Analysing and deploying microservice-based applications

PhD Thesis

By

DAVIDE NERI

UNIVERSITY OF PISA
DEPARTMENT OF COMPUTER SCIENCE
DOTTORATO DI RICERCA IN INFORMATICA
XXXII CICLO

SUPERVISORS:

Prof. Antonio Brogi
Dr. Jacopo Soldani

A.Y. 2018/2019

Abstract

The overall objective of the thesis is to support an effective analysis of microservice-based applications and the automation of their deployment over container-based platforms. The main contributions of the thesis are (i) a methodology for automating the detection of architectural smells possibly violating some main principle of microservices, and for resolving such smells via architectural refactorings, (ii) a technique for completing an application specification by automatically discovering Docker-based runtime environments capable of supporting the application components, and (iii) an architectural approach for effectively deploying microservice-based applications on top of existing container orchestrators, by also allowing to manage the lifecycle of microservices independently of the lifecycle of the containers hosting them. All the aforementioned solutions have been implemented into running prototypes and tested on concrete case studies, involving third-party software solutions.

TABLE OF CONTENTS

List of Tables

List of Figures

Part I

Opening

CHAPTER 1

INTRODUCTION

Cloud computing permits running on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [10]. This has revolutionised the way enterprise applications are built by the IT industry, where monoliths are giving way to distributed, service-based architectures. Microservice-based architectures are indeed increasingly considered an enabling approach to shorten the lead time in software development and to effectively scale software application deployments [108, 121]. The interest in microservice-based architectures is witnessed by their adoption by the major IT companies (e.g., Amazon, Facebook, Google, Netflix and Spotify).

Microservices are an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, e.g., HTTP resource API [108]. Microservice-based architectures can be seen as service-oriented architectures that satisfy some key principles [169]. These include shaping services around business concepts, adopting a culture of automation, decentralising all aspects of microservices (from governance to data management), ensuring their independent deployability and scalability, and isolating failures [121]. Given that microservices have solid module boundaries, different services can be written in different programming languages and they can also be managed by different teams of developers [108].

Microservices are typically deployed in the cloud by exploiting container-based virtualisation [24, 128]. Container-based virtualisation has gained significant acceptance, because it provides a lightweight solution for running multiple isolated user-space instances (called *containers*). Such instances are particularly suited to package, deploy and manage service-based applications [14]. Developers can bundle each service along with the dependencies they need to run in isolated containers and execute them on top of a container run time (e.g., Docker [58],

Rkt [50], Dynos [87]).

Container orchestrators (e.g., Docker Compose [63], Kubernetes [157]) allow to automate the deployment and management of containers. They indeed permit scaling, discovering, load-balancing and interconnecting containers over clusters of hosts. Compared to other existing virtualisation approaches, like virtual machines, containers feature faster start-up times and less overhead [98, 102, 110].

Microservice-based applications can be composed of hundreds of interacting microservices. As the number of microservices in an architecture increases, designing a microservice-based application and checking whether it adheres to the main design principles of microservices (e.g., horizontal scalability, isolation of failure, decentralization), and —if not— understanding how to refactor it, are two key issues [147, 151]. Even if there exist tools for automatically detecting architectural smells in software applications (e.g., those reported by [11]), to the best of our knowledge, such tools do not permit to automatically identify and refactor architectural smells possibly violating the key design principles of microservices. As a result, such a kind of analyses/refactorings is currently to be manually performed.

In addition, the current support for containerising microservices should be enhanced. More precisely, developers and operators are currently required to manually package each microservice in a proper container [121], by manually identifying and configuring a container capable of satisfying the deployment requirements of a microservice, i.e., providing the runtime environment it needs to run (e.g., operating system, libraries). Such a process must be manually repeated whenever a developer wishes to modify the runtime environment actually used to run a microservice, e.g., because the latter has been updated and it now needs additional software support.

As a result, designing the architecture of a microservice-based application and planning its actual container-based deployment is inherently hard. The following is indeed widely recognised as a key research challenge:

> *How to effectively design microservice-based applications and deploy them over container-based platforms is a major challenge in enterprise IT [95, 130, 131].*

1.1 Research objectives and research contributions

This thesis aims at proposing proper solutions for tackling the aforementioned research challenge. In particular, this thesis aims at advancing the state-of-the-art on: (o_1) Analysing and refactoring microservices, (o_2) Automatically packaging microservices, and (o_3) Enacting the deployment of microservices. In the following, we discuss the three main research objectives (i.e., o_1, o_2, o_3) and for each research objective we summarize the research contributions.

(o_1) Analysing and refactoring microservices

There is a need for design-time methodologies to systematically identify architectural smells possibly violating key design principles of microservices, and to select architectural refactorings allowing to resolve such smells. To support such a kind of analyses, we first aim at providing a suitable modelling for microservices. Such modelling should allow describing the structure of microservice-based architectures and the interactions among microservices. We then aim at proposing methodologies that, based on such modelling, allow to identify the occurrence of architectural smells in a microservice-based application, and to concretely illustrate how to refactor its architecture to resolve the identified smells.

In this thesis, we first provide a taxonomy that singles out the set of architectural smells possibly violating some main principles of microservices, by also eliciting the corresponding architectural refactorings (Chapter 2). The currently available information on architectural smells indicating possible violations of the design principles of microservices is scattered over a considerable amount of literature. Unfortunately, this makes it difficult to consult the body of knowledge on the topic, both for researchers willing to investigate on microservices and for practitioners daily working with them. In this thesis, we analyse such literature, in order to identify the most recognised smells, as well as architectural refactorings for resolving the smells occurring in a microservice-based application. The result of such analysis is a taxonomy of design principles, architectural smells and corresponding refactorings.

Starting from the above mentioned taxonomy, we then propose a design-time methodology to identify the architectural smells affecting a microservice-based application, and to select refactorings allowing to resolve such smells (Chapter 3). A convenient way to represent service-based applications is by using a topology graph [18], whose nodes represent the application components, and whose arcs represent the dependencies among such components. The *Topology and Orchestration Specification for Cloud Applications* (TOSCA [124]) meets this intuition, by providing a standardised modelling language for representing the topology of a cloud application.

We hence propose a model for describing the architecture of a microservice-based application with the OASIS standard TOSCA. We indeed introduce the μTOSCA model, which allows describing service-based architectures as typed directed graphs, where nodes represent software components forming an architecture (e.g., services and databases), and arcs represent the interactions occurring among such components. In addition, μTOSCA permits also to associate nodes to the teams responsible of such nodes.

To illustrate the feasibility of our approach supporting the design of microservice-based applications, we also present a prototype tool (called μFRESHENER). μFRESHENER provides a web-based graphical user interface for (i) creating the μTOSCA representation of microservice-based applications, (ii) automatically identifying architectural smells in represented applications and (iii) exploring/applying architectural refactorings for resolving the identified smells.

(o_2) Automatically packaging microservices

The availability of techniques for automatically packaging each microservice in a proper container is also crucial for supporting the deployment of microservices. Our aim is hence to propose a solution where developers can describe only the microservices forming an application, and the software support needed by each microservice. Such a description should then be fed to tools capable of automatically searching the (image of) containers that satisfy the software requirements of each microservice. In this thesis, we define techniques for automatically searching the (image of) containers and for automatically packaging each microservice in the proper container.

Developers and operators are currently required to manually select and configure an appropriate runtime environment for each component forming their microservice-based applications. Such a process must then be manually repeated whenever a developer wishes to modify the virtual environment actually used to run a microservice, e.g., because the latter has been updated and it now needs additional software support. Our objective is instead to allow developers to describe only the components forming an application, the dependencies occurring among such components, and the software support needed by each component [17].

In this thesis, we hence propose techniques for automatically packaging each microservice in the proper environment (Chapter 4). We first propose a TOSCA-based representation for multi-component applications, and we show how to use it to specify only the components forming an application. We then present a solution to automatically complete TOSCA application specifications, by searching the runtime environments (viz., Docker containers) that provide the software support needed by the application components. The solution is prototyped by TOSKERISER, a tool that automatically packages TOSCA application specifications, by discovering and including Docker-based runtime environments.

We also propose techniques for searching the appropriate runtime environment for each microservice (Chapter 5). We hence present DOCKERANALYSER and DOCKERFINDER, two tools enabling the search for Docker-based runtime environments in TOSKERISER. DOCKERANALYSER is a tool that permits building customised analysers of Docker images. The architecture of DOCKERANALYSER is designed to crawl Docker images from a remote Docker registry, to analyse each image by running an analysis function, and to store the results into a local database. Image analysers can then be created by instantiating DOCKERANALYSER with a custom analysis function. One of such analysers is DOCKERFINDER, which automatically extracts metadata characterising Docker images, including the software support they offer. DOCKERFINDER also permits searching for images by submitting multi-attribute queries through a remotely accessible API. DOCKERFINDER is invoked by TOSKERISER to automatically search the Docker-based runtime environments needed to package the software components.

(o_3) Enacting the deployment of microservice

The possibility of independently managing the microservices forming an application from the containers used to run them is also important. Currently, available container orchestrators indeed treat containers as a sort of “black-boxes", with containers constituting the minimal deployment entity that can be orchestrated. To overcome this limitation, we aim at devising solutions for deploying microservices on top of existing container orchestrators, by also allowing to independently manage the lifecycle of microservices from that of the containers hosting them.

Microservices are typically packaged within containers, and their actual deployment is then enacted by exploiting container orchestrators. The latter is indeed widely used to automate the deployment and management of containers at a large scale, as they provide all the necessary abstractions for scaling, discovering, load-balancing and interconnecting containers over single or multi-host systems.

At the same time, container orchestrators treat containers as a sort of “black-boxes", since containers constitute the minimal deployment entity that can be orchestrated. In other words, container orchestrators can independently manage the containers forming an application, but they do not provide any support for managing the components running within such containers. The lifecycle of the components forming a (microservice-based) application is hence tightly coupled to that of their hosting containers, as each component cannot be managed independently from its hosting container.

In this thesis, we propose a solution for overcoming the aforementioned limitation, i.e., for independently managing the software components from the containers used to run them (Chapter 6). More precisely, we propose a novel architectural approach enabling a component-aware orchestration of microservice-based applications deployed over distributed containers. We indeed developed TOSKOSE, a tool that takes as input a TOSCA-based representation of a microservice-based application and produces a new deployment artifact that can be distributed on top of existing Docker container orchestrators.

The prototype toolchain

It is worth highlighting that the contributions contained in this thesis, and in particular their prototype implementations, actually constitute a prototype toolchain for supporting the analysis and the deployment of microservices (Fig. 1.1).

The input of the toolchain is a μTOSCA specification of the architecture of a microservice-based application. Such a specification can then be fed to μFRESHENER for discovering architectural smells making such application possibly violating key design principles of microservices. μFRESHENER also allows resolving the identified architectural smells by proposing suitable refactorings. The output of μFRESHENER is a refactored μTOSCA specification, where architectural smells have been resolved according to the choices made by the software architect exploiting μFRESHENER to refine the design of her application.

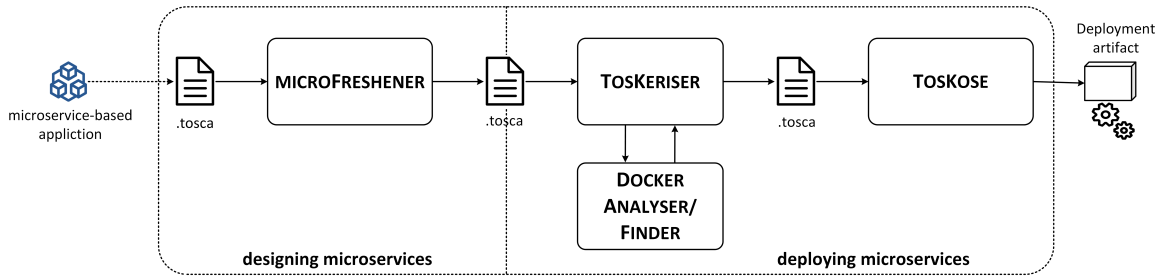


FIGURE 1.1. The toolchain formed by the prototypes of our research contributions.

The refactored TOSCA-based specification is passed to TOSKERISER, which automatically searches for an appropriate container (by querying DOCKERFINDER) for each microservice, and which packages each of such microservices in the correspondingly discovered container. The completed specification produced by TOSKERISER can then be processed by TOSKOSE, which produces the concrete artifacts allowing to deploy and manage the considered application on top of existing container orchestrators.

1.2 Thesis structure

Fig. 1.2 provides a concept map of the thesis that summarises the structure of the thesis. It shows the connections among the research challenges (i.e., designing and deploying microservices) and the three research objectives (i.e., analysing and refactoring microservices, automatically packaging microservices, enacting the deployment of microservices) and each research objective is connected to the main contributions and each contribution is connected to the articles where such contribution has been presented. The content of each chapter presented in this thesis is taken as is from a published article. For this reason, some content of the chapters may slightly overlap. More precisely, the contents of Chapters 2, 3, 4 and 5 are taken verbatim from the publications [120], [29], [26], and [28], respectively. Chapter 6 is instead taken from an article recently submitted for publication [20]. Further information about each chapter is given below.

In **Chapter 2** we present a systematic, multivocal review on microservices, where the most recognised architectural smells possibly violating key design principles of microservices are singled out, together with the architectural refactorings allowing to resolve such smells.

The systematic review presented in Chapter 2 was published in [120], which appeared in the journal "SICS Software-Intensive Cyber-Physical systems", and which was awarded with the "SummerSOC Young Researcher Award" at the "13th Symposium and Summer School On Service-Oriented Computing" (SummerSOC 2019).

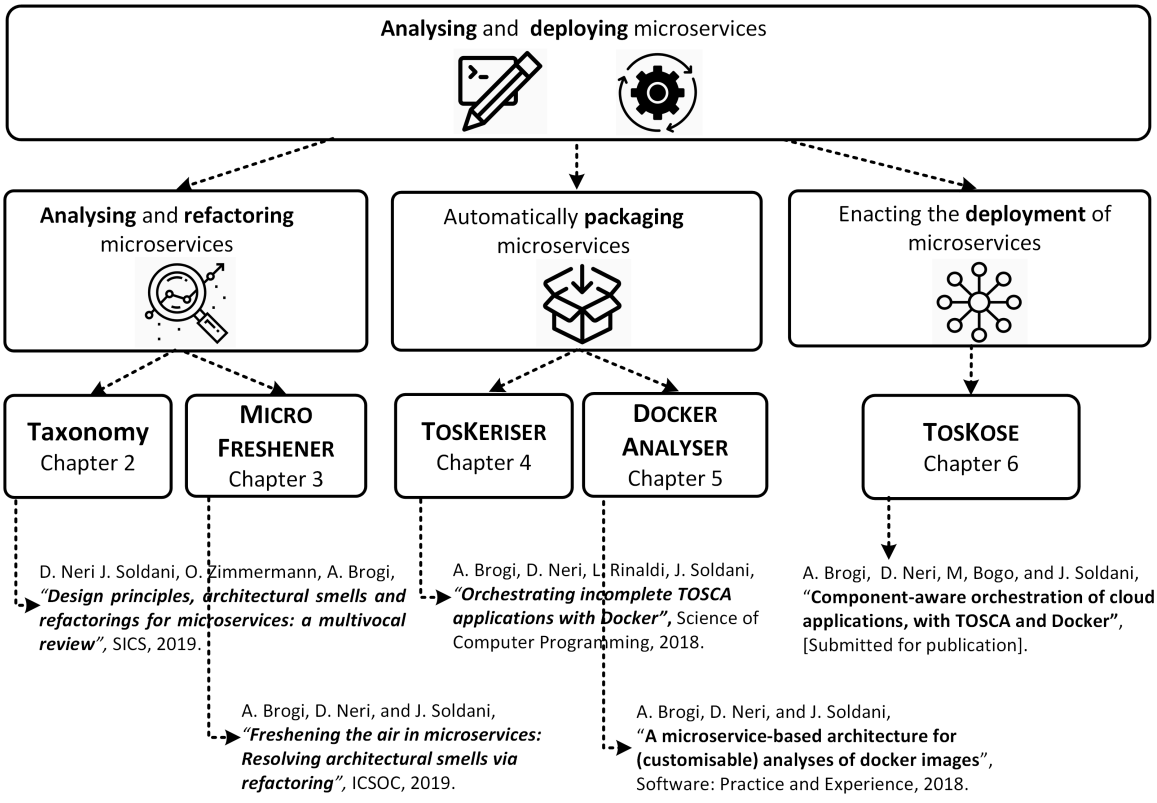


FIGURE 1.2. Concept map that shows the connection among the research challenges, research objectives, tools and articles presented in the thesis.

In **Chapter 3** we illustrate the methodology to systematically identify the architectural smells that possibly violate the main design principles of microservices. After presenting μTOSCA (a modelling of microservice-based applications with the OASIS standard TOSCA), we present our methodology to discover architectural smells and apply suitable refactorings, as well as its prototype implementation, i.e., μFRESHENER.

The methodology and prototype illustrated in Chapter 3 were published in [29], presented in a workshop of the 17th International Conference on Service-Oriented Computing (ICSOC 2019).

In **Chapter 4** we present a technique to automatically complete TOSCA application specifications, by discovering Docker-based runtime environments that provide the software support needed by the components forming such applications. We also present TOSKERISER, a tool that automatically completes TOSCA application specifications, by discovering and including Docker-based runtime environments providing the software support needed by their components. We also illustrate the effectiveness of our technique and of its prototype implementation by means of a case study based on a third-party application.

The results illustrated in Chapter 4 were published in [26], which appeared in the journal "Science of Computer Programming".

In **Chapter 5** we introduce DOCKERANALYSER, a microservice-based tool that permits building customised analysers of Docker images. We show how users can build their own image analysers by instantiating DOCKERANALYSER with a custom analysis function. We also showcase the effectiveness of DOCKERANALYSER by presenting two different use cases, each building a different analyser of Docker images by running DOCKERANALYSER with a different analysis function and different configuration options.

The results in Chapter 5 were published in [28], which appeared in the journal "Software: Practice and Experience".

In **Chapter 6** we present our architectural approach for deploying microservice-based applications on top of existing container orchestrators, by also allowing to manage each component independently from the container used to run it. We also present TOSKOSE, a tool that takes as input a TOSCA specification and produces the concrete artifacts for deploying an application according to our architectural approach and by distributing it on top of existing container orchestrators. We finally illustrate the feasibility and effectiveness of our approach by means of two concrete case studies involving third-party applications.

The results in Chapter 6 are currently submitted for publication [20].

In **Chapter 7** we summarise and discuss the research contributions presented in this thesis, by also providing perspectives for future work.

Part II

Analysing microservices

CHAPTER 2

DESIGN PRINCIPLES, ARCHITECTURAL SMELLS AND REFACTORINGS FOR MICROSERVICES: A MULTIVOCAL REVIEW

Potential benefits such as agile service delivery have led many companies to deliver their business capabilities through microservices. Bad smells are however always around the corner, as witnessed by the considerable body of literature discussing architectural smells that possibly violate the design principles of microservices. In this chapter ,we systematically review the white and grey literature on the topic, in order to identify the most recognised architectural smells for microservices and to discuss the architectural refactorings allowing to resolve them.

The content of this Chapter was published in [120], which appeared in the journal "SICS Software-Intensive Cyber-Physical systems".

2.1 Introduction

Microservices architectures, first discussed by Lewis and Fowler [108], bring various advantages such as ease of deployment, resilience, and scaling [121]. Many IT companies deliver their core business through microservice-based solutions nowadays, with Amazon, Facebook, Google, LinkedIn, Netflix and Spotify being prominent examples. To deliver on their promises, microservices must be designed in quality and style, which is unfortunately not always the case [147].

Microservice-based architectures can be seen as peculiar extensions of service-oriented architectures, characterized by an extended set of design principles [130, 169]. These principles include shaping services around business concepts, decentralising all development aspects of microservice-based solutions (from governance to data management), adopting a culture of au-

tomation, ensuring the independent deployability and high observability of microservices, and isolating failures [121]. A key research question therefore is:

> *How can architectural smells affecting design principles of microservices be detected and resolved via refactoring?*

The currently available information on architectural smells indicating possible violations of the design principles of microservices is scattered over a considerable amount of literature. Unfortunately, this makes it difficult to consult the body of knowledge on the topic, both for researchers willing to investigate on microservices and for practitioners daily working with them.

Our objective here is to systematically analyse such literature, in order to identify the most recognised smells, as well as architectural refactorings for resolving the smells occurring in an application [168]. In particular, we focus on the design principles dealing with the dynamic aspects of the interactions between microservices at runtime, i.e., on the process viewpoint, as per the 4+1 viewpoint scheme [104]. More precisely, we consider the independent deployability of microservices, their horizontal scalability, isolation of failures and decentralisation.

As recommended by Garousi et al. [71], to capture both the state of the art and the state of practice in the field, we conducted a multivocal systematic review of the existing literature, including both white literature (i.e., peer-reviewed papers) and grey literature (i.e., blog posts, industrial whitepapers and books). We selected 41 studies, published since 2014 (when the microservice-based architectural style was first discussed [108]) until the end of January 2019. Then, following the guidelines for systematic reviews [71, 132], we excerpted a taxonomy of design principles, architectural smells and corresponding refactorings. We then exploited this taxonomy to classify the selected studies, in order to distill the actual recognition of the identified smells and the usage of the corresponding refactorings.

In this chapter, we illustrate the results of our study. More precisely, we first present the obtained taxonomy, including seven architectural smells and 16 refactorings, organised by design principles. We then discuss each smell, by illustrating why it can violate the design principle it is associated with, and by showing how to resolve it by means of an architectural refactoring.

We believe that the results presented in this study can provide benefits to both researchers and practitioners interested in microservices. A systematic presentation of the state of the art and practice on architectural smells and refactorings for microservices provides a body of knowledge to develop new theories and solutions, to analyse and experiment research implications, and to establish future research directions. At the same time, it can help practitioners to better understand the currently most recognised architectural smells for microservices, and to choose among the architectural refactorings allowing to resolve such smells. This can have a pragmatic value for practitioners, who can use our study as a starting point for microservices experimentation or as a guideline for day-by-day work with microservices.

