce it. Nevertheless, the use of minimalist images may compromise the stability of the container. With stability, we mean the service being able to work correctly inside the container built with Hermit's dockerfile, just like it worked before the containerization or with the original dockerfile. Therefore, we concluded that between stability and minimalism, the first one should be prioritized, and at the moment the bloat is inevitable.

6.5.2 RQ2 - To what extent it is possible to detect the ports to expose without adding unnecessary risks to the service?

The excellent evaluations we obtained from the Hermit's dockerfiles regarding the detection of ports confirm with confidence that all the ports used by the service can be extracted from the system call **bind** and no unnecessary and risky port is added by accident.

Despite all the success we had in our experiments, we should consider that in rare situations where the ports are allocated during a specific workflow that was not executed during the dynamic analysis, Hermit will fail to detect it. This is not a critical issue, because the significant majority of the projects allocate all the necessary ports, when they startup, but exceptions may happen.

6.5.3 RQ3 - To what extent it is possible to infer a suitable entrypoint?

Despite the method of evaluation not being the ideal one, it still gives us some confidence that the entrypoint can be inferred from the system calls **exec** or **execve** or from specific patterns of the language, like the start script in the case of Node.js projects.

6.6 Comparing against the state of the art

In the review of the state of the art (*cf.* Chapter 3), we presented some solutions proposed by other researchers that are relevant for our own work.

On our literature review about minimalist images, we concluded that by starting with a lightweight image, like a distroless image, and adding all the necessary dependencies, we could have functional containers. These images lack package manager, which is needed to easily install any required dependencies. The solution for this is usually to use a multi-stage dockerfile, with a first stage to install the dependencies, and then copy them to the stage based on the distroless image. Unfortunately, when we started working with distroless images, we came to the conclusion that this approach is not reliable enough and only works for simple services. At the moment, the support provided by Hermit to this approach only works for a minority of projects, and requires more investigation to be reliable enough. So, in the end, we stayed with generic images, with package managers, despite having more bloat. For this reason, we think that the existing works and hermit still fail to achieve true minimalism in stable containers.

To infer the dependencies, we used the dynamic analysis to detect the system dependencies used by the service, and the static analysis to infer Python dependencies, indirectly through tools like `pipreqs` or `pigar`. These approaches were based in solutions proposed in the existing works [11]. However, the dependencies used during build time can't be detected by our approach and by the other works, because they are not used by the project's source code, but during the installation of dependencies of the language.

For the inference of the ports, we previously concluded that existing approaches using dynamic analysis are more effective than its static counterparts. In our work, the dynamic analysis proved to be more useful.

Currently, there are some proposals to create automatic containers, but they have not been embodied into concrete solutions [28, 13]. There is also one proprietary prototype that is dependent on virtual machines [16]. These works are not available for empirical comparisons and every data regarding their efficacy is scarce. In contrast, all of the materials we developed are openly available to the community (*cf.* Section 6.2.5). Compared to the current state, Hermit is the first freely-available concrete approach to solve this problem without depending on VMs, using dynamic and static analysis exclusively.

Not just Hermit, but the tools to compare dockerfiles and evaluate the automatically generated ones are freely-available, while offering a complete evaluation to the dockerfiles. We formulated our own approaches, due to the lack of established strategies to evaluate tools with the purpose of generating dockerfiles. Existing works don't offer concrete evaluation strategies and metrics strategies, which makes a comparison of our results with theirs very difficult.

6.7 Validation's threats

During the development of this study some threats were identified, which will be explained in this section.

- **Threat: Human judgment dependency** - The first threat happens during the differentiation of discarded projects and accepted projects, because this step is not automated and despite having a concrete criteria is still arbitrated by a human factor.

Mitigation - Definition of concrete and objective criteria to minimize the human factor as possible. The rule to discard a project is that its dockerfile must include only installation steps that depend from the package manager.

- **Threat: Possibly biased project selection** - The second threat is the lack of true randomness of projects-fetcher to obtain the projects to experiment with. We contacted GitHub Support, and they clarified that the order of the results is not indeed random, being influenced by activity data. Without a random sample, we can't be certain that our results generalize to all projects on GitHub.

Mitigation - We believe that if there is a bias, it is towards projects with more activity. We also, expanded the pagination of results to 100 (highest value supported) and choose one project randomly from the results.

- **Threat: Docker security protections** - The last threat identified results from an unexpected behavior of `strace` running in a docker container, which makes the tool not detect the system call to bind the ports. This results from a security mechanism to limit the system calls logged inside of a container. This makes the tracking process less natural

Mitigation - To handle this issue, we ask the original container which ports does it expose and inject fake syscalls containing this information.

Chapter 7

Conclusions

The automatic generation of dockerfiles would benefit developers, who would spend less time in writing them, because many developer use a trial and error approach to write them. Sometimes, due to the lack of time or experience, the resultant containers present obsolete dependencies and vulnerabilities, so we can also avoid those scenarios with the automation, and optimize the containers.

The use of containers, especially docker containers, is fairly recent, which means that there is still room for many improvements a contributions. There are already some research on different ways to optimize a dockerfile, however the automatic generation of these files, using only the analysis of the software, lacks concrete solutions, and only has a few empirical studies [16].

The existing solutions to generate dockerfiles, at the moment, rely on more resources than just the inspection of the software intended to be containerized. For example some works use a Virtual Machine to automatically generate the container and write its respective dockerfile, or only generate dockerfiles for simple code snippets.

With this dissertation, we believe that it is possible to generate dockerfiles, using only static and dynamic analysis to infer the necessary data to write it. This is supported by the different heuristics, based in the related work, to infer different aspects important in dockerfiles. With this inference, it is possible to fill the multiple fields of a dockerfile.

We made our contributions to this field with interesting conclusions that we think other researchers will benefit. With this, we are one step closer of a reliable and stable process to automatic containerize all kinds of services.

7.1 Main Contributions

These are the contributions that we referred in the conclusions:

- **Literature Review about Dockerfiles and inference techniques** — A literature research about different investigations with dockerfiles, that exist in the state of the art. Different

works related to dockerfiles were analyzed, and their conclusions will be discussed. This includes an in-depth study about what characterizes a docker container and its respective dockerfile; the different approaches to optimize a dockerfile, such as minimalism or security; the common issues present in dockerfiles developed by humans, and other improvements with promising ideas.

- **Novel approach** — The primary objective of this dissertation is to formulate a strategy to automatically generate dockerfiles using exclusively analysis of the software intended to be containerized. We proposed an original approach inspired by different techniques to infer the multiple fields that are part of a dockerfile, using exclusively dynamic analysis and static analysis. These goals are the base image, the dependencies but not bloat, the network ports and the entrypoint. However, when the projects become more complex and require less generic dockerfiles, the limitations of our approach appear. Despite this, it is still a significant contribution, which could be improved and expanded in future work. For this reason, the approach designed to be highly modular in order for other researchers to easily replace any module by any alternative developed by them.
- **Prototype** — The novel approach referred to above was the basis behind a concrete prototype tool, named **Hermit**. The tool contributes as a reference implementation, and its source code will be open-source and hosted on Github for everyone who wishes to see or contribute. With this tool we can conduct empirical studies to test the heuristic behind it. Hermit reflects the modular design of the approach, so that other researchers can replace its plug-able modules by their own implementations.
- **Empirical study design** — The last contribution is the formulation of an empirical experiment comparing dockerfiles written by developers to open-source projects with dockerfiles automatically generated by the tool referred above or by other tools with different heuristics. This includes two tools developed specifically for this purpose, which are a random selector of projects in Github (projects-fetcher) and a tool to evaluate an automatically generated dockerfile, using the original dockerfile as a baseline (dockerfile-diff). The experiments measures the bloat added to the container, the missed dependencies, the detected ports, the risk added through exposure of unnecessary ports and the entrypoint of the dockerfile automatically generated and its resultant container.
- **Empirical study replication** — A demonstration of the study referred in the previous bullet point and the respective results obtained.