The rest of the chapter is organised as follows. Sect. 2.2 defines the research problem and illustrates the research methodology. Sect. 2.3 presents a taxonomy for design principles, archi-

tectural smells and refactorings, which is retaken in Sect. 2.4 to overview the current state of the art and practice on such smells and refactorings. Sects. 2.5 and 2.6 discuss potential threats to the validity of our study and related work, respectively. Finally, Sect. 2.7 draws some concluding remarks.

2.2 Setting the stage

The objective of this survey is to identify architectural smells indicating possible violations of microservices principles, as well as the currently available solutions for refactoring microservice-based architectures in order to resolve these smells.

Scope of the survey

This survey focuses on the architectural principles of microservices that pertain to the process viewpoint, i.e., dealing with the dynamic aspects of microservices interacting at runtime [104].

We started from the principles proposed by Newman [121] and Lewis and Fowler [108], also considering the mapping to tenets proposed by Zimmermann [169]. From these works, we selected four principles:

1. The microservices forming an application should be *independently deployable*.
2. The microservices should be *horizontally scalable*.
3. *Failures* should be *isolated*.
4. *Decentralisation* should occur in all aspects of microservice-based applications, from data management to governance.

The above selection was based on three criteria:

(a) *Roots* in highly significant design time and runtime quality attributes and style-defining elements,

(b) *Consequences* of not adhering to a principle in terms of technical risk and re-engineering cost, and

(c) *Generality*, i.e., if these four principles are met, others follow or can be achieved with similar means.

For instance, independent deployability is a defining tenet in most definitions of microservices and enables decentralized continuous delivery, thereby meeting criteria (a) and (c). Scalability is a quality attribute (a) and horizontal scalability is hard to retrofit (an aspect of (b)). Failure isolation meets criteria (a) and (b). Finally, decentralization is mentioned as crucial (and novel)

in many introductions to microservices and enables independent, autonomous decision making, as required to achieve (a) and (c).

Search for studies

With the objective of capturing the state of the art and practice in the field, we searched for both white literature (i.e., peer-reviewed journal and conference articles) and grey literature (i.e., blog posts, industrial whitepapers and books), in line with what recommended by Garousi et al. [71].

The structuring of the search string was done by following the guidelines provided by Petersen et al. [132]. We indeed identified the search string guided by the PICO terms of our resarch problem, and the keywords were taken from each aspect of our research problem. Differently from Petersen et al. [132], we did not restrict our focus to specific research settings. By restricting ourselves to certain types of research settings, we could have obtained a biased or incomplete analysis, as some architectural smells or refactorings might have been over-/under-represented for a certain type of study.

As a result, our search string was formed by the following terms:

$$\texttt{microservice*} \;\wedge\; \texttt{(smell* } \vee \texttt{ antipattern* } \vee \texttt{ bad practice* } \vee \texttt{ pitfall* } \vee \texttt{ refactor* } \vee \texttt{ reengineer*)}$$

(where '*' matches lexically related terms). The search was restricted to studies published since the beginning of 2014 (when microservices were first proposed by Lewis and Fowler [108]) until the end of January 2019 (when the present study was initiated).

The search of white literature was carried out in the following indexing databases: ACM Digital Library, DBLP, EI Compendex, IEEE Xplore, INSPEC, ISI Web of Science, Science Direct, SpringerLink. Given the recency of the field and concerns with indexing, Google Scholar played a key role for the initial selection before the inclusion and exclusion stage. The search for industrial studies was instead carried out in renowned blogs in the software engineering community (such as DZone, InfoQ and TechBeacon), in the blog of ThoughtWorks, and in books published by practitioners.

Sample selection

The above described search criteria were matched by more than 150 studies, which we carefully screened to keep only those studies that were satisfying both the following inclusion criteria:

- A study is to be selected if it presents *at least one architectural smell* pertaining to one of the considered architectural principles of microservices (i.e., independent deployability, horizontal scalability, isolation of failure, or decentralisation).

- A study is to be selected if it presents *at least one refactoring* for resolving one of the architectural smells it discusses.

The inclusion criteria were defined with the ultimate goal of selecting only representative studies, discussing both the architectural smells (pertaining to the process viewpoint) and their corresponding refactorings. As a result, 41 studies were selected to be analysed further. The list of references to the selected studies is in Table 2.1, which also classifies them by colour.

2.3 A taxonomy for design principles, architectural smells and refactorings

Fig. 2.1 illustrates a taxonomy for the architectural smells pertaining to the considered design principles, and for the refactorings[1] allowing to resolve such smells. We obtained our taxonomy by following the guidelines for conducting systematic reviews in software engineering proposed by Petersen et al. [132]:

1. We established the *design principles*, by aligning them with those pertaining to the process viewpoint (as per [169]).
2. We identified the *architectural smells* by performing a first scan of the selected studies.
3. We excerpted the concrete *refactorings* directly from the selected studies after additional scans.

The identified design principles, architectural smells and refactorings were manually organised to obtain a taxonomy. The taxonomy underwent various iterations among the authors of this study, and it was submitted for validation to an external expert. This resulted in some corrections and amendments to the first version of the taxonomy, which resulted in the taxonomy displayed in Fig. 2.1.

2.4 Architectural smells and refactorings

Table 2.1 shows the classification of all selected studies based on the taxonomy introduced in Sect. 2.3. The table provides a first overview of the coverage of design principles, architectural smells and refactorings over the selected studies, despite (for reasons of readability and space) it only displays the classifications over the smells listed in the taxonomy[2]. Such coverage is also displayed in Fig. 2.2, from which we can observe that all architectural smells in the taxonomy are

[1] For the sake of clarity, in the taxonomy we follow the naming of integration patterns proposed by Hohpe and Woolf [90].

[2] The detailed classification, displaying each occurrence of each refactoring, is publicly available at `https://github.com/di-unipi-socc/microservices-smells-and-refactorings`.

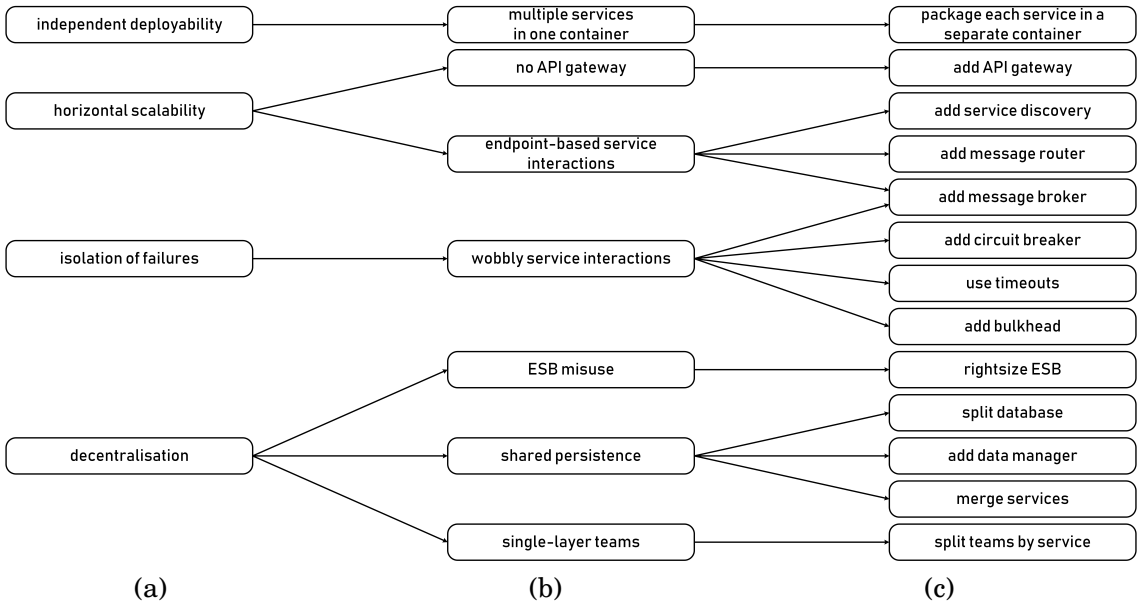


FIGURE 2.1. A taxonomy for (a) the design principles pertaining to the process viewpoint, (b) the architectural smells possibly violating such principles, and (c) the refactorings resolving such smells.

significantly recognised by the authors of the selected studies, hence making it worthy to discuss them in detail.

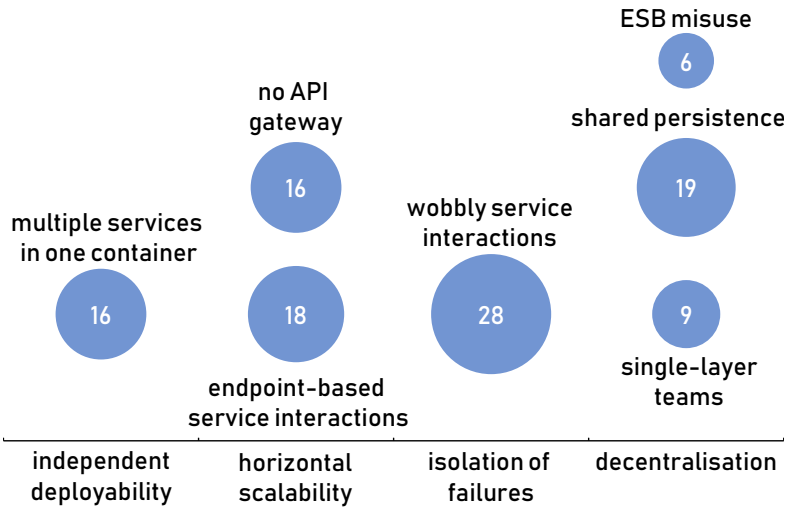


FIGURE 2.2. Coverage of the architectural smells in the selected studies. The size of each bubble is directly proportional to the number of selected studies discussing the corresponding smell. This number is also reported within each bubble.

	colour	**independ. deployab.**	**horizontal scalability**		**isolation of failures**	**decentralisation**		
		multiple services in one cont.	*no API gateway*	*endpoint-b. service inter.*	*wobbly service inter.*	*ESB misuse*	*shared persist.*	*single layer teams*
[7]	*g*		✓					
[8]	*w*			✓	✓			
[12]	*w*	✓		✓	✓			✓
[13]	*w*	✓	✓	✓	✓			
[15]	*g*		✓	✓	✓		✓	
[21]	*g*		✓	✓	✓	✓		
[39]	*g*						✓	
[40]	*g*	✓	✓		✓		✓	
[41]	*w*	✓						✓
[51]	*g*				✓			
[55]	*w*		✓	✓	✓			
[56]	*w*		✓	✓	✓			
[64]	*w*	✓						
[69]	*w*						✓	
[72]	*g*							✓
[74]	*g*				✓			
[75]	*g*						✓	✓
[91]	*g*	✓		✓	✓	✓	✓	
[92]	*g*	✓	✓	✓	✓	✓	✓	
[95]	*w*	✓			✓			
[99]	*w*				✓		✓	✓
[101]	*w*				✓		✓	
[103]	*g*	✓	✓	✓	✓			
[108]	*g*			✓	✓	✓		✓
[115]	*g*	✓						
[111]	*g*			✓	✓			
[119]	*g*		✓	✓	✓		✓	✓
[121]	*g*	✓		✓	✓			
[123]	*g*	✓	✓	✓	✓			
[137]	*g*				✓		✓	
[138]	*g*		✓	✓			✓	
[139]	*g*		✓		✓		✓	
[141]	*g*				✓			
[142]	*g*			✓	✓		✓	
[144]	*w*	✓						
[147]	*w*	✓	✓		✓		✓	
[151]	*w*		✓	✓		✓	✓	
[152]	*w*						✓	✓
[153]	*w*	✓					✓	
[163]	*g*			✓	✓		✓	✓
[169]	*w*	✓				✓		

TABLE 2.1. References to the selected studies, and their classification by *colour* (i.e., *w*hite or *g*rey literature) and according to the taxonomy in Fig. 2.1.

We hereafter illustrate how (according to the authors of the selected studies) each design principle can be affected by each corresponding architectural smell, as well as how each smell can be resolved by applying a corresponding refactoring. When multiple refactorings are applicable to resolve an architectural smell, to provide a first measurement of how much a refactoring is used to resolve it, we display the weight[3] of each refactoring by exploiting %-based pie charts.

2.4.1 Independent deployability

In microservice-based applications, each microservice should be operationally independent from the others, meaning that it should be possible to deploy and undeploy a microservice indepedently from the others [121]. This indeed impacts on the initial deployment of a microservice, which can get started without waiting for other microservices to be running, as well as on the possibility of adding/removing replicas of a microservice at runtime.

We discuss below the MULTIPLE SERVICES IN ONE CONTAINER smell, showing how it violates the above principle and how it can be resolved.

Multiple services in one container Containers (such as Docker containers) provide an ideal way to deploy microservices addressing the above requirement, if properly used. Each microservice can indeed be packaged in a container image, and different instances of a same microservice can be launched by spawning different containers from the corresponding image. With this view, the orchestration of the deployment and management of a microservice-based application can be performed by exploiting the currently available support for orchestrating Docker containers [91].

The above is the right way of using containers, at least according to the authors of 16 of the selected studies. They indeed highlight how placing multiple services in one container would constitute an architectural smell for the independent deployability of microservices. If two microservices would be packaged in the same Docker image, spawning a container from such image would result in launching both microservices. Similarly, stopping the container would result in stopping both microservices. In other words, by placing two microservices in the same container, these services would operationally depend one another, as it would not be possible to launch a new instance of one of such microservices, without also launching an instance of the other.

If the MULTIPLE SERVICES IN ONE CONTAINER smell occurs, the solution is to refactor the application in such a way that each microservice is packaged in a separate container image.

2.4.2 Horizontal scalability

The possibility of adding/removing replicas of a microservice is a direct consequence of the independent deployability of microservices. To ensure its horizontal scalability, all the replicas of

[3]We measure the weight of a refactoring as the percentage of its occurrences among all occurrences of all refactorings for the same smell. This is analogous to what done by Pahl et al. [127] to measure weights while classifying studies on cloud container technologies.

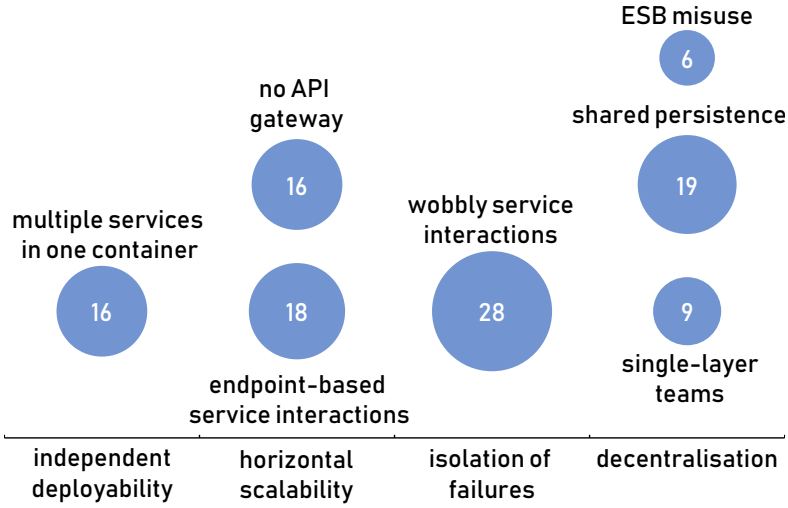


FIGURE 2.3. Weights (w) and occurrences (o) of the refactorings for the ENDPOINT-BASED SERVICE INTERACTIONS smell.

a microservice m should be reachable by the microservices invoking m [91].

In the selected studies, two architectural smells emerged as possibly violating the horizontal scalability of microservices, i.e., ENDPOINT-BASED SERVICE INTERACTIONS and NO API GATEWAY, which we discuss hereafter.

Endpoint-based service interactions. This smell occurs in an application when one or more of its microservices invoke a specific instance of another microservice (e.g., because its location is hardcoded in the source code of the microservices invoking it, or because no load balancer is used). If this is the case, when scaling out the latter microservice by adding new replicas, these cannot be reached by the invokers, hence only resulting in a waste of resources.

From the selected studies, it became evident that the ENDPOINT-BASED SERVICE INTERACTIONS smell can be resolved by applying three different refactorings (Fig. 2.3). The most common solution is to introduce a service discovery mechanism. Such mechanism can be implemented as a service storing the actual locations of all instances of the microservices in an application [139]. Microservice instances send their locations to the service registry at startup, and they are unregistered at shutdown. When willing to interact with a microservice, a client can then query the service discovery to retrieve the location of one of its instances.

The other two possible solutions share the same goal, i.e., decoupling the interaction between two microservices by introducing an intermediate integration pattern. Nine of the selected studies indeed suggest to introduce a message router (e.g., a load balancer), so that the requests to a microservices are routed towards all its actual instances. Four of the selected studies instead suggest to exploit message brokers (e.g., message queues) to decouple the interactions between

two or more microservices.

No API gateway. When a microservice-based application lacks an API gateway, the clients of the application necessarily have to invoke its microservices directly. The result is a situation similar to that of the ENDPOINT-BASED SERVICE INTERACTIONS smell, with the invoker being a client of the application. The client indeed interacts only with the specific instances of the microservices it needs. If one of such microservices is scaled out and the client still keeps invoking the same instance of the microservice, then we have a waste of resources.

The authors of all the selected studies discussing the NO API GATEWAY smell agree that the solution to this smell is to add one API gateway to the application. The latter act as single entry points for all clients, and they handle requests either by routing them or by fanning them out to the instances of the microservices that must handle them [139].

It is worth noting that, even if the NO API GATEWAY smell results in a similar situation to that of the ENDPOINT-BASED SERVICE INTERACTIONS smell, the refactorings to resolve them are different. The reason for this resides in the main difference between the two architectural smells. The NO API GATEWAY smell occurs at the edge of the architecture of a microservice-based application, with the clients of the application directly invoking its microservices, while the ENDPOINT-BASED SERVICE INTERACTIONS smell occurs in between its microservices [119]. Given this, the introduction of an API gateway can be useful not only for facilitating the horizontal scalability of the microservices forming an application, but also for various other reasons. For instance, rather than implementing end-user authentication or throttling in each microservice, these can be implemented once for the whole application in the API gateway [7].

2.4.3 Isolation of failures

Microservices can fail for many reasons (e.g., network or hardware issues, application-level issues, bugs), hence becoming unavailable to serve other microservices. Additionally, communication fails from time to time in any kind of distributed system, and this is even more likely to occur in microservice-based systems, simply because of the amount of messages xchanged among microservices [95]. Microservice-based applications should hence be designed so that each microservice can tolerate the failure of any invocation to the microservices it depends on [108]. If this is ensured, then a microservice-based application results to be much more resilient than a monolithic application, simply because failures affects only few microservices in an application, instead of the whole monolith [121].

The authors of the selected studies identify and discuss an architectural smell that can possibly violate the isolation of failures in microservice-based solutions. This is the WOBBLY SERVICE INTERACTIONS smell, which we discuss hereafter.

Wobbly service interactions. The interaction of a microservice m_i with another microservice m_f is “wobbly" when a failure in m_f can result in triggering a failure also in m_i. This typically happens when m_i is directly consuming one or more functionalities offered by m_f, and m_i is not

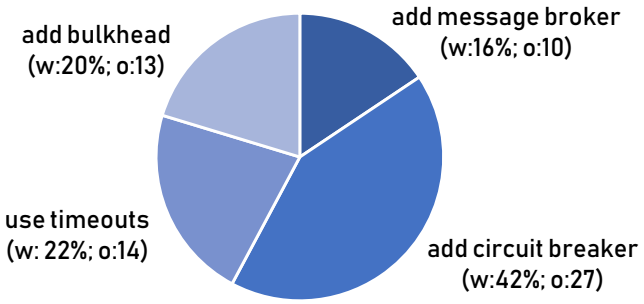


FIGURE 2.4. Weights (w) and occurrences (o) of the refactorings for the WOBBLY SERVICE INTERACTIONS smell.

provided with any solution for handling the possibility of m_f to fail and be unresponsive. If this is the case, m_i will also fail in cascade, and (in a worst case scenario) the failure of m_i can result in triggering the failure of other microservices, which in turn trigger other cascading failures, and so on [95].

To avoid WOBBLY SERVICE INTERACTIONS (such as the one between m_i and m_f described above), the authors of the selected studies identify four possible solutions (Fig. 2.4). The most common solution is the usage of a circuit breaker to wrap the invocations from a microservice to another. In the normal “closed" state, the circuit breaker forwards the invocations to the wrapped microservice, and it monitors their execution to detect and count failing invocations. Once the frequency of failures reaches a certain (customisable) threshold, the circuit breaker trips and “opens" the circuit. All further calls to the wrapped microservice will “safely fail", as the circuit breaker will immediately return an error message to the calling microservices. The latter can then exploit the error messages returned by the circuit breaker to avoid failing themselves [108].

Following the same baseline idea of circuit breakers, ten of the selected studies propose to decouple the interaction between invoking and invoked microservices by exploiting a message broker (e.g., a message queue). The usage of a broker allows the invoker to send its requests to the broker, and allows the invoked microservice to process such requests when it is available. In this way, there is no direct interaction between the two microservices, and the invoker does not fail when the invoked microservice fails (as the former continues to send messages to the broker).

On the other hand, the usage of message brokers is more costly compared to circuit breakers. The reason is that message brokers require to intervene on the interaction protocol between two microservices, which should start putting and getting messages to/from the broker. Instead, with circuit breakers the interaction protocol between two microservices is unaltered, as a circuit breaker simply wraps the invocation of a microservice. This is the reason why message brokers are much less discussed than circuit breakers.

The most discussed alternative to circuit breakers are however timeouts, which are a simple

yet effective mechanism allowing a microservice to stop waiting for an answer from another microservice, when the latter is unresponsive (e.g., since it failed or due to network issues). Well-placed timeouts provide fault isolation, as the fact that a microservice is unresponsive does not create any other issue in the microservices invoking it [121]. However, such a kind of solution might not likely to be applicable nowadays, as some of the APIs used to remotely invoke microservices have few or no explicit timeout settings [121]. Note that the timeout can be also set in the invoker (e.g., by setting the timeout on an HTTP request), hence it is not always requested to have a timeout setting on the invoked service.

Finally, another alternative is the usage of bulkheads, whose ultimate goal is to enforce the principle of damage containments (like bulkheads in ships, which prevent water to flow across sections). The idea is that, if cascading failures cannot be avoided, they should at least be limited by exploiting bulkheads. More precisely, the microservices forming an application should be logically and/or physically partitioned so as to ensure that the failure of a microservice can be propagated at most to the other microservices in the same partition, by preventing the rest of the system from being affected by such failure [123].

2.4.4 Decentralisation

Decentralisation should occur in all aspects of microservice-based applications [121]. This also means the business logic of an application should be fully decentralised and distributed among its microservices, each of which should own its own domain logic [169].

The authors of the selected studies indentify and discuss three architectural smells possibly violating the above principle, i.e., the ESB MISUSE, SHARED PERSISTENCE and SINGLE-LAYER TEAMS smells. We hereafter discuss them, by also illustrating the refactorings currently employed to resolve them.

ESB misuse. The misuse of Enterprise Service Buses (ESB) products is considered to be an architectural smell by the microservice community. When positioned as a single central hub (with the services as spokes), an ESB may become a bottleneck both architecturally and organizationally [130]. “Smart endpoints & dumb pipes" has been a recommended practice since the very beginnings of service-oriented architectures [169] that regrettably has not always been followed in all SOA implementations. Such ESB abuse may lead to undesired centralisation of business logic and dumb services [121]. The microservices community therefore (re-)emphasizes the decoupling of microservices and their cohesiveness [108].

Whenever a central ESB is used for connecting microservices in an application, the topology should be refactored to remove the dependency on a single middleware component instance. Multiple instances should instead be used, and they should implement queue-based asynchronous messaging. The latter only permits adding and removing messages, hence forming a “dumb pipe". The “smart" part should be left to the microservices, which implement the logic for deciding when/how to process the messages in the message broker [151]. Additional infrastructure logic, for

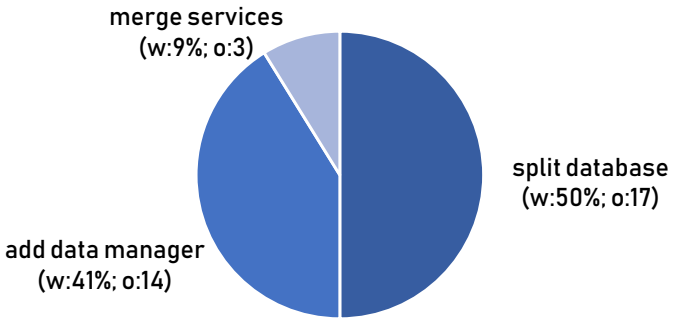


FIGURE 2.5. Weights (w) and occurrences (o) of the refactorings for the SHARED PERSISTENCE smell.

instance traffic management capabilities, may be placed in side cars accompanying each service. This repositioning and rectification of ESB middleware improves the decoupling characteristics of the services architecture and reestablishes the original “smart endpoints & dumb pipes" recommendations from the first wave of service-orientation.

Shared persistence. The SHARED PERSISTENCE smell occurs whenever two microservices access and manage the same database, possibly violating the decentralisation design principle [147].

The three currently available solutions for refactoring microservices and resolving the SHARED PERSISTENCE smell are shown in Fig. 2.5.

Although the ultimate goal of these three solutions is the same (i.e., having each database accessed by only one microservice), they are very diverse in spirit. They apply to different situations, highly depending on the microservices accessing the same database.

The most discussed solution is to actually split a database shared by multiple microservices, in such a way that each microservice accesses and manages only the data it needs. This solution is the one requiring less intervention on the microservices, as they would continue to use the same protocol to interact with the databases. At the same time, splitting a database into a set of independent databases is not always possible or easy to achieve. Also, if some data is to be replicated among the databases obtained from the split, then mechanisms for (eventual) data consistency should be introduced after the refactoring [147]. Given the above, the split of database is recommended when the microservices accessing the same database implement separate business logics working on disjoint portions of such database [92].

The most discussed alternative is to introduce an additional microservice, acting as “data manager". The data manager becomes the only microservice interacting with and managing the database, and the microservices that were accessing the database now have to interact with the data manager to ask for accessing and updating the data. While this solution introduces some additional communication overhead, it is considered as always applicable, and the data manager

can also be enriched with additional logic for processing the data it manages [92].

Finally, it is worth commenting on the refactoring discussed in three of the selected studies, i.e., merging the microservices accessing the same database. The idea is that, when multiple microservices access the same database, this may be a signal of the fact that the application has been split too much, by obtaining too fine-grained microservices processing the same data. If this is the case, then the possibility of merging such microservices is a concrete option to be evaluated [151].

Single-layer teams. To maximize the autonomy that microservices make possible, the governance of microservices should be decentralised and delegated to the teams that own the microservices themselves. As pointed out by Zimmermann [169], even if this is not a technical concern, it is related to the process viewpoint due to its cross-cutting nature. The microservice community indeed strongly emphasizes the connection between architecture and organisation, especially concerning the integration of the microservices in an application [72, 75, 108].

The classical approach of splitting teams by technology layers (e.g., user interface teams, and middleware teams, and database teams) is hence considered an architectural smell, as any change to a microservice may result in a cross-team project having take time and budgetary approval [108]. This may be the case for the refactorings discussed so far.

The microservice approach to team splitting is orthogonal to the above, as each microservice should be assigned to a full-stack team whose members span across all technology layers. In this way, the interactions for updating a microservice (e.g., to apply one of the refactorings discussed in this section) are limited to the team managing such microservice, which can independently decide how to proceed and implement the updates [41].

In short, if the governance of a microservice-based is organised by SINGLE-LAYER TEAMS, this is an architectural smell. The solution is to split teams by microservice, rather than by technology layer [108].

2.5 Threats to validity

Following the taxonomy developed by Wohlin et al. [162], four potential theats may affect the validity of our study. These are the threats to *external* validity, the threats to *internal* and *construct* validity, and the threats to *conclusions* validity, which we discuss hereafter.

External validity. As per Wohlin et al. [162], the external validity concerns the applicability of a set of results in a more general context. Since we selected the primary studies from a very large extent of online sources, the identified architectural smells and refactorings may only be partly applicable to the broad area of disciplines and practices on microservices, hence threatening external validity.

To reinforce the external validity of our findings, we organised two feedback sessions during our analysis of the existing literature. We analysed the discussion following-up from the feedback

session, and we exploited this qualitative data to fine-tune both our research methods and the applicability of our findings. We also prepared a GitHub repository[4], where we placed the artifacts produced during our analysis, so as to make it available to all who wish to deepen their understanding on the data we produced. We believe that this can help in making our results and observations more explicit and applicable in practice.

Additionally, one may argue that our selection criteria are too restrictive. The rationale behind such criteria is that we aim focusing only on representative studies, by requiring selected studies to discuss at least an architectural smell *and* a refactoring for resolving it. There is however a risk of having missed some relevant literature, as a study might not explicitly mention the architectural smells and refactorings in our taxonomy (Fig. 2.1). To mitigate this threat, we carefully checked both selection criteria against each candidate study, by verifying whether a study was discussing the problems characterised by an architectural smell, and whether it was discussing the architectural changes characterising a refactoring. Even if a study was not explicitly referring to a smell/refactoring, but it was reporting on the corresponding problems/changes, the study was included in the selected literature.

Finally, there is a risk of having missed relevant grey literature, since industrial studies may exploit a different terminology than ours (e.g., a blog post discussing some architectural smells and refactorings may not employ the term “smell" or “refactor"). To mitigate this threat to validity, we included relevant synonyms in the search string, and we exploited the features offered by search engines, which naturally support including related terms in string-based searches.

Construct and internal validity. The internal validity concerns the validity of the method employed to study and analyse data (e.g., the potential types of bias involved), while the construct validity concerns the generalisability of the constructs under study [162].

To mitigate the corresponding potential threats, the obtained taxonomy underwent various iterations among the authors of this study to avoid bias by triangulation, and it was submitted for validation to an external expert. The same process was applied to the classification of the selected studies, and to the results of the analysis.

Conclusions validity. The conclusions validity concerns the degree to which the conclusions of a study are reasonably based on the available data [162].

In this perspective, and with the aim of performing a sound analysis of the data we retrieved, we exploited inter-rater reliability assessment to limit potential biases in our observations and interpretations. Additionally, the observations and conclusions discussed in this chapter were independently drawn, and they were then double-checked against the selected studies and related studies in a joint discussion session.

[4] `http://github.com/di-unipi-socc/microservices-smells-and-refactorings`.

2.6 Related work

There exist various studies on microservices, aimed at analysing and classifying the state of the art and practice on microservices. Pahl and Jamshidi [128] and Taibi et al. [153] present two first systematic mapping studies on microservices. Pahl and Jamshidi [128] discuss agreed and emerging concerns on microservices, position microservices with respect to current cloud and container technologies, and elicit potential research directions. Taibi et al. [153] instead report on architectural patterns common to microservice-based solutions, by discussing the advantages, disadvantages and lessons learned of each pattern. However, neither Pahl and Jamshidi [128] nor Taibi et al. [153] provide an overview both on the architectural smells applicable to microservices and on the refactorings for resolving such smells.

Two other examples are the industrial surveys by Di Francesco et al. [54] and by Ghofrani and Lübke [73], which both discuss the current state of practice on microservices in the IT industry. Both report on empirical studies conducted in the form of surveys for practictioners working everyday with microservices, to elicit the challenges and advantages on employing microservices. This differs from our study, as we aim at distilling the architectural smells that can affect the architecture of a microservice-based solution, as well as the refactorings allowing to resolve such smells.

Similar considerations apply to the systematic review by Soldani et al. [147], who provide an overview on the state of practice on microservices. Soldani et al. systematically analyse the grey literature on microservices, in order to identify the technical/operational advantages and disadvantages of the microservice-based architectural style. The objective of Soldani et al. hence differs from ours, as we aim at discussing concrete architectural smells and refactorings for the microservice-based architectural style.

In this perspective, the objective of the studies by Taibi and Lenarduzzi [151], by Bogner et al. [19], and by Carrasco et al. [41] is much closer to ours. Taibi and Lenarduzzi [151] report on a survey submitted to practictioners experienced with microservices. The survey allowed Taibi and Lenarduzzi to identify 11 microservice-specific architectural smells, each with a refactoring solution allowing to resolve it. Of such smells and refactorings, only four can be related to the design principles of microservices pertaining to the process viewpoint (see Table 2.1). By integrating the work by Taibi and Lenarduzzi with other carefully selected white/grey literature, we managed to extend the set of architectural smells and refactorings pertaining to the process viewpoint with three additional smells and ten additional refactorings.

Bogner et al. [19] present a systematic literature review identifying and documenting architectural smells in SOA-based architectural styles, including microservices. Altough the main focus of their review is on the broader SOA, several smells apply also to microservices. However, the review by Bogner et al. [19] differs from ours, as it focuses only on white literature, and since it does not discuss the architectural refactorings allowing to resolve the identified smells.

Carrasco et al. [41] systematically analyses the white and grey literature on architectural

smells that can occur while migrating from monoliths to microservice-based solutions. They present nine common smells with their potential solutions, which all pertain to the actual development and operation of microservice-based applications (i.e., development and physical viewpoints). The study by Carrasco et al. [41] hence differs from ours, as we focus on the dynamic aspects of microservices that interact at runtime (i.e., process viewpoint).

Similar considerations apply to the study by Furda et al. [69], which focuses on multitenancy, statefulness, and data consistency. Their objective is indeed supporting the migration of enterprise legacy source code to microservices. Finally, the Microservices API Pattern (MAP) language suggests design improvements in the form of an informal cheat sheet. The first MAP patterns have been published by Stocker et al. [150] and by Zimmermann et al. [170].

In summary, to the best of our knowledge, there is currently no study classifying the architectural smells possibly violating the design principles of microservices pertaining to the process viewpoint, together with the refactorings that permit resolving such smells. The latter is precisely the scope of our study, which we have presented in this chapter.

2.7 Conclusions

We presented the results of a multivocal review focused on identifying architectural smells indicating possible violations of the independent deployability, horizontal scalability, fault isolation and decentralisation of microservices, as well as the refactorings allowing to resolve such smells. More precisely, we presented a taxonomy organising seven architectural smells and 16 refactorings, by associating each smell with the design principle(s) it violates, and each refactoring with the smell it resolves. We then provided an overview of the actual recognition of such smells and refactorings in the selected literature. We also discussed why each architectural smell violates the design principle it pertains to, and how each architectural refactoring allows resolving its corresponding smell.

We believe that our study can be of help to both researchers and practitioners interested in microservices. Together with the review by Carrasco et al. [41], our results can help them to understand the well-known architectural smells for microservices, and to choose among the refactorings allowing to resolve such smells. This can have a pragmatic value for practitioners, who can exploit the results of our study in their daily work with microservices. It can also help researchers to shape new solutions and to establish future research directions.

We plan to exploit our results to develop a design-time support for eliminating architectural smells from microservice-based applications. Our idea is to exploit existing languages for the specification of microservice-based applications (such as TOSCA [124], for instance). We then plan to develop a tool for processing the specification of a microservice-based application, to automatically detect the architectural smells occurring in such application, and to suggest the architectural refactorings resolving such smells.

CHAPTER 3

FRESHENING THE AIR IN MICROSERVICES: RESOLVING ARCHITECTURAL SMELLS VIA REFACTORING

The adoption of microservice-based architectures is becoming common practice for enterprise applications. Checking whether an application adheres to the main design principles of microservices, and —if not— understanding how to refactor it, are two key issues in that context. In this chapter we present a methodology to systematically identify the architectural smells that possibly violate the main design principles of microservices, and to select suitable architectural refactorings to resolve them. We also present a prototype implementing the methodology, based on a novel representation of microservices in TOSCA.

This Chapter was published in [29], presented in a workshop of the 17th International Conference on Service-Oriented Computing (ICSOC 2019).

3.1 Introduction

Microservice-based architectures are increasingly considered an enabling technology to shorten the lead time in software development and to effectively scale software application deployments [108, 121]. The interest in microservice-based architectures is witnessed by their adoption by the major IT companies (like Amazon, Facebook, Google, Netflix and Spotify, just to mention some).

Microservice-based architectures can be seen as service-oriented architectures that satisfy some key principles [169]. These include shaping services around business concepts, adopting a culture of automation, decentralising all aspects of microservices (from governance to data management), ensuring their independent deployability and high observability, and isolating failures [121]. As the adoption of microservices is becoming common practice for enterprise appli-

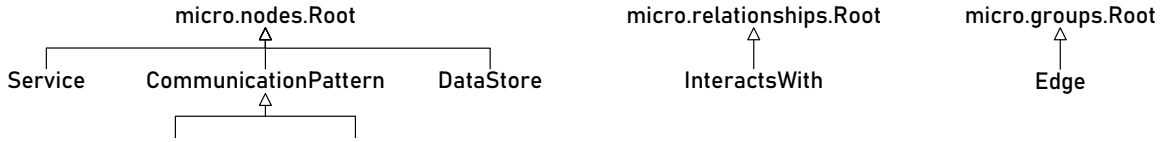


FIGURE 3.1. The node types, relationship types and group types defining μTOSCA.

cations, checking whether an application adheres to the main design principles of microservices, and —if not— understanding how to refactor it, are two key issues [120, 147].

In this chapter, we present a methodology to systematically identify architectural smells possibly violating key design principles of microservices, and to select architectural refactorings allowing to resolve such smells. We take as starting point the industry-driven review presented in [120], which singled out a set of architectural smells possibly violating some main principles of microservices, by also eliciting the architectural refactorings allowing to resolve each smell. In particular, we consider four of the architectural smells in [120], each with the architectural refactorings that permit resolving it.

Our proposal is to model the architecture of a microservice-based application with the OASIS standard TOSCA [124]. We hence introduce μTOSCA, which allows to specify service-based architectures as typed directed graphs. Based on such representation, we formally define the conditions to identify the occurrence the considered architectural smells in a microservice-based application, and we illustrate how to refactor its architecture to resolve identified smells.

We also present μFRESHENER, a prototype showcasing our methodology. We believe that our methodology and its prototype implementation can provide a valuable decision support for designing microservice-based architectures.

The rest of the chapter is organized as follows. Sect. 3.2 introduces μTOSCA. Sects. 3.3 and 3.4 illustrate our methodology to identify/resolve architectural smells and its prototype implementation, respectively. Finally, Sect. 3.5 discusses related work and Sect. 3.6 draws some concluding remarks.

3.2 Modelling service-based architectures with μTOSCA

TOSCA [124] allows to represent service-based architectures as typed directed graphs, where nodes represent software components, and arcs represent the interactions occurring among such components. We hereby present the μTOSCA type system[1], providing building blocks for such a representation (Fig. 3.1).

Nodes can be services, communication patterns or data stores. A `Service` is a component running some business logic, e.g., a service managing users' orders in an e-commerce application.

[1] `https://di-unipi-socc.github.io/microTOSCA/microTOSCA.yml`.

A `CommunicationPattern` is a component implementing a messaging pattern decoupling the communication among two or more components. Fig. 3.1 contains two communication patterns from [90]: `MessageRouter` (e.g., load balancers, API gateways) and `MessageBroker` (e.g., message queues). Finally, a `DataStore` is a component storing the data pertaining to a certain domain, e.g., a database of orders in an e-commerce application.

Nodes can be interconnected via `InteractsWith` relationships, to model that a source node invokes functionalities offered by a target node. Such relationships can be enriched by setting the boolean properties `circuit_breaker`, `timeout` and `dynamic_discovery`. The first two properties allow to indicate whether the source node is interacting with the target node via a circuit breaker or by setting proper timeouts, to avoid that the source fails/gets stuck waiting for an answer from the target when the latter is unresponsive (e.g., because it failed). Property `dynamic_discovery` allows to specify whether the endpoint of the target of the interaction is dynamically discovered (e.g., by exploiting a discovery service).

Nodes can also be placed in an `Edge` group, to define the subset of application components directly accessed from outside of the application.

Formally, the architectures represented with μTOSCA are triples, whose elements are (i) the typed nodes and (ii) the relationships forming the graph representing the architecture of an application, and (iii) the group of nodes defining the edge of the architecture.

Definition 3.1 (Architecture)**.** The *architecture* of an application is represented by a triple $A = \langle N, R, E \rangle$, where

(i) N is a finite set of typed nodes representing application components,

(ii) R is a finite multiset[2] of pairs of nodes representing the relationships occurring among application components, and

(iii) $E \subseteq N$ is a non-empty set of nodes defining the edge of the architecture.

Def. 3.1 allows to describe an architecture where (a) a node interacts with itself. It also allows to specify that (b) a data store is invoking functionalities offered by another component or being accessed by something different from a service internal to the application, Finally, Def. 3.1 allows to indicate that (c) a message broker is invoking functionalities offered by other components or that no component is placing messages in a broker, and that (d) a message router is not routing messages towards other components or that it is never invoked. To avoid such undesirable situations, we hereafter consider an architecture to be *well-formed* when none of cases (a-d) is occurring.

[2]Multiple relations from component x to component y indicate that x interacts with y in different ways (e.g., directly in one case, via a circuit breaker in another case).

Notation 3.1 (Types)**.** *We write* $x.\mathsf{type}$ *to denote the* type *of a node* x*, and we write* ◯, mB, mR *and* ⛁ *to visually denote the µTOSCA types* `Service`*,* `MessageBroker`*,* `MessageRouter` *and* `DataStore`*, respectively. Given two types* t *and* t'*, we also write* $t \geq t'$ *iff* t *extends or is equal to* t'*.*

Definition 3.2 (Well-formedness)**.** An architecture $A = \langle N, R, E \rangle$ is *well-formed* iff

(a) $\forall \langle x, y \rangle \in R : x \neq y$,

(b) $\forall x \in N : x.\mathsf{type} \geq$ ⛁ $\Rightarrow$
$((\nexists \langle x, y \rangle \in R) \wedge x \notin E \wedge (\forall \langle y, x \rangle \in R : y.\mathsf{type} \geq$ ◯$))$,

(c) $\forall x \in N : x.\mathsf{type} \geq$ mB $\Rightarrow ((\nexists \langle x, y \rangle \in R) \wedge (\exists \langle y, x \rangle \in R))$, and

(d) $\forall x \in N : x.\mathsf{type} \geq$ mR $\Rightarrow ((\exists \langle x, y \rangle \in R) \wedge (x \in E \vee \exists \langle y, x \rangle \in R))$.

We hereafter assume architectures to be well-formed.

3.3 Discovering and resolving architectural smells

The architectural smells violating the *horizontal scalability*, *isolation of failures* and *decentralisation* of microservice-based applications, as well as the architectural refactorings allowing to resolve them, have been classified in [120]. An excerpt of the resulting taxonomy is reported in Fig. 3.2.

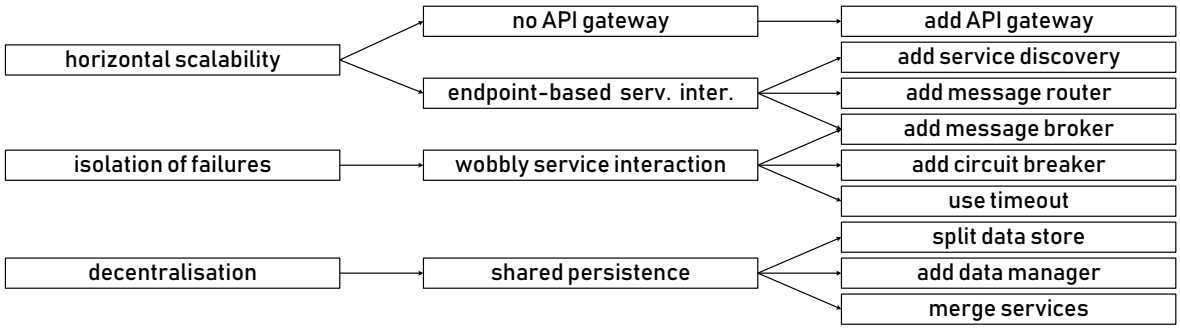


FIGURE 3.2. A taxonomy for (a) design principles of microservices, (b) architectural smells, and (c) architectural refactorings [120]

Starting from such taxonomy, we hereby formalise the conditions allowing to automatically determine the occurrence of smells in architectures modelled with μTOSCA, and we illustrate how to refactor an architecture to resolve each identified smell. In doing so, we exploit the graphical support provided by Fig. 3.3.