7.2 Future Work

Like we said, our research added another step to make us closer from automatic containerization. But, there are other steps we think that could be explored afterwards, regarding the prototype, the empirical study or other additional studies.

7.2.1 Prototype Upgrades

- **Automatic generation of security profiles** — Not just generate dockerfiles, but seccomp profiles too. This file limits some operations inside the container to strengthen its security. For example, limits the system calls authorized.
- **Automatic generation of stable and optimized distroless base images** — Complete and stabilize the experimental feature that generates multi-stage dockerfiles. Our proposal is to use the bazel, the build system used to build the distroless images, to build specific, optimized and suitable base images to each service.
- **Add more iterations to the dockerfile generation process** — It would be interesting to explore the possibility of testing the recently generated dockerfile, study the resultant container, receive its feedback and apply the respective improvements based in the feedback. This improvement is based in a recent work authored by Henkel [10].
- **Work on an integration with Dockerlive [19]** — Dockerlive is a tool that helps developers to write dockerfiles. We believe that both tools would benefit if they were integrated together in a single tool that automatically generates dockerfiles and helps developers to expand on it. Hermit could generate a more basic dockerfile, where Dockerlive could expand to handle more complex and non-generic projects ¹. One of the reasons, that justifies this integration, is the constant changes that existing dockerfiles are regularly submitted to [26, 27].
- **Improve automatically generated dockerfiles' complexity** — Expand the current approach in order to handle more complex services. This includes adding strategies that could infer installation instructions that currently we can't support. For example, installation of dependencies that can't be installed with the package manager.
- **Support more languages** — Work to include strategies to containerize other languages popular with docker like Golang or Java.

7.2.2 Empirical Study Expansions

- **Study Hermit's dockerfiles security** — It would be interesting to study if Hermit's dockerfiles have more or less security than the original dockerfiles. Our suggestion to do that is to fetch the CVEs associated with the dependencies existent in both containers. With this information, we can evaluate what potential exploits Hermit's dockerfiles add or remove in comparison to the original dockerfiles.
- **Poll developers about their experience with Hermit** — Poll different developers about their opinions of the Hermit when compared to a manual generation of dockerfiles. Some questions could be :

¹<https://marketplace.visualstudio.com/items?itemName=david-reis.dockerlive>

- What was the success of the tool in their projects?
- What properties the dockerfile where missed or are unnecessary?
- How they think that the tool improved their workflow?

Appendix A

Empirical Study

The instructions necessary to install and replicate our study, step by step, can be found here:

A.1 Study Replication Steps

1. Download the main repository and its subprojects with:

```
git clone --recurse-submodules git@github.com:Raidenkyu/automatic-service-  
containerization.git
```

2. change to the `hermit/` directory and run:

```
npm run install-hermit
```

3. change to the `dockerfile-diff/` directory and run:

```
npm run install-dockerfile-diff
```

4. To get a random project from GitHub, change to the `projects-fetcher/` directory and run:

```
npm start
```

If you receive an error saying that the limit of requests was exceeded, please wait some minutes before retrying. If the operation succeeds you can find a project in `projects.csv`.

5. Navigate to the url of the project fetched in the previous step and clone the the respective project.
6. Change to the directory of the project and see if the original dockerfile of the project is too complex to hermit or not. Then rate the status of the project in the projects dataset as accepted or discarded. If the project was discarded, you don't need to follow the remaining steps.

7. Run hermit to generate the dockerfile with:

```
hermit -c -t <number of seconds to timeout>
```

Usually 10 seconds are enough.

8. Now you should have both the original "Dockerfile" and "Dockerfile.hermit". To compare them and evaluate hermit's dockerfile run:

```
dockerfile-diff Dockerfile Dockerfile.hermit
```

If evaluation process ended without problems then update the "generation" column in the file `projects.csv` with "successful". If not update with "failed-build", but if you think that by running some installation commands it will work update with "require-extra-steps". In the case of "failed-build" then you can ignore the following instructions.