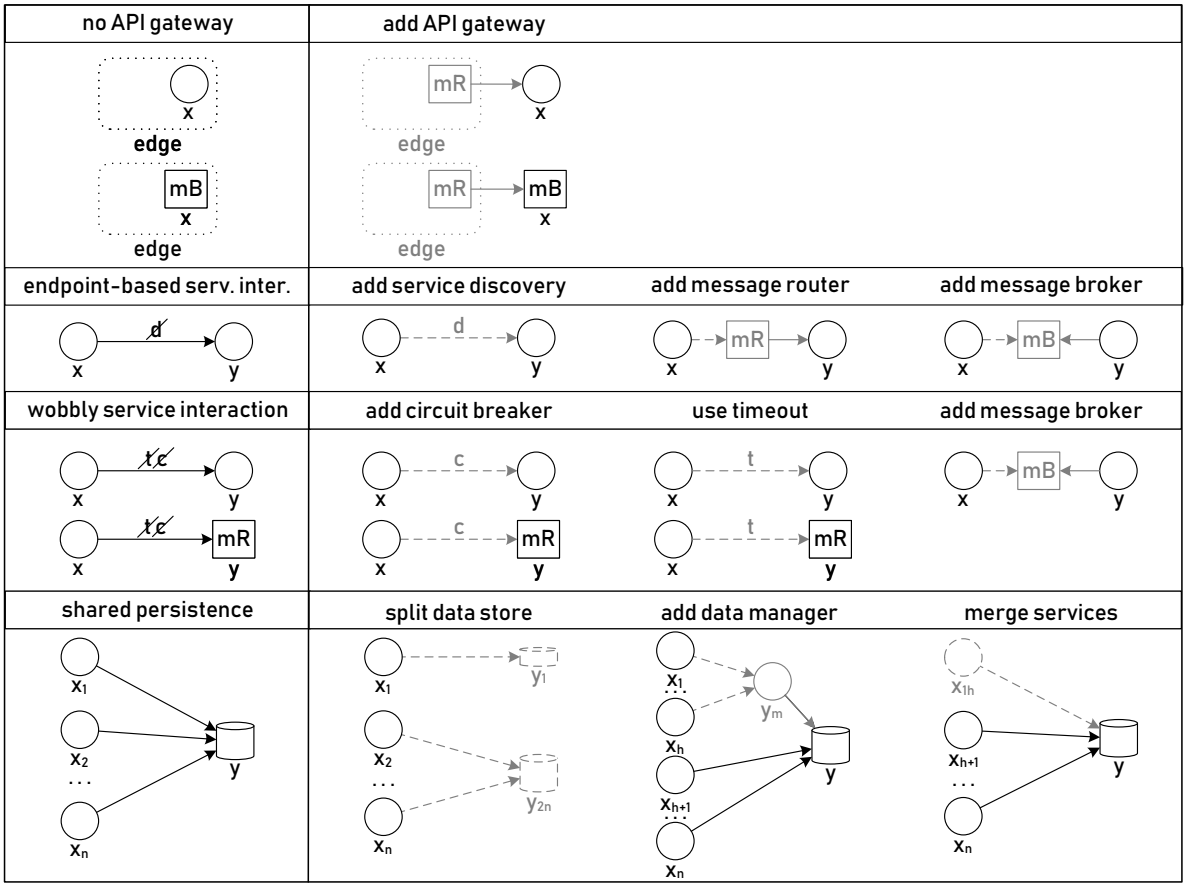


FIGURE 3.3. Visual representation of the architectural smells (left column) and refactorings (right column) in Fig. 3.2, with the Edge group denoted by a dotted line and interactions depicted as arrows. Labels d, c and t represent that properties `dynamic_discovery`, `circuit_breaker` and `timeout` are true, while $\not d$, $\not c$ and $\not t$ represent that they are false. Updates due to refactorings are in grey, with mandatory updates being dashed. Solid grey lines indicate updates that may be implemented by reusing existing components.

3.3.1 Architectural smells possibly violating horizontal scalability.

The possibility of adding/removing replicas of a microservice is a direct consequence of the independent deployability of microservices. To ensure its horizontal scalability, all the replicas of a microservice m should be reachable by the microservices invoking m [91]. In [120], two architectural smells emerged as possibly violating the horizontal scalability of microservices, i.e., NO API GATEWAY and ENDPOINT-BASED SERVICE INTERACTION, which we discuss hereafter.

The NO API GATEWAY smell occurs whenever the external clients of an application directly interact with some internal services. If one of such services is scaled out, the horizontal scalability of microservices may get violated because external clients may still keep invoking the same instance, without reaching any replica.

To identify the occurrence of a NO API GATEWAY smell, we should hence check whether some application components are accessed without passing through an API gateway, i.e., whether the edge of the architecture contains something that is not a message router.

Definition 3.3 (No Api Gateway)**.** Let $A = \langle N, R, E \rangle$ be an architecture. A node $x \in N$ indicates a NO API GATEWAY smell iff

$$x \in E \wedge x.\mathsf{type} \not\geq \boxed{\text{mR}}$$

Fig. 3.3 illustrates the possible NO API GATEWAY smells, due to a component x (either a service or a message broker) being placed at the edge of an architecture. The figure also shows the architectural refactorings resolving the occurrence of NO API GATEWAY smells. In both cases, the refactoring consists in introducing a message router (e.g., a gateway or a load balancer), or reusing one already available in the application. Such a message router will act as an API gateway, hence avoiding x to get directly accessed from outside of the application.

The ENDPOINT-BASED SERVICE INTERACTION smell occurs in an application when a service x directly invokes another service y (e.g., because the location of y is hardcoded in the source code of x, or because no message router is used). If this is the case, when scaling out service y by adding new replicas, these cannot be reached by x, hence only resulting in a waste of resources [120]. Formally, this happens whenever there is a direct interaction from x to y, where x is not using any support for dynamically discovering the actual endpoint of y.

Notation 3.2 (Properties of relations)**.** *Given an architecture $A = \langle N, R, E \rangle$, we write $\langle x, y \rangle.\mathsf{p}$ to denote the property* p *of a relationship $\langle x, y \rangle \in R$.*

Definition 3.4 (Endpoint-based Service Interaction)**.** Let $A = \langle N, R, E \rangle$ be an architecture. A relation $\langle x, y \rangle \in R$ indicates an ENDPOINT-BASED SERVICE INTERACTION smell iff

$$x.\mathsf{type} \geq \bigcirc \wedge y.\mathsf{type} \geq \bigcirc \wedge \langle x, y \rangle.\mathsf{dynamic\_discovery} = \mathsf{false}$$

A visual representation of an ENDPOINT-BASED SERVICE INTERACTION is in Fig. 3.3, where a service x is directly invoking another service y. The figure also illustrates the architectural refactorings allowing to resolve the occurrence of an ENDPOINT-BASED SERVICE INTERACTION smell, all sharing the same goal, i.e., decoupling the interaction between two services by introducing an intermediate integration pattern. Such refactorings predicate only on the value of property `dynamic_discovery` of the relationship outgoing from x.

The most common solution is to add a service discovery mechanism to dynamically resolve the endpoint of the service targeted by the interaction [139]. The other possible solutions instead consist in decoupling the interaction between two services by exploiting a message router or a message broker, respectively. In all cases, the interaction outgoing from x must necessarily be updated, while the message router/broker used to decouple the interaction may also be already available and reused to implement the architectural refactoring.

3.3.2 Architectural smell possibly violating isolation of failures.

Microservice-based architectures should be designed to isolate failures, meaning that each microservice should tolerate the failure of any invocation to the microservices it depends on [108]. In [120], the WOBBLY SERVICE INTERACTION smell emerged as possibly violating the isolation of failures in microservices.

The interaction between two microservices is “wobbly" when a failure in the microservice targeted by the interaction can result in triggering a failure also in the source, potentially starting a cascade of failures [95]. This typically happens when a microservice x is consuming functionalities offered by another microservice (directly or through a message router), and x is not provided with any solution for handling the possibility of the target microservice to fail and be unresponsive, such as a circuit breaker or a timeout.

Definition 3.5 (Wobbly Service Interaction)**.** Let $A = \langle N, R, E \rangle$ be an architecture. A relation $\langle x, y \rangle \in R$ indicates a WOBBLY SERVICE INTERACTION smell iff

$$x.\mathsf{type} \geq \bigcirc \wedge (y.\mathsf{type} \geq \bigcirc \ \vee \ y.\mathsf{type} \geq \boxed{\text{mR}}) \wedge$$
$$\langle x, y \rangle.\mathsf{circuit\_breaker} = \mathsf{false} \wedge \langle x, y \rangle.\mathsf{timeout} = \mathsf{false}.$$

The possible WOBBLY SERVICE INTERACTIONs are illustrated in Fig. 3.3, which shows that such a kind of interactions occurs when a service x is interacting with another service or with a message router (dispatching the messages outgoing from x to other microservice), and such interaction are not equipped with a support for tolerating failures, i.e., no circuit breaker or timeout is used.

Fig. 3.3 also illustrates the architectural refactorings allowing to resolve WOBBLY SERVICE INTERACTION smells. Such refactorings predicate only on the value of the properties `circuit_breaker` and `timeout` of the relationship outgoing from x.

The easiest solutions consist replacing the WOBBLY SERVICE INTERACTION between x and y with one exploiting a circuit breaker to wrap the invocations outgoing from service x or using a timeout. Both solutions allow x not to get stuck waiting for an answer from y. Another possible solution is to decouple the interactions between x and y through a message broker, with the latter being a new one, or one already available in the application. The usage of a broker allows x to send its requests to the broker, with y processing such requests when it is available, hence avoiding x to get stuck or fail when y fails.

It is worth noting that, when x and y are both services, applying the refactoring based on the usage of a message broker allows to also resolve the occurrence of an endpoint-based service interaction smell, if any. At the same time, when x is a service and y is a message router, such a refactoring would not be local to x and y, but rather it would involve acting on the rest of the architecture. It would indeed require to apply a solution like the one for the situation where x and y are both services to *all* services that can be reached through the message router y, by exploiting a single message broker or multiple brokers depending on the actual application needs.

3.3.3 Architectural smell possibly violating decentralisation.

Decentralisation should occur in every aspect of microservice-based architectures, including data management [121]. In this perspective, each data store should be directly accessed by only one service [120]. The SHARED PERSISTENCE smell hence occurs whenever multiple services interact with the same data store y.

Definition 3.6 (Shared Persistence)**.** Let $A = \langle N, R, E \rangle$ be an architecture. A set of relations $R(y) = \{\langle x, y \rangle \in R\}$ indicates a SHARED PERSISTENCE smell iff

$$y.\text{type} \geq \text{⛁} \wedge (\exists \langle x_1, y \rangle, \langle x_2, y \rangle \in R(y) : x_1 \neq x_2).$$

A visual representation of the SHARED PERSISTENCE smell is in Fig. 3.3, where $x_1 \ldots x_n$ are all the services accessing the data store y. The figure also shows the three architectural refactorings for reducing the amount of services accessing the same data store, hence ultimately allowing to resolve the occurrence of a SHARED PERSISTENCE smell. Although their goal is the same, such refactorings are very diverse in spirit, and apply to different situations, highly depending on the services accessing the same data store.

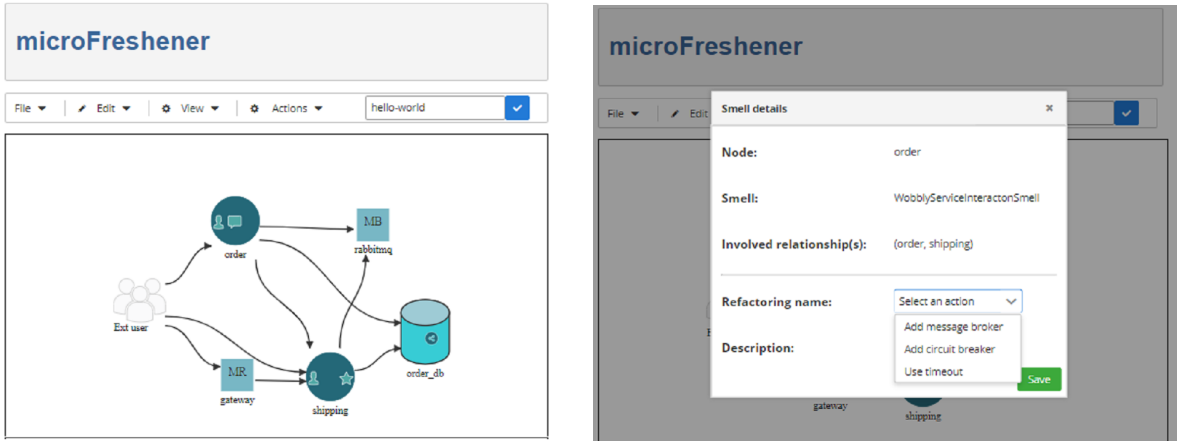
If a service x_1 is the only service accessing a portion of the data stored in the data store y, then y can be split in two different data stores y_1 and y_{2n}, with y_1 only storing the portion of data accessed by x_1, and with y_{2n} storing the rest of the data. The service x_1 then becomes the only accessing y_1, while y_{2n} is accessed by the other services $x_2 \ldots x_n$.

Other possible solutions to reduce the amount of services accessing the same data store y are exploiting a data manager or merging some of the services accessing the data store. Exploiting a data manager consists in adding a service y_m, or reusing one already available, to proxy the access of services $x_1 \ldots x_h$ (with $h \leq n$) to the data store y. The other refactoring instead consists in merging the services $x_1 \ldots x_h$ (with $h \leq n$) into a single service x_{1h}. The rationale behind this last refactoring is that, when multiple services access the same data store, this may be indicating that the application has been split too much, by obtaining too fine-grained services processing the same data [151].

3.3.4 Some important remarks.

Our approach focuses on the *architecture* of an application, by identifying smells based on the interactions among the components forming an application, and by suggesting refactorings of the architecture itself. The concrete implementation of an architectural refactoring (i.e., the actual updates of the application sources) are left to the application owner, in a similar way as the concrete implementation of a design pattern is left to developers. Hence, the application owner can decide which refactoring to apply also based on the cost for actually implementing it.

Also, the architectural smells discussed in this section indicate *potential* violations of design principles of microservices. This means that the occurrence of an architectural smell does not mean that a design principle is necessarily violated, hence not necessarily needing to be resolved



(a) (b)

FIGURE 3.4. Snapshots of (a) the editing/analysis and (b) refactoring views of μFRESHENER.

by applying a corresponding architectural refactoring. Even if an architectural smell is denoting an actual violation of a design principle, the application owner may still decide to not apply any refactoring as well, e.g., because updating the application sources in accordance with an architectural refactoring is too expensive. Another possible reason for choosing to not resolve an architectural smell can be that an application architect intentionally structured the corresponding part of the application as that, due to some contextual requirement. For instance, she may have decided to share a same data store among multiple services for overriding reasons. If this is the case, she will ignore the corresponding shared persistence smell, otherwise she would break her contextual requirements.

In any case, manually identifying architectural smells, deciding whether to resolve them and which refactoring to apply is not easy. It would be helpful to have a support system automatically identifying the smells affecting the architecture of an application and allowing to explore among multiple possible refactorings to resolve them. One such support system is presented in the following section.

3.4 μFRESHENER: A prototype implementation

To illustrate the feasibility of our approach and support the design of microservice-based applications, we implemented a prototype tool (called μFRESHENER) publicly available on GitHub[3]. μFRESHENER provides a web-based graphical user interface for (i) editing μTOSCA specifications, (ii) automatically identifying architectural smells in specified applications and (iii) exploring/applying architectural refactorings for resolving the identified smells.

[3] https://github.com/di-unipi-socc/microFreshener.

Fig. 3.4 provides two snapshots of the GUI of μFRESHENER. Fig. 3.4(a) shows the editing and analysis view, where one can add/remove nodes and relationships from an architecture, and where automatically identified smells are displayed with icons on top of corresponding nodes. By clicking on one of such icons, one can open the view in Fig. 3.4(b), which permits selecting the architectural refactoring to apply to resolve the selected smell. Once selected, the architecture modelling is updated in accordance with the architectural refactoring. Note that the refactoring is only applied to the μTOSCA specification of an application (and not on its sources), and that one can go back and forth in architectural refactorings, by undoing/redoing them by clicking on the corresponding buttons.

3.5 Related work

Even if there exists studies classifying various architectural smells for microservices (e.g., [41, 120, 151]), to the best of our knowledge, ours is the first systematic approach for identifying *and* resolving architectural smells possibly violating the design principles of microservices in existing microservice-based applications.

[13] and [82] report on design patterns and decision models to design microservice-based applications (from scratch, or by migrating from monoliths to microservices). [13] also illustrates different possible solutions to resolve potential design issues. In both cases, this is done by retrieving information from practitioners or industry-scale projects, and by organising such information in informal guidelines, which can be used for driving the design of microservice-based application. Our approach tries to further support the design of microservices, by providing a systematic solution to identify and resolve the architectural smells affecting an already existing application, also providing a tool support.

Systematic solutions for modelling and analysing microservice-based architectures anyway exist, even if conceived for different purposes. For instance, [155] presents *MicroDSL*, a domain-specific language for modelling microservice-based architectures, where microservices interact through RESTish protocols, and models are then used to generate an executable deployments of specified applications. [37] instead proposes a Petri net-based solution for the runtime verification of the orchestration of microservice-based application on top of Netflix's Conductor. Even if conceived for different purposes, [155] and [37] share our baseline idea of elicitating all interactions among microservices to analyse an architecture. They however differ in the goals of the proposed approachs, due to which they focus on modelling specific types of applications (microservices interacting with REST in [155], Conductor-based applications in [37]).

[38] and [48] present two other existing tools for analysing of microservices, which exploit a modelling closer to ours. Indeed, they both model a microservice-based architecture as a graph, whose nodes represent components and whose arcs represent interactions. They however do not support modelling the edge of an architecture, nor distinguishing whether a component is

a service, communication pattern or a data store. This, along with our willingness to exploit a standard to model microservice-based architectures, is the reason why developed μTOSCA instead of reusing the modelling in [38] or [48].

Another tool worth mentioning is [144], which provides an approach to automate the testing of microservice-based applications. [144] relates to our approach as it allow to systematically check whether the interfaces of running microservices adhere to a given specification. It however requires to run the microservice-based application to be tested, while ours is a design-time support not needing to actually run an application.

[70] and [143] instead present solutions for detecting smells in the design of a single service (specified in UML and ARCHERY, respectively). [9], [67] and [159] focus on identifying smells in the structuring of the sources of a service, and propose refactorings for resolving detected smells. All such approaches however differ from ours, as they focus on the design of a single service, while our approach focuses on the architectural smells due to the interactions among all components forming a microservice-based application. In other words, such approaches and ours can complement each other, to permit identifying both the architectural smells affecting a single service and those due to the interactions among the components in an application.

Similar considerations apply to [84] and [83]. Even if with different approaches (self-adaptation in [84], aspect-oriented ambients in [83]), they both focus on analysing a single microservice to determine whether the its granularity is optimal, or whether it needs some adaptation to rightsize its granularity. We instead focus on analysing the interactions among all microservices forming an application to identify and resolve architectural smells.

Finally, it is also worth relating our work with [76], [77], [112] and [113]. Starting from the idea that service interactions are the main mechanism to program the microservices forming an application, [77] proposes to develop microservice-based applications with Jolie, a language for developing service compositions by programming their interactions. Our approach follows the same idea, as we consider service interactions as the basis for identifying architectural smells.

[76], [112] and [113] instead propose different solutions for microservice-based architecture recovery, i.e., identifying the microservices forming an application and the interactions among them. [76] and [112] also show how automatically recovered architectures can be analysed for identifying issues (i.e., unnecessary service interactions in [76], dependency cycles in [112]). [76], [112] and [113] could then be used in conjunction with our approach, to first derive the architecture of a microservice-based application, and then identify and resolve the architectural smells affecting such application.

3.6 Conclusions

We have presented a methodology to identify the architectural smells possibly violating design principles of microservices, and to apply architectural refactorings to resolve them. We have also

presented the μFRESHENER, implementing our methodology to support the design of microservice-based applications.

While our methodology and the μFRESHENER prototype can be actually applied to analyse and improve existing microservice-based applications, users must currently define (with the GUI of μFRESHENER) or provide a μTOSCA description of the architecture of their applications. To increase the usability of our prototype, we plan to develop plug-ins to automatedly extract the μTOSCA description of the architecture of an application (from its code structure like in [85, 112, 156] and/or from its runtime behaviour like in [76, 113]).

We also plan to extend the architectural smells that can be identified and resolved with our methodology (and with μFRESHENER), by starting from the smells and refactoring classified in [41, 120, 151]. As a concrete example, we plan to extend μTOSCA with a type for grouping nodes to represent team assignment (i.e., which components are assigned to which team), to formalise the team-related architectural smells available in [120], to correspondingly extend μFRESHENER to identify and resolve such smells, and to feature team-wise usage of μFRESHENER.

Finally, we plan to extend μFRESHENER to account for the container orchestrator (e.g., Docker Compose, Kubernetes) used to deploy the application, as the container orchestration layer can resolve some smells possibly present at the architecture layer. We also intend to validate the effectiveness of our methodology and the usability μFRESHENER against real-world microservice-based applications (possibly involving hundreds of interconnected services).

Part III

Deploying microservices

CHAPTER 4

ORCHESTRATING INCOMPLETE TOSCA APPLICATIONS WITH DOCKER

Cloud applications typically consist of multiple interacting components, each requiring a virtualised runtime environment providing the needed software support (e.g., operating system, libraries). In this chapter ,we show how TOSCA and Docker can be effectively exploited to orchestrate multi-component applications, even if their (runtime) specification is incomplete. More precisely, we first propose a TOSCA-based representation for multi-component applications, and we show how to use it to specify only the components forming an application. We then present a way to automatically complete TOSCA application specifications, by discovering Docker-based runtime environments that provide the software support needed by the application components. We also discuss how the obtained specifications can be automatically orchestrated by existing TOSCA engines.

The results illustrated in this Chapter were published in [26], which appeared in the journal "Science of Computer Programming".

4.1 Introduction

Cloud computing permits running on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [10]. This has revolutionised the way applications are built in the IT industry, where monoliths are giving way to distributed, component-based architectures. Modern cloud applications typically consist of multiple interacting components, which (compared to monoliths) permit better capitalising the benefits of cloud computing [35].

At the same time, the need for orchestrating the management of multi-component applications across heterogeneous cloud platforms has emerged [23, 109]. The deployment, configuration, en-

actment and termination of the components forming an application must be suitably orchestrated. This must be done by considering all the dependencies occurring among the components forming an application, as well as the fact that each application component must run in a virtualised environment providing the software support it needs [66].

Developers and operators are currently required to manually select and configure an appropriate runtime environment for each application component, and to explicitly describe how to orchestrate such components on top of the selected environments [121]. As we discuss in Sect. 4.2, such process must then be manually repeated whenever a developer wishes to modify the virtual environment actually used to run an application component, e.g., because the latter has been updated and it now needs additional software support.

The current support for developing cloud applications should be enhanced. In particular, developers should be required to describe only the components forming an application, the dependencies occurring among such components, and the software support needed by each component [17]. Such description should be fed to tools capable of automatically selecting and configuring an appropriate runtime environment for each application component, and of automatically orchestrating the application management on top of the selected runtime environments. Such tools should also allow developers to change the virtual environment running an application component whenever they wish (e.g., by automatically replacing a previously selected environment with another satisfying the new/updated requirements of an application component).

In this chapter, we present a solution geared towards providing such an enhanced support. Our solution is based on TOSCA [124], the OASIS standard for orchestrating cloud applications, and on Docker, the de-facto standard for cloud container virtualisation [127]. The main contributions of this chapter are indeed the following:

- We propose a TOSCA-based representation for multi-component applications, which can be used to specify the components forming an application, the dependencies occurring among them, and the software support that each component requires to effectively run.
- We present TOSKERISER, a tool that automatically completes TOSCA application specifications, by discovering and including Docker-based runtime environments providing the software support needed by the application components. The tool also permits changing –when/if needed– the runtime environment used to host a component.

The obtained application specifications can then be processed by orchestration engines supporting TOSCA and Docker (such as TOSKER [30], for instance). Such engines will automatically orchestrate the deployment and management of the corresponding applications on top of the given runtime environments.

This chapter extends [25] by (a) extending the approach of [25] to permit hosting groups of software components on the same Docker container, by (b) providing a detailed description of the implementation of TOSKERISER, and by (c) presenting two novel case studies comparing the

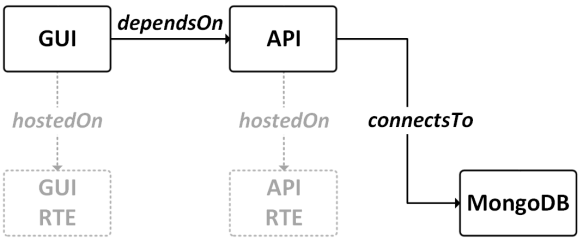


FIGURE 4.1. Running example: The application *Thinking*.

orchestration of the management of applications with and without our solution (based on three KPIs) and illustrating the usefulness of groups.

The rest of the chapter is organised as follows. Sect. 4.2 illustrates an example further motivating the need for an enhanced support for orchestrating the management of cloud applications. Sect. 4.3 provides some background on TOSCA and Docker. Sect. 4.4 shows how to specify application-specific components only, with TOSCA. Sect. 4.5 then presents our tool to automatically determine appropriate Docker-based environments for hosting the components of an application. Sect. 4.6 illustrates the two case studies, while Sects. 4.7 and 4.8 discuss related work and draw some concluding remarks, respectively.

4.2 Motivating scenario

Consider the open-source web-based application *Thinking*[1], which allows its users to share their thoughts, so that all other users can read them. *Thinking* is composed by three interconnected components (Fig. 4.1), namely (i) a *MongoDB* storing the collection of thoughts shared by end-users, (ii) a Java-based REST *API* to remotely access the database of shared thoughts, and (iii) a web-based *GUI* visualising all shared thoughts and allowing to insert new thoughts into the database. As indicated in the documentation of the *Thinking* application:

(i) The *MongoDB* component can be obtained by directly instantiating a standalone Docker-based service, such as `mongo`[2], for instance.

(ii) The *API* component must be hosted on a virtualised environment supporting maven (version 3), java (version 1.8) and git (any version). The *API* must also be connected to the *MongoDB*.

(iii) The *GUI* component must be hosted on a virtualised environment supporting nodejs (version 6), npm (version 3) and git (any version). The *GUI* also depends on the availability of the *API* to properly work (as it sends GET/POST requests to the *API* to retrieve/add shared thoughts).

[1] `https://github.com/di-unipi-socc/thinking`.
[2] `https://hub.docker.com/_/mongo/`.

Docker containers work as virtualised environments for running application components [127]. However, we currently have to manually look for the Docker containers offering the software support needed by *API* and *GUI* (or to manually extend existing containers to include such support). We then have to manually package the *API* and *GUI* components within such Docker containers, and to explicitly describe the orchestration of the management ofall the Docker containers in our application. In other words, we must identify, develop, configure and orchestrate the deployment and management of all components in Fig. 4.1, including those not specific to the *Thinking* application (viz., the lighter nodes *API RTE* and *GUI RTE*).

The above process must be manually repeated whenever we wish to change the Docker containers used to run the components of *Thinking*. Suppose, for instance, that we wish to host *GUI* and *API* on the same container. We should remove their containers from the application, we should manually look for a new container providing the software support needed by both components, and we should re-describe — possibly from scratch — the orchestration of *GUI* and *API* on the newly added container.

Especially in the latter case, our effort would be lower if we were provided with a support requiring us to describe our application only, and automating all remaining tasks. More precisely, we should only be required to specify the thicker nodes and dependencies in Fig. 4.1. The support should then be able to automatically complete our specification, and to exploit the obtained specification to automatically orchestrate the deployment and management of the *Thinking* application. In this chapter, we show a TOSCA-based solution geared towards providing such a support.

4.3 Background

4.3.1 TOSCA

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [124]) is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based and machine-readable modelling language that permits describing cloud applications. Obtained specifications can then be processed to automate the deployment and management of the specified applications. We hereby report only those features of the TOSCA modelling language that are used in this chapter[3].

TOSCA permits specifying a cloud application as a service template, which is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 4.2). The topology template is essentially a typed directed graph, which describes the topological structure of a multi-component cloud application. Its nodes (called node templates) model the application

[3]A more detailed, self-contained introduction to TOSCA can be found in [17, 34].

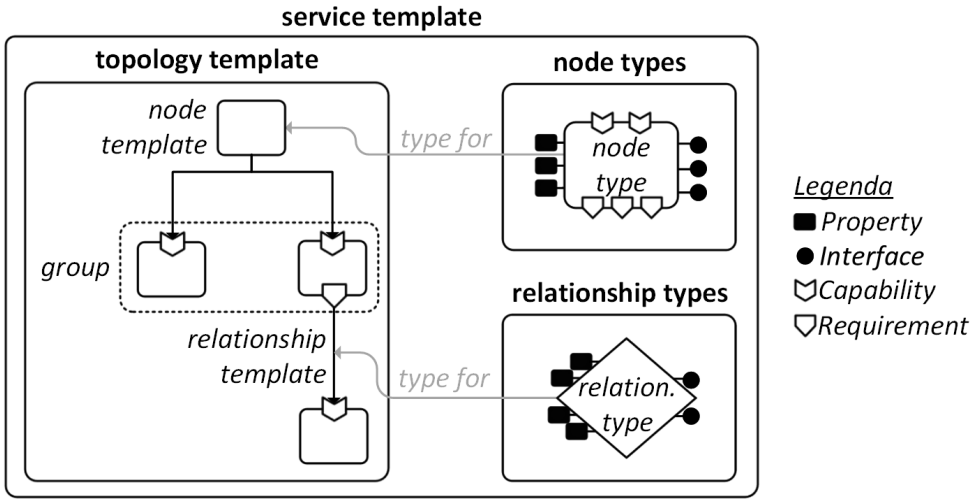


FIGURE 4.2. The TOSCA metamodel [124].

components, while its edges (called relationship templates) model the relations occurring among such components.

Node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components' requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components. As the TOSCA type system supports inheritance, a node/relationship type can be defined by extending another, hence permitting the former to inherit the latter's properties, requirements, capabilities, interfaces, and operations (if any).

Node templates can also be logically grouped, typically to define groups of nodes to be managed together, and/or to uniformly apply the same management policy to all the nodes forming a group (e.g., placing all nodes in a group on the same host, simultaneously scaling all the nodes forming of a group). A TOSCA group represents a logical grouping of node templates that need to be orchestrated together to achieve some management goal. As such goals can be many, the actual purpose of each group is specified by means of its group type.

To concretely realise the deployment and management of the nodes forming an application, node templates and relationship templates also specify the artifacts needed to actually perform their deployment or to implement their management operations. As TOSCA allows artifacts to represent contents of any type (e.g., scripts, executables, images, configuration files, etc.), the metadata needed to properly access and process them is described by means of artifact types.

TOSCA applications are then packaged and distributed in so-called CSARs (*Cloud Service ARchives*). A CSAR is essentially a zip archive containing an application specification along with

the concrete artifacts realising the deployment and management operations of its components.

4.3.2 Docker

Docker (`https://docker.com`) is a Linux-based platform for developing, shipping, and running applications through container-based virtualisation. Container-based virtualisation [148] exploits the kernel of the operating system of a host to run multiple isolated user-space instances, called *containers*.

Each Docker container packages the applications to run, along with whatever software support they need (e.g., libraries, binaries, etc.). Containers are built by instantiating so-called Docker *images*, which can be seen as read-only templates providing all instructions needed for creating and configuring a container. Docker images can be created by developing *Dockerfiles*, which contain all the commands to be executed to create an image (e.g., installing the needed support, setting the main process to run). Existing Docker images are distributed through so-called Docker *registries* (e.g., Docker Hub — `https://hub.docker.com`), and new images can be built by extending existing ones.

Docker containers are volatile, and the data produced by a container is (by default) lost when the container is stopped. This is why Docker introduces *volumes*, which are specially-designated directories (within one or more containers) whose purpose is to persist data, independently of the lifecycle of the containers mounting them. Docker never automatically deletes volumes when a container is removed, nor does it remove volumes that are no longer referenced by any container.

Docker also allows containers to intercommunicate, by creating virtual networks, which span from bridge networks (for single hosts), to complex overlay networks (for clusters of hosts). Docker also provides built-in orchestration tools, such as Docker Compose (`https://docs.docker.com/compose/`), which permits creating multi-container Docker applications, and managing them on a single host or in a cluster of hosts[4].

4.4 Specifying applications only, with TOSCA

Multi-component applications typically integrate various and heterogeneous software components [66]. We hereby propose a TOSCA-based representation for such components (Sect. 4.4.1). We also illustrate how it can be used to specify only the components that are specific to an application, and to constrain the Docker containers that can be used to actually host such components (Sect. 4.4.2).

[4] A more detailed introduction to Docker can be found in [114, 145].

4.4.1 A TOSCA-based representation for applications

We first define three different TOSCA node types[5] to distinguish Docker containers, Docker volumes, and software components that can be used to build a multi-component application (Fig. 4.3).

tosker.nodes.Container permits representing Docker containers, by indicating whether a container requires a *connection* (to another Docker container or to an application component), whether it has a generic *dependency* on another node in the topology, or whether it needs some persistent *storage* (hence requiring to be attached to a Docker volume). *tosker.nodes.Container* also permits indicating whether a container can *host* an application component, whether it offers an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of another container or application component). It also lists the operations to manage a container (which correspond to the basic operations offered by Docker [114]).

To complete the description, *tosker.nodes.Container* provides placeholder properties for specifying port mappings (*ports*) and the environment variables (*env_variables*) to be configured in a running instance of the corresponding Docker container. It also provides two properties (*supported_sw* and *os_distribution*) for indicating the software support provided by the corresponding Docker container and the operating system distribution it runs.

The above listed elements are all optional, viz., node templates of type *tosker.nodes.Container* can optionally instantiate/implement them. Additionally, requirements and capabilities can be instantiated multiple times in a node of type *tosker.nodes.Container* (e.g., if a container requires two distinct connections to two different components, two requirements *connection* have to be instantiated).

tosker.nodes.Volume permits specifying Docker volumes, and it defines an optional capability *attachment* to indicate that a Docker volume can be used to satisfy the *storage* requirements of Docker containers. It also lists the operations to manage a Docker volume (which corresponds to the basic operations offered by the Docker platform [114]).

tosker.nodes.Software permits describing the software components forming a multi-component application. It permits specifying whether an application component requires a *connection* (to a Docker container or to another application component), whether it has a generic *dependency* on another node in the topology, and that it has to be *host*ed on a Docker container or on another component *tosker.nodes.Software* also permits indicating whether an application component can *host* another component, whether it provides an *endpoint* where to connect to, or whether it offers some *feature* (to satisfy a generic *dependency* requirement of a container/application component). Finally, *tosker.nodes.Software* indicates

[5]Their actual TOSCA definition is publicly available at `https://github.com/di-unipi-socc/tosker-types`.

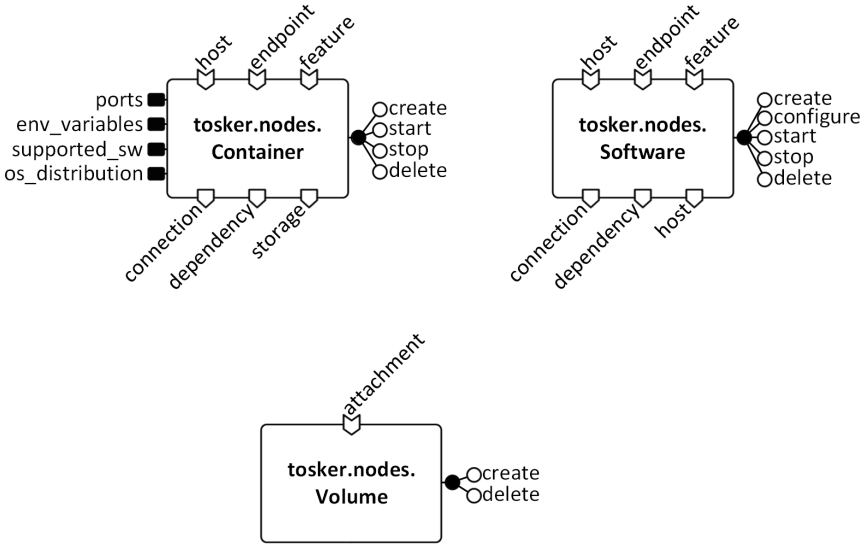


FIGURE 4.3. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

the operations to manage an application component (viz., *create*, *configure*, *start*, *stop*, *delete*).

All above listed elements are optional, as node templates of type *tosker.nodes.Software* can optionally instantiate them. Requirements and capabilities can also be instantiate multiple times in a node of type *tosker.nodes.Software* (e.g., two instances of the requirement *connection* permits indicating that a component requires two distinct connections to two different components).

The interconnections and interdependencies among the nodes forming a multi-component application can then be indicated by exploiting the TOSCA normative relationship types [124]. Namely, *tosca.relationships.AttachesTo* can be used to attach a Docker volume to a Docker container, *tosca.relationships.ConnectsTo* can indicate interconnections between Docker containers and/or application components, *tosca.relationships.HostedOn* can be used to indicate that an application component is hosted on another component or on a Docker container, and *tosca.relationships.DependsOn* can be used to indicate generic dependencies between the nodes of a multi-component application[6].

4.4.2 Specifying application-specific components only

The TOSCA types introduced in the previous section can be used to specify the topology of a multi-component application. We hereby illustrate, by means of an example, how to specify in

[6]The TOSCA specification [124] explains how to validly instantiate normative relationship types.

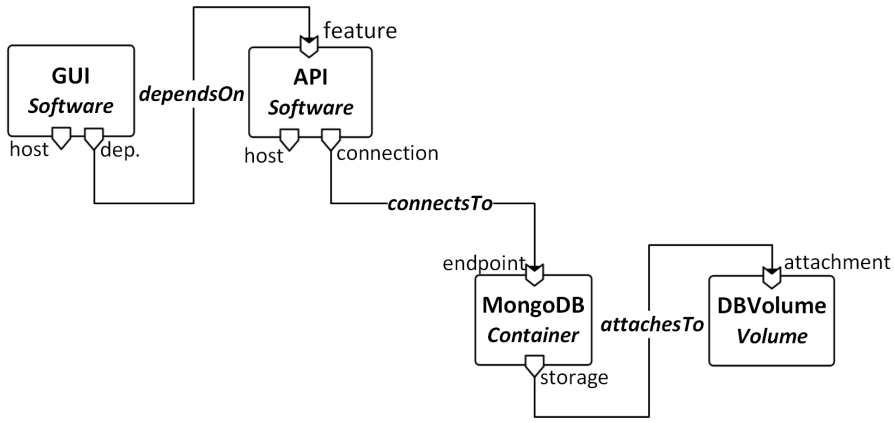


FIGURE 4.4. A specification of our running example in TOSCA (where nodes are typed with *tosker.nodes.Container*, *tosker.nodes.Volume*, or *tosker.nodes.Software*, while relationships are typed with TOSCA normative types [124]).

TOSCA only the fragment of a topology that is specific to an application (by also constraining the Docker containers that can be used to actually host the components in such fragment).

Example 4.1. *Consider again the application* Thinking *in our motivating scenario (Sect. 4.2). The components specific to* Thinking *(viz.,* MongoDB*,* API*, and* GUI*) can be specified in TOSCA as illustrated in Fig. 4.4:*

- MongoDB *is obtained by directly instantiating a Docker container* `mongo` *(modelled as a node of type* tosker.nodes.Container*). The latter is attached to a Docker volume where the shared thoughts will be persistently stored.*
- API *is a software component (viz., a node of type* tosker.nodes.Software*).* API *requires to be connected to the back-end* MongoDB*, to remotely access the database of shared thoughts.*
- GUI *is a software component (viz., a node of type* tosker.nodes.Software*).* GUI *depends on the availability of* API *to properly work (as it sends HTTP requests to the* API *to retrieve/add shared thoughts).*

Please note that the requirements host *of both* API *and* GUI *are left pending (viz., there is no node satisfying them). This is because the actual runtime environment of* API *and* GUI *is not specific to the application* Thinking*, and it should be automatically determined among the many possible (as we will discuss in Sect. 4.5). The only effort required to the developer is to specify constraints on the configuration of the Docker containers that can effectively host* API *and* GUI *(e.g., which software support they have to provide, which operating system distribution they must run, which port mappings they must expose, etc.).*

```
node_filter:
 type: tosker.nodes.Container
 properties:
  - supported_sw:
    - mvn: 3.x
    - java: 1.8.x
    - git: x
  - ports:
    - 8080: 8000
  - os_distribution: ubuntu
```

(a)

```
node_filter:
  type: tosker.nodes.Container
  properties:
    - supported_sw:
      - node: 6.x
      - npm: 3.x
      - git: x
    - ports:
      - 3000: 8080
```

(b)

FIGURE 4.5. Constraints on the Docker containers that can effectively run the software components (a) *API* and (b) *GUI* (specified within their requirements *host*).

TOSCA natively supports the possibility of expressing constraints on the nodes that can satisfy requirements left pending [124], through the clause `node_filter` that can be indicated within a requirement. `node_filter` permits specifying the type of a node that can satisfy a requirement, and it permits constraining the properties of such node.

We can hence exploit `node_filter` to indicate that the software components in an application must be hosted on Docker containers (viz., on nodes of type *tosker.nodes.Container*). We can also indicate constraints to configure such containers (e.g., which port mappings they must expose, or which environment variables they should define), to define the operating system distribution they must run, and to indicate the software distributions they must support. The latter can be indicated with pairs *name: version*, where *version* indicates the prefix number of the desired software version followed by an *x* (e.g., `java: 1.8.x` is an alias for all versions of java starting with 1.8).

Example 4.2. *Consider again the multi-component application* Thinking*, modelled in TOSCA as in Fig. 4.4. The pending requirements* host *of* API *and* GUI *must constrain the nodes that can actually satisfy them.*

The requirement host *of* API *can express the constraints on the Docker containers that can effectively host it with the* `node_filter` *in Fig. 4.5.(a). The latter indicates that* API *needs to run on a Docker container, viz., a node of type* tosker.nodes.Container*, which supports maven (version 3), java (version 1.8) and git (any version). It also indicates a port mapping to be configured in the hosting container and that such container must be based on an Ubuntu distribution*[7].

Analogously, the requirement host *of* GUI *can constrain the Docker containers for hosting it with the* `node_filter` *in Fig. 4.5.(b). The latter prescribes that* GUI *must run on a Docker*

[7]Constraining the operating system distribution is particularly useful when the artifacts implementing the management operations of a software component require to perform distribution-specific system calls (e.g., a *.sh* script performing a command `apt-get`, which is supported only in Debian-based distributions).

container supporting node (version 6), npm (version 3) and git (any version). It also requires the hosting container to expose the indicated port mapping. The obtained (incomplete) TOSCA specification is publicly available on GitHub[8].

4.4.3 Specifying groups of components to be hosted on the same container

An application developer may also wish to group some of the components forming her application, and to host all the nodes forming a group in the same container. This would allow, for instance, to reduce the network traffic produced by the components of an application.

TOSCA natively permits grouping the nodes forming an application in groups, and it allows specifying the actual purpose of each group by means of its type [124]. We hence defined a new group type *tosker.groups.DeploymentUnit*, whose purpose is precisely to indicate that the nodes it contains must all be hosted on the same container. Given the nature of *tosker.groups.DeploymentUnit*, the following conditions must be satisfied while defining a group of such type:

(i) A group of type *tosker.groups.DeploymentUnit* can only contain nodes of type *tosker.nodes.Software*.

(ii) If the requirement *host* of a node within a group of type *tosker.groups.DeploymentUnit* is satisfied, then such requirement must be satisfied by another node within the same group or by a node of type *tosker.nodes.Container*.

(iii) If a node within a group of type *tosker.groups.DeploymentUnit* satisfies the requirement *host* of another node, then the latter node must be part of the same group.

(iv) The groups of type *tosker.groups.DeploymentUnit* in a TOSCA application specification must be all disjoint (viz., a node cannot be simultaneously part of two different groups).