9. To generate the file `evaluations.csv` the results collected by `dockerfile-diff` run:

```
dockerfile-diff print
```

10. Change to the `hermit-study/` directory and update the files of both datasets, in the `res/` directory with the versions that resulted from this experiment.
11. Open `hermit.ipynb` in a Jupyter Notebook and analyse how the evaluations changed.

References

- [1] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.
- [2] Meriem Belguidoum and Fabien Dagnat. Dependency management in software component deployment. *Electronic Notes in theoretical computer science*, 182:17–32, 2007.
- [3] Kyle Brown, Bobby Woolf, Cees De Groot, Chris Hay, Joseph Yoder, and Paulo Merson. Patterns for developers and architects building for the cloud, 2021. [Online, accessed on 2021-07-01].
- [4] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.
- [5] Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. Fogdocker: Start container now, fetch image later. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 51–59, 2019.
- [6] Jessica Forde, Tim Head, Chris Holdgraf, Yuvi Panda, Gladys Nalvarete, Benjamin Ragan-Kelley, and Erik Sundell. Reproducible research environments with repo2docker. In *ICML 2018 - RML Workshop*, 2018.
- [7] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confiner: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 443–458, 2020.
- [8] Katharina Gschwind, Constantin Adam, Sastry Duri, Shripad Nadgowda, and Maja Vukovic. Optimizing service delivery with minimal runtimes. In *International Conference on Service-Oriented Computing*, pages 384–387. Springer, 2017.
- [9] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. Rudsea: recommending updates of dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 796–801, 2018.
- [10] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d’Amorim, and Thomas W. Reps. Shipwright: A human-in-the-loop system for dockerfile repair. *CoRR*, abs/2103.02591, 2021.
- [11] Eric Horton and Chris Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *arXiv*, volume abs/1905.11127, 2019.
- [12] D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5, 2016.

- [13] Yash Kumar, Tejesh Mehta, Sundip Patel, Praful Rana, Casey Flynn, Nathan Allen Meyers, and Tony Li. Automatic container definition, June 14 2016. US Patent 9,367,305.
- [14] Eric Lubin et al. *VM2Docker: automating the conversion from virtual machine to docker container*. PhD thesis, Massachusetts Institute of Technology, 2015.
- [15] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
- [16] Vineet Hareesh Palan. A novel approach to containerize existing applications. Master’s thesis, Florida Institute of Technology, 2018.
- [17] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486, 2017.
- [18] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 51–56, 2017.
- [19] David Reis. Live docker containers. Master’s thesis, Faculdade de Engenharia da Universidade do Porto, 2020.
- [20] RightScale. 2019 State of the Cloud Report, 2019.
- [21] Deepika Saxena and Navneet Sharma. Docker security analysis mechanism with linux platform. In Vijay Singh Rathore, Nilanjan Dey, Vincenzo Piuri, Rosalina Babo, Zdzislaw Polkowski, and João Manuel R. S. Tavares, editors, *Rising Threats in Expert Applications and Solutions*, pages 595–601, Singapore, 2021. Springer Singapore.
- [22] Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 2021.
- [23] Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Engineering software for the cloud: Patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, SugarLoafPLOP ’16, USA, 2016. The Hillside Group.
- [24] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a pattern language for engineering software for the cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs*, PLoP ’18, USA, 2018. The Hillside Group.
- [25] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139, 2020.
- [26] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Dockerfile changes in practice: A large-scale empirical study of 4,110 projects on github. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 247–256, 2020.
- [27] Yang Zhang, Huaimin Wang, and Vladimir Filkov. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences*, 62(1):1–3, 2019.

- [28] SONG Zhuo, LI Gen, Xu Zhou, MA Chouxian, XIE Chenglong, Kan Wu, Zhaohui Sun, XU Xiali, YI Chungen, YANG Yao, et al. Container dockerfile and container mirror image quick generation methods and systems, September 17 2020. US Patent App. 16/618,402.