The first condition is due to the fact that, according to Sect. 4.4.1, nodes of type *tosker.nodes.Software* can be hosted on other nodes, while *tosker.nodes.Container* and *tosker.nodes.Volume* cannot be hosted on other nodes. The second, third and last conditions instead ensure that, whenever a software component is hosted on another software component, then both components are deployed within the same Docker container.

Example 4.3. *Consider again the application* Thinking *in our motivating scenario (Sect. 4.2). Example 4.1 showed how to specify the components forming such application in TOSCA, while Example 4.2 illustrated how to indicate constraints on the Docker containers that can effectively run its software components* API *and* GUI.

[8] `https://github.com/di-unipi-socc/TosKeriser/blob/master/data/examples/thinking-app/thinking/thinking.yaml`.

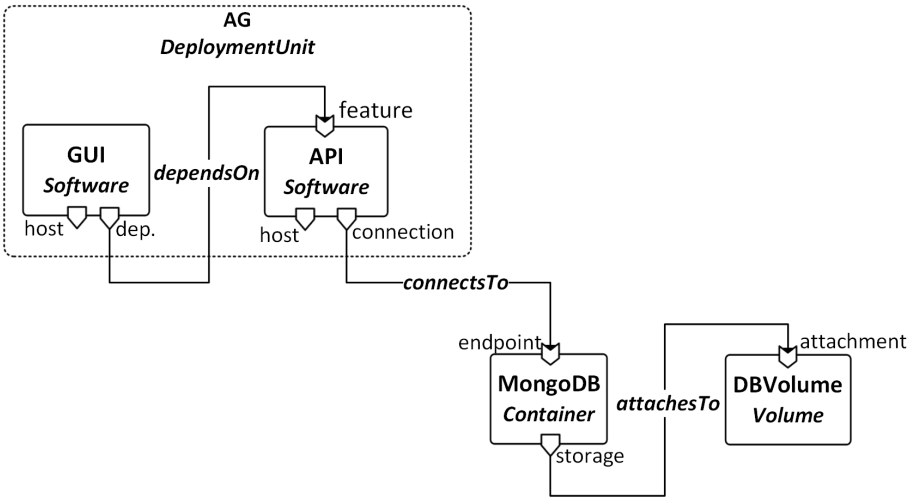


FIGURE 4.6. A specification of our running example in TOSCA, including a group (of type *tosker.groups.DeploymentUnit*) specifying that *API* and *GUI* must be hosted by the same Docker container.

Suppose now that we wish to host both API *and* GUI *on the same container. This can be constrained by just indicating that* API *and* GUI *form a group of type* tosker.groups.DeploymentUnit *(Fig. 4.6 — the corresponding TOSCA specification is publicly available on GitHub*[9]*).*

The tool used to complete the specification of Thinking *will then have to automatically determine a Docker container capable of satisfying the requirements* host *of both* API *and* GUI *(Fig. 4.5).*

4.5 Completing TOSCA specifications, with Docker

We hereby present TOSKERISER, an open-source prototype tool[10] that automatically completes "incomplete" TOSCA application specifications (describing only application-specific components, and indicating constraints on the Docker containers that can be used to host such components — as discussed in the previous section).

TOSKERISER is part of an open-source toolchain allowing to orchestrate multi-component applications with TOSCA and Docker (Fig. 4.7). TOSKERISER inputs a CSAR file containing a TOSCA application specification. It then identifies the set of software components whose requirement *host* has to be fulfilled, and it exploits DOCKERFINDER[11] to identify the Docker

[9] `https://github.com/di-unipi-socc/TosKeriser/blob/master/data/examples/thinking-app/thinking_group/thinking_group.yaml`.

[10] The Python sources of TOSKERISER are publicly available on GitHub at `https://github.com/di-unipi-socc/toskeriser` (under MIT license). TOSKERISER is also available on PyPI, and it can be directly installed on Linux hosts by executing the command `pip install toskeriser`.

[11] DOCKERFINDER [27] is a tool allowing to search for Docker containers based on multiple attributes, including the distributions of software they support and the operating system they are based on.

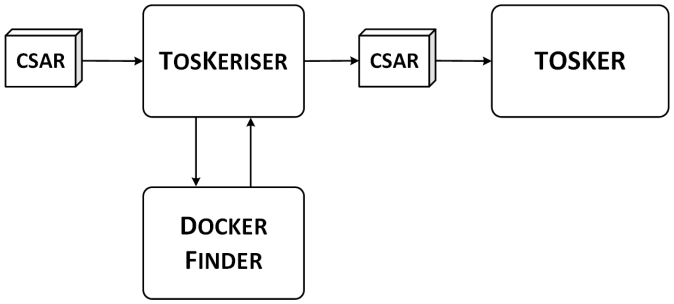


FIGURE 4.7. Open-source toolchain for orchestrating multi-component applications with TOSCA and Docker.

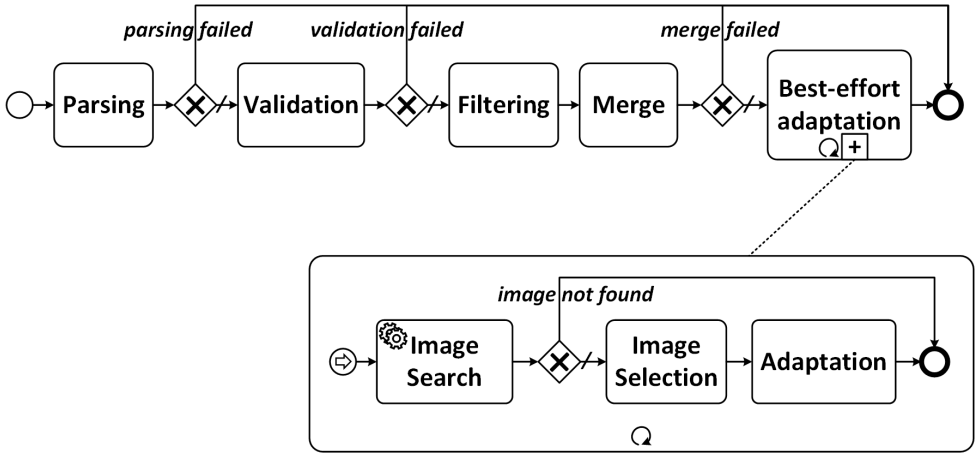


FIGURE 4.8. BPMN modelling of the process that TOSKERISER performs to automatically complete TOSCA application specifications.

containers providing the support needed by such components. TOSKERISER then completes the application topology by properly including the discovered containers, and it outputs the resulting CSAR file. Such file can then be passed to TOSKER [30] (or to any other orchestration engine offering the needed support for TOSCA and Docker), which will automatically deploy and manage the actual instances of the specified application.

We hereafter first detail how TOSKERISER concretely proceeds for automatically completing TOSCA application specifications (Sect. 4.5.1), and we then show how to use TOSKERISER in practice (Sect. 4.5.2).

4.5.1 How TOSKERISER completes applications, concretely

TOSKERISER completes TOSCA application specifications according to the workflow illustrated in Fig. 4.8.

Parsing. TOSKERISER initially parses and validates the TOSCA application specification con-

tained in the CSAR given as input. More precisely, the step *Parsing* first exploits the OpenStack TOSCA parser library [126] to check whether the specification is syntactically correct. If this is not the case, the parser generates an error report, which is then returned by TOSKERISER. Otherwise, it generates an internal representation of the input specification, which is passed to the step *Validation*.

Validation. The step *Validation* type-checks the (internal representation of the) TOSCA application specification, by verifying the following three conditions:

(v_1) The property constraints expressed in the `node_filter` clause of each node are not conflicting one another (viz., by requiring different versions of the same software distribution, by defining different mappings for the same port, or by defining twice an environment variable),

(v_2) the constraints on operating system and on software distributions are defined by using names of operating systems and software distributions that are actually supported by TOSKERISER[12], and

(v_3) the groups of type *tosker.groups.DeploymentUnit* do not violate the four conditions listed in Sect. 4.4.3.

If at least one out of the conditions v_1, v_2 or v_3 does not hold, then TOSKERISER stops by returning an appropriate error message. Otherwise, the application specification is passed to the step *Filtering*.

Filtering. The step *Filtering* scans the application specification to identify the nodes that have to be hosted on automatically discovered Docker containers. A node needs to be hosted on an automatically discovered Docker container if it satisfies all following conditions:

(f_1) It is of type *tosker.nodes.Software*,

(f_2) it is not hosted on another node of type *tosker.nodes.Software*, and

(f_3) its requirement *host* is not satisfied (viz., it is not connected to any container), or it is part of a group where there exists a node whose requirement *host* is not satisfied.

The result of the step *Filtering* is a set of pairs $\langle nodes, conds \rangle$, where *nodes* is a set of nodes[13], and where *conds* is a multi-set containing the sets of hosting constraints specified by the nodes in *nodes* (viz., each element of *conds* is a set of constraints specified within the requirement *host* of a node in *nodes*). The set of pairs $\langle nodes, conds \rangle$ is then passed to the step *Merge*.

[12]All names of software distributions currently supported by TOSKERISER can be displayed by running the command line instruction `toskerise -supported_sw`.

[13]If a node is not part of a group, then *nodes* will be the singleton set containing only such node. If a node is instead part of a group, then *nodes* will be the set of the nodes that are members of such group.

Merge. For each pair ⟨*nodes*,*conds*⟩, the step *Merge* merges the constraints specified by the sets in *conds* in a single set of *mergedConds* (which will have to be satisfied by the automatically discovered Docker container used to host the corresponding *nodes*).

Given a pair ⟨*nodes*,*conds*⟩, the step *Merge* first checks whether two distinct sets in *conds* impose conflicting constraints (viz., they require different versions of the same software distribution, different operating system distributions, different mappings for the same port, or different values for the same environment variable). If this is not the case, *Merge* proceeds in merging the sets of constraints in *conds* in a single set *mergedConds*. The latter is essentially the set union of all sets in *conds*. The only exception is on version matching of software distributions, as multiple compatible constraints on the version of a same software distribution result in keeping the most stringent constraint (e.g., the constraints `java:1.x`, `java:1.8.x` and `java:1.8.4` result in keeping only the constraint `java:1.8.4`).

The result of the step *Merge* is a set of pairs ⟨*nodes*,*mergedConds*⟩, which is passed to the step *Best-effort adaptation*.

Best-effort adaptation. The purpose of the step *Best-effort adaptation* is to actually enact the completion of the TOSCA application specification, by first trying to determine suitable Docker container for each of the pairs ⟨*nodes*,*mergedConds*⟩ (viz., a Docker container satisfying all hosting requirements *mergedConds* of the nodes in *nodes*), which could then be included within the TOSCA application specification.

This step is "best-effort". Namely, despite it looks for a Docker container satisfying the hosting constraints *mergedConds* for each pair ⟨*nodes*,*mergedConds*⟩, it may happen that such a container is not available. If this is the case, the step *Best-effort adaptation* simply skips the corresponding pair, and it continues adapting the remaining ones. The end-user is however informed by TOSKERISER, which prints out a warning message for each skipped pair.

Such behaviour is obtained by applying to each pair ⟨*nodes*,*mergedConds*⟩ the following three sub-steps (Fig. 4.8):

- The step *Image Search* exploits the hosting constraints in *mergedConds* to build an appropriate query for DOCKERFINDER and to invoke it. If DOCKERFINDER return an empty set of images of Docker containers, then the instance of the sub-process terminates. This would indeed mean that it is not possible to automatically determine a Docker container capable of satisfying the hosting requirements of all the nodes in *nodes*. Otherwise, the set of images is passed to the step *Image Selection*.

- The purpose of step *Image Selection* is to pick one out of the images of Docker containers returned by DOCKERFINDER. The current prototype of TOSKERISER either automatically picks the first image returned by DOCKERFINDER, or it permits manually selecting the image among those returned by DOCKERFINDER (depending on the runtime configuration of TOSKERISER— see Sect. 4.5.2).

- The step *Adaptation* finally includes the selected image of Docker container within the TOSCA application specification by creating a new node of type *tosker.nodes.Container* for modelling such container, and by adding all relationships modelling that the nodes in *nodes* have to be hosted on the newly created node.

Finally, a new CSAR containing the completed TOSCA application specification is returned by *Best-effort adaptation*, and hence by TOSKERISER. An obtained CSAR can then be run "as is" with any orchestration engine providing the needed support for TOSCA and Docker (e.g., TOSKER [30]), provided that all the requirements *host* of the packaged TOSCA application specification have been fulfilled by appropriate containers. This is because there is no need for further adaptation or configuration to be enacted.

4.5.2 How to use TOSKERISER

TOSKERISER is currently implemented as a command-line tool, which can be actually run by executing the following command:

```
$ toskerise FILE [COMPONENTS] [OPTIONS]
```

where `FILE` is the (YAML or CSAR) file containing the TOSCA application specification to be completed. `COMPONENTS` is an optional list, which permits restricting the completion process to a subset of the software components contained in the input application specification (by default, the completion process is applied to all software components). `OPTIONS` is instead a list of additional options, which permit further customising the execution of TOSKERISER. Among all options that can be indicated, the following are the most interesting:

`-constraints` The option `-constraints` permits customising the discovery of Docker images by indicating additional constraints (e.g., by allowing to search for images whose size is lower of `200MB`).

`-policy` This option allows to indicate which images of Docker containers to privilege, among all those that can satisfy the requirement *host* of a software component. The policy `top_rated` (default) privileges images best rated by Docker users, while policies `size` and `most_used` privilege smallest images and most pulled images, respectively.

`-interactive` (or `-i`) This option allows users the manually select the image of the Docker container to be used for satisfying the *host* requirement of a software component, from a list that contains only the best images (according to the privileging policy — see `-policy`).

`-force` (or `-f`) The option `-force` instructs TOSKERISER to search for a new Docker container for each considered component, even if the requirement *host* of such component is already satisfied, viz., even if such requirement is already connected to a container in the application

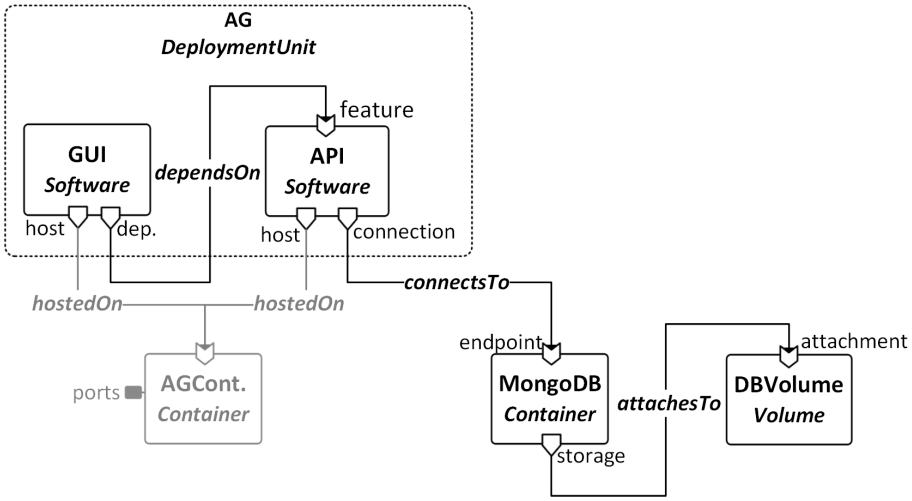


FIGURE 4.9. Application topology obtained by completing the partial topology of the application *Thinking* (Fig. 4.4). Lighter nodes and relationships are those automatically included by TOSKERISER.

specification. In other words, it instructs TOSKERISER to ignore the condition f_3 during the step *Filter* (see Sect. 4.5.1).

Example 4.4. *Consider again the application* Thinking *in our motivating scenario, whose corresponding TOSCA representation is displayed in Fig. 4.6. The CSAR file (*`thinking.csar`*) containing the TOSCA application specification of* Thinking *is publicly available on GitHub*[14]*. Such file can be automatically completed by executing the following command:*

```
$ toskerise thinking.csar --policy size
```

The above will generate a new CSAR file (`thinking.completed.csar`*), which contains the TOSCA specification of* Thinking*, whose topology is completed by including a new Docker container, called* AGContainer*, which is used to host both* API *and* GUI *(Fig. 4.9, lighter node). Such node provides the software support and the port mappings needed by both* API *and* GUI*. We can then run such file with* TOSKER *[30] (or with another orchestration engine supporting both TOSCA and Docker), which will be capable of automatically deploying and managing actual instances of the specified application.*

Please note that we run TOSKERISER *with the option* `-policy size`*. The latter instructs* TOSKERISER *to concretely implement* AGContainer *with the smallest among all images of Docker containers providing the needed software support. Suppose now that we wish to change the container used to host* GUI *and* API*, e.g., because we now wish to select the container that is most*

[14] `https://github.com/di-unipi-socc/TosKeriser/blob/master/data/examples/thinking-app/thinking_group.csar`.

used by Docker users. We can run again TOSKERISER *on the obtained specification, by setting the option* `-f` *to force* TOSKERISER *to replace the Docker container previously included in the specification:*

```
$ toskerise thinking.completed.csar -f --policy most_used
```

This will result in replacing the Docker container implementing AGContainer *by selecting (among all images of Docker containers that can provide the software support needed by* API *and* GUI*) the image that is most used by Docker users.*

4.6 Case studies

We hereby present two case studies based on two different applications[15]. The first case study is used to compare the initial effort required to deploy an application with and without our solution, based on three KPIs (viz., lines of code to be added/changed/deleted, files to be added/changed-/deleted, and programming languages employed — Sect. 4.6.1). The second case study is instead used to compare the effort for maintaining an existing, third-party application with and without our solution, based on the same KPIs (Sect. 4.6.2). We finally present an example illustrating the usefulness of using groups in such case studies (Sect. 4.6.3).

4.6.1 First deployment of a new application

The objective of this first case study is to compare the effort required for performing the first deployment of a newly developed application, with and without our solution. We hence developed from scratch a toy application, called *PingPong* (which we publicly released on GitHub[16]). *PingPong* is composed by 3 interconnected components, viz., *Ping*, *Proxy* and *Pong*. *Ping* is connected to *Proxy*, whose objective is to act as a proxy for all requests sent to *Pong*, and which is hence connected to *Pong*. The behaviour of *PingPong* is as follows: *Ping* sends "ping" messages to *Proxy*, which forwards such messages to *Pong*. The latter replies with "pong" messages, which are sent to *Ping* (by passing through *Proxy*). *Ping* also provides a simple web-based interface allowing to start and stop the ping-pong of messages.

The technical requirements of the components of *PingPong* are as follows. *Ping* is implemented in JavaScript, it must be hosted on a runtime environment supporting npm (version 5) and node (version 8), and it must be connected to *Proxy*. *Proxy* is implemented in Go, it must be installed in a Docker container supporting go (version 1.8) and tar (any version), and it must be connected to *Pong*. *Pong* is implemented in Python, and its runtime environment must support python (version 3), pip (any version) and tar (any version). Additionally, to reduce the network traffic generated

[15]The sources of the case studies and experiments reported in this section are publicly available online at `https://github.com/di-unipi-socc/toskeriser/tree/master/data/examples`.

[16]`https://github.com/di-unipi-socc/ping-pong`.

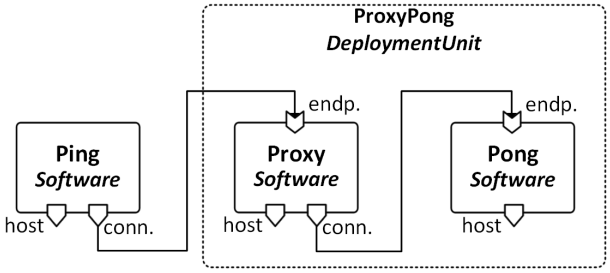


FIGURE 4.10. A specification of the topology of the *PingPong* application in TOSCA (where all relationships are of type *tosca.relationships.ConnectsTo*).

by the components of *PingPong*, *Proxy* and *Pong* must be deployed on the same container, while *Ping* can be hosted in a separate container.

First deployment. To compare the initial effort required to deploy a newly developed application with and without our solution, we performed the deployment of *PingPong* both with our TOSCA-based approach and with the support currently offered by Docker.

Our specification of *PingPong* in TOSCA is illustrated in Fig. 4.10. We modelled all components as nodes of type *tosker.nodes.Software*, and we interconnected them with relationships of type *tosca.relationships.ConnectsTo*. We also indicated all hosting requirements in the requirements *host* of *Ping*, *Proxy* and *Pong*, and we specified the deployment group *ProxyPong*. We also implemented 15 shell scripts for implementing the management operations to install, configure, start, stop and delete each component. We then exploited TOSKERISER to automatically complete the obtained specification of *PingPong*. This resulted in effectively completing the application specification, which we successfully run with TOSKER.

The Docker-based deployment of *PingPong* was instead implemented as follows. We first wrote two Dockerfiles, one for installing *Ping* in a container offering the software support it needs, and one for installing *Proxy* and *Pong* in a container offering the software support they need. We then developed a Docker Compose file orchestrating the deployment of the containers obtained from such Dockerfiles. The obtained Docker Compose file was then successfully run with Docker.

Summary. Table 4.1 compares the effort required to perform the first deployment of the *PingPong* application with and without our solution, in terms of the lines of code and files to be added, changed and deleted, and of the programming languages to be employed. The table highlights that the initial effort required by our solution is slightly higher (in terms of lines of code and number of files) than that currently required by Docker. This is mainly due to the fact that TOSCA requires to initially specify more information with respect to Docker. Most of the bash commands contained in the shell scripts written for the TOSCA-based deployment are indeed also contained in the Dockerfiles written for the Docker-based deployment.

The higher amount of information to be initially provided may be perceived as a drawback of our approach (and of TOSCA, as well), as it increases the initial effort for deploying multi-

KPI	TOSKERISER	Docker-based
Lines of code	141 *a:141,c:0,d:0*	89 *a:89,c:0,d:0*
Files	16 *a:16,c:0,d:0*	3 *a:3,c:0,d:0*
Languages	2 *TOSCA,bash*	3 *Dockerfile,Docker Compose,bash*

TABLE 4.1. Initial effort required to deploy the *PingPong* application with TOSKERISER and with Docker. The abbreviations *a*, *c* and *d* denote *added*, *changed* and *deleted*, respectively.

Node	Needed software distributions
Frontend	npm (version 2.15), node (version 4), git (any version)
Catalogue	go (version 1.7), git (any version)
Users	go (version 1.7), git (any version)
Carts	java (version 1.8)
Orders	mvn (version 3), java (version 1.8), git (any version)
Payment	go (version 1.7), git (any version)
Shipping	java (version 1.8)

TABLE 4.2. Technical requirements of the main services in *Sock Shop*.

component applications. However, it actually pays off while maintaining an application (e.g., when the requirements of components change, or when we wish to re-group the components of an application), as we will show in the next section.

4.6.2 Maintenance of a third-party, existing application

Sock Shop [160] is an open-source, service-based application. *Sock Shop* is publicly available on GitHub[17], and it is maintained by Weaveworks (`https://www.weave.works`) and Container Solutions (`https://container-solutions.com`) The application simulates the user-facing part of an e-commerce website selling socks, and it is composed by 14 interconnected components.

The main components of *Sock Shop* are a *Frontend* displaying a graphical user interfaces for e-shopping socks, a set of pairs of services and databases for storing and managing the catalogue of available socks (viz., *Catalogue* and *CatalogueDB*), the users of the application (viz., *Users* and *UsersDB*), the users' shopping carts (viz., *Carts* and *CartsDB*), and the users' orders (viz., *Orders* and *OrdersDB*), and two services for simulating the payment and shipping of orders (viz., *Payment* and *Shipping*). The technical requirements of the above mentioned services are recapped in Table 4.2.

The *Sock Shop* application is then completed by three other components, namely *Edge Router*,

[17] `https://github.com/microservices-demo/microservices-demo`.

KPI	TOSKERISER	Docker-based
Lines of code	303	268
Files	26	8
Languages	2	3
	TOSCA,bash	*Dockerfile,Docker Compose,bash*

TABLE 4.3. Values of the considered KPIs for the the initial deployment of *Sock Shop* with TOSKERISER, and for its already existing Docker-based deployment.

RabbitMQ and *Queue Master*. The *Edge Router* redirects user requests to the *Frontend*. The *RabbitMQ* is a message queue that is filled of shipping requests by the *Shipping* service. The shipping requests are then consumed by the *Queue Master*, to simulate the actual shipping of orders.

As *Sock Shop* is intended to aid the demonstration and testing of solutions for orchestrating multi-component applications, we exploited it to compare the effort for maintaining an existing, third-party application with and without our solution. More precisely, we exploited it to measure the effort needed for addressing three subsequent changes in the deployment of *Sock Shop*:

(i) *Frontend* requires a new version of npm,

(ii) *Frontend* and *Catalogue* must be installed in the same container, and

(iii) *Orders*, *Users* and *Carts* must be installed in the same container.

While the Docker-based deployment of *Sock Shop* was already available in its GitHub repository[18], we had to develop from scratch its specification with our TOSCA-based representation. Our specification of *Sock Shop* in TOSCA is illustrated in Fig. 4.11 and it is publicly available on GitHub[19]. We modelled all databases and infrastructure components as nodes of type *tosker.nodes.Container*, and we exploited the Docker containers already configured by Weaveworks to actually implement them. We instead specified the services *Frontend*, *Catalogue*, *Users*, *Carts*, *Orders*, *Payment* and *Shipping* as nodes of type *tosker.nodes.Software*, each having a pending requirement *host* that specifies the hosting constraints of the node (Table 4.2). We also developed 25 shell scripts for implementing the management operations offered by *Frontend*, *Catalogue*, *Users*, *Carts*, *Orders*, *Payment* and *Shipping*[20]. We then completed the specification of *Sock Shop* with TOSKERISER, and we successfully run the completed specification with TOSKER.

Table 4.3 recaps the initial values of the KPIs we consider for the *Sock Shop* application. The specification of *Sock Shop* with our approach required us to manually write 303 lines of

[18] `https://github.com/microservices-demo/microservices-demo/tree/master/deploy/docker-compose`.

[19] `https://github.com/di-unipi-socc/TosKeriser/tree/master/data/examples/sockshop-app`.

[20] The management operations of a component have to be implemented by an associated artifact only when the component actually needs such operations [124]. For instance, as *Users* does not require to be configured, we do not need to develop a script for implementing its management operation *configure*.

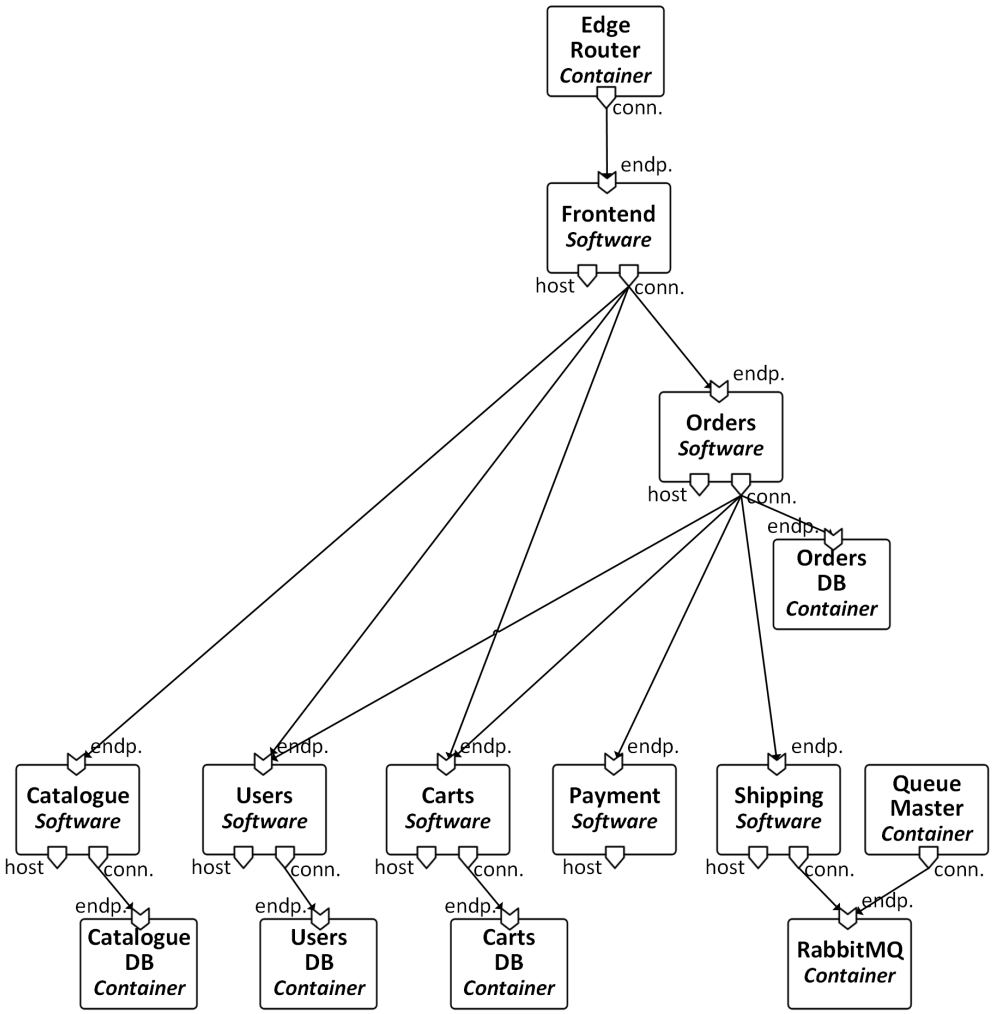


FIGURE 4.11. A specification of the topology of *Sock Shop* in TOSCA (where all relationships are of type *tosca.relationships.ConnectsTo*).

code in 26 different files, by exploiting 2 different languages (TOSCA and bash). The already available Docker-based deployment of *Sock Shop* instead counts 268 lines of code in 8 different files, by exploiting 3 different languages (Dockerfile, Docker Compose and bash). This confirms that the initial effort with our approach is higher. At the same time, it is important to observe that the differences between our approach and that based on Docker here are relatively lower (with respect to the case of *PingPong*). This is because the impact of the additional information to be provided with our TOSCA-based representation is lowered by the higher amount of bash commands needed to install the services forming *Sock Shop*, which are contained both in the shell script implementing the management operations in our solutions and in the Dockerfiles required by the Docker-based deployment.

Case (i). We first considered the case of a component requiring to upgrade the software support

KPI	TOSKERISER	Docker-based
Lines of code	1 *a:0,c:1,d:0*	1 *a:0,c:1,d:0*
Files	1 *a:0,c:1,d:0*	1 *a:0,c:1,d:0*
Languages	1 *TOSCA*	2 *Dockerfile,bash*

TABLE 4.4. Effort required to update the deployment of the *Sock Shop* application, in order to provide its *Frontend* with the new version of npm it requires. The abbreviations *a*, *c* and *d* denote *added*, *changed* and *deleted*, respectively.

provided by the container hosting it, which is a frequent issue while maintaining in multi-component applications [123]. We considered the case of *Frontend* requiring to upgrade the version of npm supported by its hosting container from 2.15 to 3.10. We then compared the effort required to update the deployment based on our approach and that based on the support currently provided by Docker.

Our approach allowed us to update the TOSCA-based representation of *Sock Shop* by simply replacing the constraint on npm in the requirement *host* of *Frontend*, viz., `npm: 2.15.x` was replaced by `npm: 3.10.x`. The update in the Docker-based deployment instead required us to manually change the Dockerfile installing *Frontend* in its container. More precisely, it required us to add the a new line instructing to upgrade the npm version supported by the container, viz.,

```
RUN npm i npm@3.10 -g
```

at the beginning of the Dockerfile of *Frontend*. The corresponding efforts (in terms of the three KPIs we consider) is reported in Table 4.4.

Both updates led to runnable instances of *Sock Shop*, with the desired, updated support for its *Frontend*. Although they were also similar in terms of the considered KPIs, by looking at the concrete changes that we performed, we can already appreciate some concrete differences. To update the specification of *Sock Shop*, our approach required us to change the actual value assigned to a pre-existing constraint (and the Docker container providing the desired version of npm was then automatically determined). To update the Docker-based specification of *Sock Shop*, we instead had to manually look for the bash command allowing to upgrade the distribution of npm, and to insert such command in the Dockerfile of *Frontend* in such a way that the desired version of npm is available when needed. In the latter case, we were also required to manually check that no conflicts were generated by the newly inserted command.

Case (ii). We then considered the case of being required to deploy two different components in the same container, e.g., to reduce the network traffic generated by the components of *Sock Shop*.

KPI	TOSKERISER	Docker-based
Lines of code	4 *a:4,c:0,d:0*	176 *a:114,c:4,d:58*
Files	1 *a:0,c:1,d:0*	4 *a:1,c:1,d:2*
Languages	1 *TOSCA*	3 *Dockerfile,Docker Compose,bash*

TABLE 4.5. Effort required to update the deployment of the *Sock Shop* application, in order to deploy *Frontend* and *Catalogue* within the same container. The abbreviations *a*, *c* and *d* denote *added*, *changed* and *deleted*, respectively.

We focused on grouping *Frontend* and *Catalogue*, as the former often interacts with the latter to display the socks available in the e-shop.

We added the group to our TOSCA-based representation of *Sock Shop* by defining a group of type *tosker.groups.DeploymentUnit*. More precisely, we added the following lines at the end of the specification of *Sock Shop*:

```
groups:
    my_group1:
        type: tosker.groups.DeploymentUnit
        members: [ front-end, catalogue ]
```

We then run TOSKERISER (with the option `-f` set), and we obtained an updated specification hosting *Frontend* and *Catalogue* on the same container (providing all the software support they need).

We instead updated the Docker-based deployment of *Sock Shop* by deleting the Dockerfiles installing *Frontend* and *Catalogue*, and by creating a new Dockerfile installing both components in an appropriate container. We then updated the Docker Compose file specifying the orchestration of the containers of *Sock Shop*, which had to refer the newly created Dockerfile instead of the deleted ones.

Despite both updates led to runnable instances of *Sock Shop* (with *Frontend* and *Catalogue* grouped together), the effort required by our approach was by far lower with respect to that required by the Docker-based deployment (Table 4.5). This is even more evident if we compare the lines and files changed with those of the initial specification (Table 4.3). With our approach, we reuse 100% of the lines and files we already wrote, as we only add 4 lines to 1 file. The update to the Docker-based deployment instead has a much higher impact and it experiences a much lower reuse, as the initial deployment counts 268 lines of code distributed over 8 files, and since we had to edit 176 lines of code over 4 files. Additionally, while with our approach we were required to only work with the TOSCA language, the update to the Docker-based deployment required us to work with three different languages.

KPI	TOSKERISER	Docker-based
Lines of code	9	164
	a:3,c:6,d:0	*a:100,c:0,d:64*
Files	1	4
	a:0,c:1,d:0	*a:1,c:2,d:3*
Languages	1	3
	TOSCA	*Dockerfile,Docker Compose,bash*

TABLE 4.6. Effort required to update the deployment of the *Sock Shop* application, in order to deploy *Orders*, *Users* and *Carts* within the same container. The abbreviations *a*, *c* and *d* denote *added*, *changed* and *deleted*, respectively.

Case (iii). We finally considered the grouping of *Orders*, *Users* and *Carts*, which we wished to install within the same container, and we compared the effort required to perform the corresponding update with our approach and with the support currently provided by Docker.

We updated our TOSCA-based representation of *Sock Shop* by adding a group of type *tosker.groups.DeploymentUnit*, viz., by adding the following lines at the end of the specification of *Sock Shop*:

```
my_group2:
    type: tosker.groups.DeploymentUnit
    members: [ orders, user, carts ]
```

By running TOSKERISER (with the option `-f` set), we discovered that there were conflicting requirements on the port mappings required by the grouped components. We hence had to change 6 other lines of our specification for reconfiguring the port mappings in order to avoid the discovered conflicts. We then re-run TOSKERISER (with the option `-f` set) and we obtained an updated specification hosting the three components on the same container (providing all the software support they need).

The update to the Docker-based deployment instead required us much more effort. We had to delete the Dockerfiles installing *Orders*, *Users* and *Carts*, and to create a new Dockerfile installing the three components in a container providing the needed support. In doing so, we had to manually manage the issues due to conflicting port mappings (already known thanks to the above mentioned run of TOSKERISER), by configuring *Orders*, *Users* and *Carts* to listen on different ports of their container. This required us also to update the Dockerfile packaging *Frontend* and *Catalogue* in a container, to allow such components to connect to the newly configured *Orders*, *Users* and *Carts*. We finally had to update the Docker Compose file specifying the orchestration of the containers of *Sock Shop*, which had to refer the newly created Dockerfile instead of the deleted ones.

Table 4.6 illustrates the measured effort for performing both above mentioned updates, in terms of lines of code and files to be added, changed and deleted, and of languages to be employed.

The table highlights how case (iii) is another example showing that the maintenance effort with our approach is much lower than that without our approach. What we can observe is indeed very similar to the case of grouping *Frontend* and *Catalogue*. Additionally, while TOSKERISER automatically discovers conflicting requirements, the same does not hold for the support currently provided by Docker.

Summary. Finally, consider the effort required by three changes together (Table 4.7). Our approach required us to overall edit 14 lines of code, by only touching the file containing the TOSCA application specification. The impact on the initial specification was hence minimum, as the latter consisted in writing 303 lines of code distributed over 26 different files. The effort required by the Docker-based deployment was instead highly impacting on the initial specification. Indeed, while the initial specification consisted of 268 lines of code distributed over 8 files, we were required to edit 341 lines of code over 9 files.

KPI	TOSKERISER	Docker-based
Lines of code	14 *a:7,c:7,d:0*	341 *a:215,c:4,d:112*
Files	1 *a:0,c:1,d:0*	9 *a:2,c:2,d:5*
Languages	1 *TOSCA*	3 *Dockerfile,Docker Compose,bash*

TABLE 4.7. Overall effort for updating the deployment of *Sock Shop*, in order to address cases (i), (ii) and (iii). The abbreviations a, c and d denote *added*, *changed* and *deleted*, respectively.

We can hence observe that despite our approach required us a slightly higher effort for developing the initial specification of *Sock Shop*, such effort actually paid off by the maintainability of the obtained specification.

4.6.3 On groups

How to group the components forming an application depends on the governance of the application itself. For instance, as the performances (in terms of delay and throughput) of Docker networking are low on average [167], we may wish to reduce the network traffic generated by the components of an application. Grouping can help reducing the the network traffic generated by the components of an application, as shown by the following experiments on the applications *PingPong* and *Sock Shop*.

***PingPong*.** We measured the network traffic generated by the containers running components of *PingPong* for processing 100 "ping" requests. More precisely, we run *PingPong* with two different configurations, one with each component in a different container, and one grouping *Proxy* and *Pong* in a single container (with *Ping* in a separate container). For both deployments, we iterated

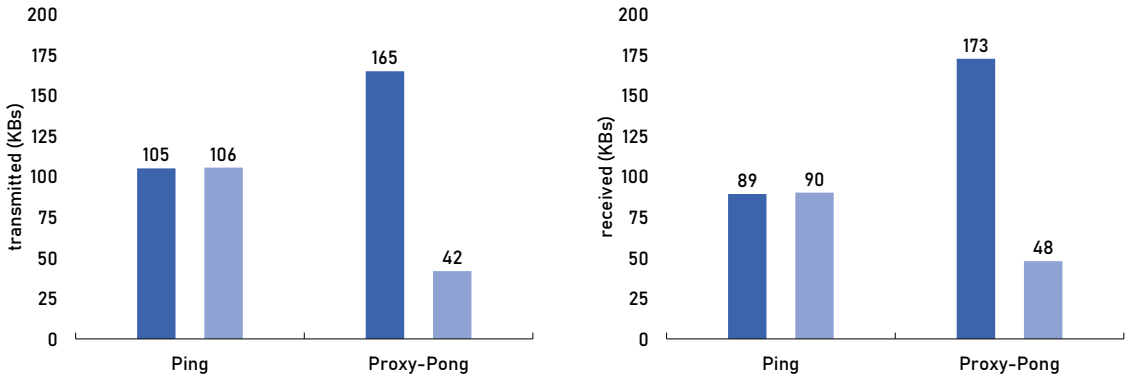


FIGURE 4.12. Average network traffic (a) transmitted and (b) received by the components of *PingPong* for processing 100 "ping" requests. Darker histograms plot the values for a deployment of *PingPong* with each component in a separate container, while lighter histograms plot the values for a deployment of *PingPong* with the components *Proxy* and *Pong* hosted in the same container.

50 times a test executing 100 "ping" requests, and we measured the network traffic generated by the containers running the components of *PingPong* for each iteration.

Fig. 4.12 shows the average network traffic transmitted by the containers running the components of *PingPong* for executing the above illustrated test. While the network traffic of *Ping* keeps stable in both deployments, by grouping *Proxy* and *Pong* in a single container, we effectively reduced the average network traffic generated by the containers running the components of *PingPong*. The average network traffic generated by the deployment with one component per container was indeed 532.84 KBs (270.45 KBs transmitted, 262.39 KBs received), while that of the deployment with *Proxy* and *Pong* in the same container was 285.90 KBs (147.60 KBs transmitted, 138.30 KBs received). This means that by hosting *Proxy* and *Ping* on the same container we reduced the average network traffic of around 46%.

Sock Shop. We also prepared a test for comparing the network traffic generated by two different deployments of *SockShop*, viz., its default deployment (with each component in a separate container), and the deployment obtained at the end of the case study discussed in Sect. 4.6.2 (with two groups placing *Frontend* and *Catalogue* in one container, and *Carts*, *Users* and *Orders* in another single container). Each test consisted in executing an end-to-end test of *Sock Shop*[21], which simulates an end-user interacting with the web-based interface of *Sock Shop* to perform an order of a given pair of socks. We repeated such test 50 times for both deployments, to measure the average network traffic transmitted by the containers running the components of *Sock Shop*.

Table 4.8 shows the average network traffic transmitted and received by the containers

[21] `https://github.com/di-unipi-socc/e2e-tests`.

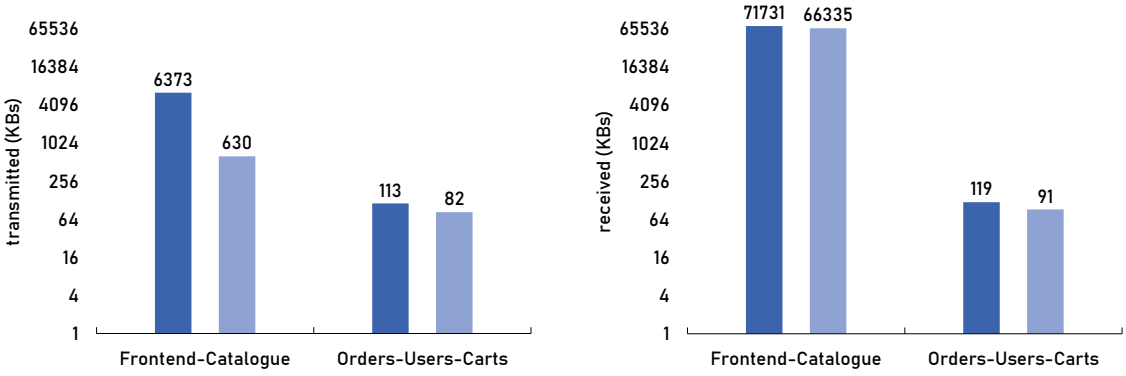
	Transmitted (KBs)	Received (KBs)
Carts	31.01	24.93
CartsDB	17.97	15.61
Catalogue	106.50	5672.96
CatalogueDB	18.87	48.95
Frontend	6266.88	66058.24
Orders	48.77	65.70
OrdersDB	6.78	37.85
Payment	6.34	0.51
Shipping	6.30	0.92
Users	33.53	28.71
UsersDB	18.11	18.69

(a)

	Transmitted (KBs)	Received (KBs)
CartsDB	18.25	15.00
CatalogueDB	18.53	49.20
Frontend-Catalogue	629.56	66334.72
Orders-Users-Carts	82.48	91.45
OrdersDB	6.70	23.63
Payment	6.33	0.51
Shipping	6.16	0.93
UsersDB	17.64	18.54

(b)

TABLE 4.8. Average network traffic of the containers running the main components of *Sock Shop* for 10 iterations of end-to-end test on (a) the default deployment of *Sock Shop* and (b) a deployment grouping *Frontend* and *Catalogue*, and *Orders*, *Users* and *Carts*.



(a) (b)

FIGURE 4.13. Average network traffic (a) transmitted and (b) received by the containers running the grouped components of *Sock Shop* executing its end-to-end test. Darker histograms plot the values for a deployment of *Sock Shop* with each component in a separate container, while lighter histograms plot the values for a deployment of *Sock Shop* with the groups *Frontend-Catalogue* and *Orders-Users-Carts*. Values are displayed by exploiting a logarithmic scale on the y-axis.

running the main components of *Sock Shop*, in the two different deployments discussed above. By introducing the groups *Frontend-Catalogue* and *Orders-Users-Carts*, we effectively managed to reduce the average network traffic generated by the internals of *Sock Shop* of around 14%. This is mainly thanks to the reduction of the traffic generated by the containers running the grouped components (which can be observed in Fig. 4.13).

Summary. The above reported experiments showed that the network traffic generated by the

containers running the components of both *PingPong* and *Sock Shop* was effectively reduced by grouping multiple components in a single container.

4.7 Related work

We presented a solution for automatically completing TOSCA specifications, which is much in the spirit of [89]. The goal of [89] is indeed to reduce the effort paid by TOSCA delevopers, by allowing them create incomplete application topologies, which then have to be automatically completed. Developers can focus on modelling the components that are specific to their applications, by also indicating the types of nodes needed to host them (e.g., a web server or a DBMS). The solution proposed by [89] automatically adds nodes and relationships to an incomplete TOSCA specification, in order to build the software stack needed to run each of its component. Such nodes and relationships are taken from a finite alphabet of supported node/relationship types, and a manual refinement step is foreseen for developers to specify the configuration of the nodes automatically included in their topologies. However, the approach presented in [89] only checks type-compatibility between specified nodes and those automatically included to form their runtime environments. We instead allow developers to impose additional constraints on the nodes that can be used to host a component (e.g., by allowing to indicate that an application component requires a certain software support on a certain operating system distribution). Additionally, our solution does not require further adaptation/configuration of the Docker containers automatically included in an application.

Other approaches worth mentioning are [32], [33] and [146], whose goal is however different from ours. They indeed focus on allowing to reuse portions of existing TOSCA applications while developing new applications. This means that [32], [33] and [146] can still be used to automatically determine the runtime environment needed by the components of TOSCA applications. They indeed allow to abstractly specify desired nodes, and they can determine actual implementations for such nodes by matching and adapting existing TOSCA application specifications. [32], [33] and [146] however differ from our approach as they look for type-compatible solutions, without constraining the actual values that can be assigned to a property (hence not allowing to indicate the software support that must be provided by a Docker container, for instance).

If we broaden our view beyond TOSCA, we can identify various other efforts that have been recently oriented to try devising systematic approaches to adapt multi-component applications to work with heterogeneous cloud platforms. For instance, [57] and [81] propose two approaches to transform platform-agnostic source code of applications into platform-specific applications. In contrast, our approach does not require the availability of the source code of an application, and it is hence applicable also to third-party components whose source code is not available nor open.

[80] proposes a framework allowing developers to write the source code of cloud applications as if they were “on-premise" applications. [80] is similar to our approach, since, based on cloud

deployment information (specified in a separate file), it automatically generates all artefacts needed to deploy and manage an application on a cloud platform. [80] however differs from our approach, as artefacts must be (re-)generated whenever an application is moved to a different platform, and since the obtained artefacts must be manually orchestrated on such platform. Our approach instead produces portable TOSCA application specifications, which can be automatically orchestrated by engines supporting both TOSCA and Docker (e.g., TOSKER [30]).

In general, most existing approaches to the reuse of cloud applications support a from-scratch development of cloud-agnostic applications, and do not account for the possibility of adapting existing (third-party) components. To the best of our knowledge, ours is the first approach for adapting existing multi-component applications to work with heterogeneous cloud platforms, by relying on a natural combination of two standards (viz., TOSCA [124] and Docker) to achieve cloud interoperability. TOSCA is indeed exploited to specify the orchestration of multi-component applications in a cloud-agnostic manner, for which it has proven abilities [17, 125]. Docker is instead exploited to standardise the virtual runtime environments of the components forming an application to Linux-based containers, which are portable and widely supported by cloud platforms (as Docker is the de-facto standard for container-based virtualisation [127]).

4.8 Conclusions

Cloud applications typically consist of multiple heterogeneous components, whose deployment, configuration, enactment and termination must be suitably orchestrated [66]. This is currently done manually, by requiring developers to manually select and configure an appropriate runtime environment for each component in an application, and to explicitly describe how to orchestrate such components on top of the selected environments.

In this chapter, we have presented a solution for enhancing the current support for orchestrating the management of cloud applications, based on TOSCA and Docker. More precisely, we have proposed a TOSCA-based representation for multi-component applications, which allows developers to describe *only* the components forming an application, the dependencies among such components, and the software support needed by each component. We have also presented a tool (called TOSKERISER), which can automatically complete the TOSCA specification of a multi-component application, by discovering and configuring the Docker containers needed to host its components.

The obtained application specifications can then be processed by orchestration engines supporting TOSCA and Docker, like TOSKER [30], which can process specifications produced by TOSKERISER, to automatically orchestrate the deployment and management of the corresponding applications.

TOSKERISER is integrated with DOCKERFINDER [27], and it produces specifications that can be effectively processed by TOSKER [30]. TOSKERISER, DOCKERFINDER and TOSKER are all open-

source prototypes, and their ensemble provides a first support for automating the orchestration of multi-component applications with TOSCA and Docker. Future work on this ensemble regards its engineering. In this perspective, we plan to evaluate and improve the performances of each tool (TOSKERISER, DOCKERFINDER and TOSKER) and of their ensemble as well.

We also plan to further extend the open-source ensemble composed by TOSKERISER, DOCKERFINDER and TOSKER, to pave the way towards the development of a full-fledged, open-source support for orchestrating multi-component applications with TOSCA and Docker. In this direction, it is worth highlighting that despite DOCKERFINDER can provide information on all Docker images available on Docker Hub, it may be the case that no existing image is providing the combination of software support and operating system distribution needed by a group of application components. This would hence impede TOSKERISER to complete the TOSCA application specification containing such group of components. A tool supporting the creation of ad-hoc images (configured from scratch, if needed) would permit overcoming this limitation. The development of such tool and its integration with TOSKERISER are in the scope of our future work.

Another interesting direction for future work is to investigate whether existing approaches for reusing fragments of TOSCA applications (e.g., TOSCAMART [146]) can be included in TOSKERISER. This would permit completing TOSCA specifications by hosting the components of an application not only on single Docker containers, but also on software stacks already employed in other existing solutions.

Another interesting direction is to integrate our open source environment TOSKERISER and TOSKER with existing approaches allowing to determine the optimal deployment of multi-component applications on virtual infrastructures (such as Zephyrus [6], for instance). The output of TOSKERISER could indeed be provided as input to a tool like Zephyrus, along with a description of the virtual machines where the application components can run and a set of deployment constraints (e.g., desired number of replicas of each component, co-installation requirements, conflicting components, etc.). Zephyrus could automatically determine an optimal application deployment of the application components on the available infrastructure.

CHAPTER 5

A MICROSERVICE-BASED ARCHITECTURE FOR (CUSTOMISABLE) ANALYSES OF DOCKER IMAGES

In this chapter , we introduce DOCKERANALYSER, a microservice-based tool that permits building customised analysers of Docker images. The architecture of DOCKERANALYSER is designed to crawl Docker images from a remote Docker registry, to analyse each image by running an analysis function, and to store the results into a local database. Users can build their own image analysers by instantiating DOCKERANALYSER with a custom analysis function and by configuring the architecture. More precisely, the steps needed to obtain new analysers are: (i) replacing the analysis function used to analyse crawled Docker images, (ii) setting the policy for crawling Docker images, and (iii) setting the scalability options for obtaining a scalable architecture. We also present 2 different use cases, i.e., 2 different analysers of Docker images created by instantiating DOCKERANALYSER with 2 different analysis functions and configuration options. The 2 use cases show that DOCKERANALYSER decreases the effort required to obtain new analysers versus building them from scratch.

The results of this Chapter were published in [28], which appeared in the journal "Software: Practice and Experience".

5.1 Introduction

Container-based virtualisation [102, 110] has gained significant acceptance, because it provides a lightweight solution for running multiple isolated user-space instances (called *containers*). Such instances are particularly suited to package, deploy and manage complex, multi-component applications [14]. Developers can bundle application components along with the dependencies they need to run in isolated containers and execute them on top of a container run time

(e.g., Docker [58], Rkt [50], Dynos [87]). Compared to previous existing virtualisation approaches, like virtual machines, the use of containers features faster start-up times and less overhead [98].

The current de-facto standard technology for container-based virtualization is Docker [52, 127], a platform for building, shipping, and running applications inside portable containers. Docker containers run from Docker images, which are the read-only templates used to create them. A Docker image permits packaging a software component together with all the software dependencies needed to run it (e.g., libraries, binaries). In addition, Docker provides the ability to distribute and search (images of) containers that were created by other developers through Docker registries. Given that any developer can create and distribute its own created images through Docker registries, other users have at their disposal plentiful repositories of heterogeneous, ready-to-use images. In this scenario, public registries (such as the official Docker Hub [59]) are playing a central role in the distribution of images [60].

However, images stored in Docker registries are described by fixed attributes (e.g., name, description, owner of the image), and this makes it difficult for users to analyse and select the images satisfying their needs. Also, needs may differ from user to user, depending on the actual exploitation of Docker images they wish to carry out. For instance, a developer may want to deploy her application on a Docker image supporting precise software distributions (e.g., *Python 2.7* and *Java 1.8*), an end-user may want to assign custom tags to her images in order to ease their retrieval, or a data scientist may wish to analyse images to discover interesting, recurring patterns.

Currently, a support for performing analyses on large set of Docker images is missing. Users are required to manually check whether an image satisfies their needs by looking at the attributes provided by the Docker registry or on the image features by running it in a container.

In this chapter, we present DOCKERANALYSER, a tool that permits building customised analysers of Docker images. Users can create their own Docker image analysers by simply instantiating DOCKERANALYSER with a user-defined analysis function that produces descriptions of Docker images. The analysis function can be any Python code that, given the name of a Docker image, scans such image to extract some metadata that are used to generate the description of the image. DOCKERANALYSER is designed to provide a scalable architecture for running the analysis function provided by the users on large set of Docker images in a fully automated way. Users are only required to provide the analysis function, while DOCKERANALYSER provides the other functionalities for crawling Docker images from a Docker registry, running the analysis function on each image, storing the results of the analysis function in a local storage, and allowing to query the storage through a RESTful API.

To illustrate this, we implemented two different analysers of Docker images, namely DOCKERFINDER and DOCKERGRAPH.

- DOCKERFINDER collects the software distributions supported by an image, and it permits searching for images supporting such software distributions. For instance, if a developer

wishes to package her application into a Docker image satisfying certain software requirements (e.g. *Python 2.7* and *Java 1.8*), she can query DOCKERFINDER and select the image that best satisfies such requirements.

- DOCKERGRAPH creates a directed graph whose nodes are names of (repositories of) Docker images, and whose arcs connect each image i to its *parent* image (viz., the image that has been used as the basis to create i), if any. Many applications can take advantage of the graph created by DOCKERGRAPH. For instance, if a parent image is affected by a security flaw, DOCKERGRAPH can be used for retrieving all the images that are built starting from such image.

We deployed both DOCKERGRAPH and DOCKERFINDER as multi-container Docker applications where the microservices of the analysers run inside Docker containers.

We chose to implement DOCKERANALYSER as a suite of interacting microservices mainly because of the configurability properties of microservice-based architectures [108, 121]. For instance, replaceability in a microservice-based architecture allows replacing a microservice with another offering the same interface, without affecting any of the other microservices composing the architecture [121]. In DOCKERANALYSER, replaceability allows obtaining different analysers by just changing the actual implementation of the microservice running the analysis function. We illustrate this by showing how DOCKERFINDER and DOCKERGRAPH are built by changing the implementation of such microservice.

The results presented in this chapter extend those presented in [27]. [27] describes a tool that analyses Docker images by executing a fixed analysis function. DOCKERANALYSER is a generalisation of that in [27], and such generalisation permits customising the analysis function executed by the architecture in order to create different analysers of Docker images.

The rest of the paper is organised as follows. Sect. 5.2 describes the microservice-based architecture of DOCKERANALYSER. Sect. 5.3 introduces the DOCKERANALYSER tool. Sect. 5.4 presents 2 use cases of analysers of Docker images (DOCKERFINDER and DOCKERGRAPH) obtained by customising the analysis function of DOCKERANALYSER. Sect. 5.5 discusses related work. Sect. 5.6 draws some conclusions.

5.2 DOCKERANALYSER architecture

The objective of DOCKERANALYSER (Fig. 5.1) is to permit building analysers of Docker images. A new analyser of Docker images can be created by instantiating DOCKERANALYSER with a different analysis function (contained in the *deploy package*). We implemented DOCKERANALYSER as a suite of interacting microservices.

- *A*nalysis. DOCKERANALYSER crawls and analyses each image contained in the Docker registry it is connected to. The analysis of the images is performed by running the analysis function provided by the user.

- *S*torage. DOCKERANALYSER stores all produced image descriptions into a local storage. The storage is then made accessible to external users through a RESTful API.

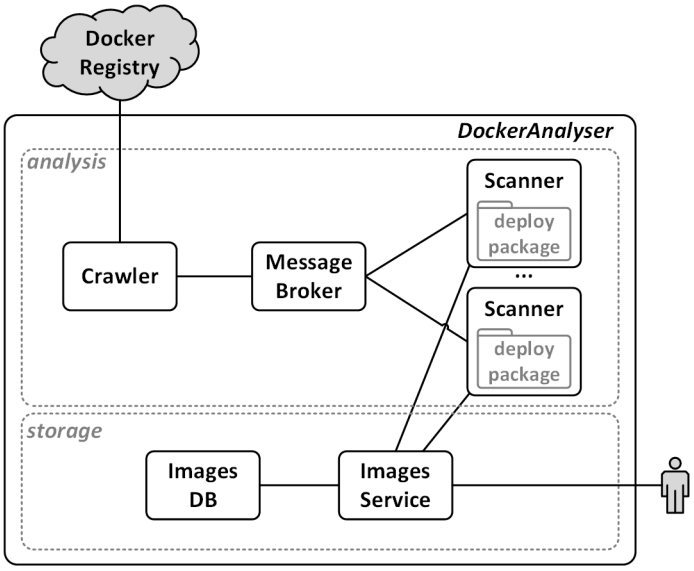


FIGURE 5.1. Microservice-based architecture of DOCKERANALYSER.

We now detail the microservices composing the architecture of DOCKERANALYSER (Fig. 5.1). We separately discuss the microservices in the *analysis* group (Sect. 5.2.1) and those in the *storage* (Sect. 5.2.2) group.

5.2.1 Microservices in the Analysis group

As illustrated in Fig. 5.1, the *analysis* is carried out by a *Crawler*, a *Message Broker*, and (one or more) *Scanner*s.

Crawler. The *Crawler* crawls the Docker images to be analysed from a remote Docker registry. More precisely, the *Crawler* crawls the *names* of the images from the registry, and it passes such names to the *Message Broker*. The *Crawler* can be configured by the users to implement two different crawling policies: randomly or sequentially. The former permits crawling a random sample of images, while the latter permits crawling all the images sequentially. In both cases, the total number of images to be crawled can be configured.

Message Broker. A message broker is an intermediary service whose purpose is to take incoming messages from one or multiple sources, to process such messages, and to route them to one or

more destinations [43]. The *Message Broker* of DOCKERANALYSER receives the names of the images to be analysed (from the *Crawler*), it stores them into a messages queue, and it permits the *Scanner*s to retrieve them. The goal of the *Message Broker* is to decouple the *Crawler* from the *Scanner*s.

Scanner. The *Scanner* retrieves the name of the images from the *Message Broker*, and for each name received it runs the analysis function. More precisely, given a user-defined function *analysis*, each *Scanner* continuously works as follows:

1. It retrieves an image name *i* from the *Message Broker*.

2. It runs the analysis function *analysis* on the image name *i* producing a description *descr*=analysis*(i)*.

3. It sends the generated description *descr* to the *Images Service* that stores the description into the local storage.

The description *descr* sent to the *Images Service* is a JSON object containing the information obtained by running the analysis function on the image. It is worth noting that the *Scanner*, depending on the analysis function executed, can be the most time consuming service in the architecture. For example, if the function *analysis* requires downloading all layers of a Docker image locally then it can require up to minutes to download a single image. In order to decrease the time needed to analyse images, the number of *Scanner* microservices can be increased by exploiting the scalability property of microservice-based architectures (see Sect. 5.4.1 for a concrete example of analyser exploiting such scalability to reduce the time to analyse images).

5.2.2 Microservices in the *storage* group

DOCKERANALYSER stores all image descriptions produced by the *Scanner*s into a local storage. The images descriptions stored in the local storage are made accessible through a RESTful API. To accomplish such a *storage* functionality, DOCKERANALYSER relies on a microservice composed by the *Image Service* and *Image Database* (Fig. 5.1).

Images Database. The *Images Database* is the local repository where the image descriptions are stored. Given that different analysis functions can produce different image descriptions, the *Images Database* has been implemented as a NoSQL database without a fixed model.

Images Service. The *Images Service* is a RESTful service that permits adding, deleting, updating, and searching image descriptions inside the *Images Database*. The *Images Service* interface is used both by other microservices in DOCKERANALYSER (for adding, deleting, updating images descriptions) and by external users (for submitting queries to the local repository).

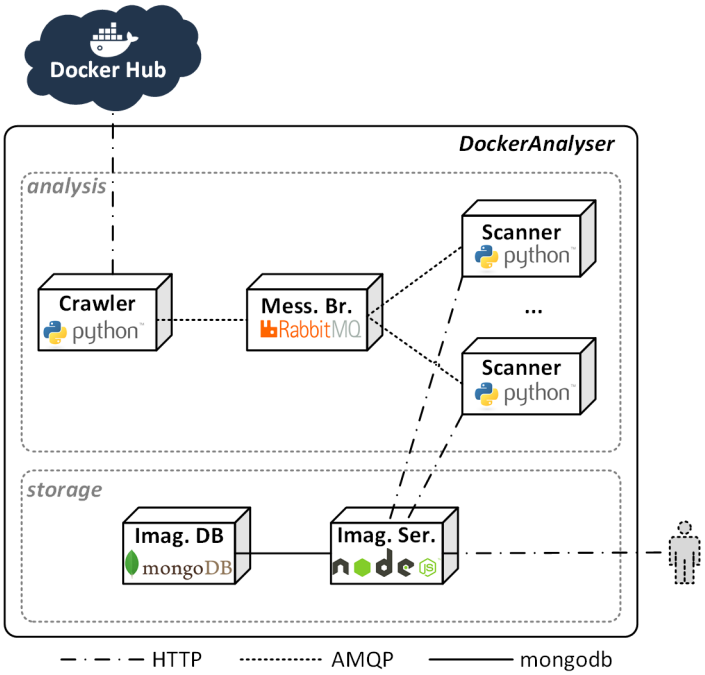


FIGURE 5.2. DOCKERANALYSER as a multi-container Docker application.

5.3 DOCKERANALYSER

We hereby illustrate the implementation of DOCKERANALYSER[1] (Sect. 5.3.1) and we then show the steps needed to obtain different analysers of Docker images (Sects. 5.3.2 and 5.3.3).

5.3.1 Implementation of DOCKERANALYSER

The microservice-based architecture of DOCKERANALYSER has been implemented as a multi-container Docker application, where each microservice is implemented and shipped in its own Docker container. Fig. 5.2 illustrates such a multi-container Docker application by representing each Docker container as a box labelled with the name of the microservice it implements, and with the logo of the official Docker image used to ship such microservice. Fig. 5.2 shows also the communication protocol exploited by the microservices to interact each other (viz., HTTP, AMQP), and the Docker registry from which to retrieve the images to be analysed (viz., the Docker Hub). We now separately discuss the implementation of the microservices in the *analysis* and *storage* groups.

Analysis. The *Message Broker* is implemented by directly exploiting the official Docker image for RabbitMQ[2]. The *Crawler* and the *Scanner* are instead implemented as Python modules, which

[1]The source code of DOCKERANALYSER is available on GitHub `https://github.com/di-unipi-socc/DockerAnalyser`.

[2]RabbitMQ Docker image `https://hub.docker.com/_/rabbitmq/`

are shipped in Docker containers based on the official Docker image for Python[3]. Both modules exploit the Python library *pika* [133] for communicating (via AMQP) with the *Message Broker*. *Crawler* uses also the Python library *requests* [135] for interacting with the Docker Hub REST API. The *Scanner* module is configured to import and run the user-defined analysis function. By default, the analysis function is a *void* function that given an image to be analysed it sends the same image to the *Images Service*. The steps needed to execute a custom analysis function in DOCKERANALYSER are listed in Sect. 5.3.2.

Storage. The *Images Database* is implemented as a NoSQL database hosted on a MongoDB container[4]. The *Images Service* is a RESTful API implemented in JavaScript, which is shipped in a container based on the official Docker image for NodeJS[5]. The *Images Service* API provides the HTTP methods for adding, updating, deleting, and searching image descriptions. To do so, it exploits the JavaScript framework *express* [122] to run a web server and *mongoose* [107] sto interact (via mongodb) with the *Images Database*. The *Images Service* API returns the image descriptions as JSON documents.

5.3.2 How to create new Docker image analysers

A user can instantiate DOCKERANALYSER in order to obtain new Docker image analysers. The steps needed for obtaining a new analyser consist of (i) replacing the analysis function, (ii) selecting the crawling policies of the *Crawler* microservice, and (iii) setting the scaling options of the *Scanner* microservice.

The analysis function is replaced by instantiating the *Scanner* microservice of DOCKERANALYSER with a user-defined analysis function. More precisely, the steps needed to instantiate DOCKERANALYSER with a customised analysis function are the following:

1. Clone the GitHub repository of DOCKERANALYSER locally.
2. Create a folder `F` (that represents the *deploy package* – Fig. 5.1) and inside the created folder create the following files:
   a) The `analysis.py` file that contains the code of the custom analysis function,
   b) The `requirements.txt` file that contains the Python library dependencies[6],
   c) Any other file needed by the analysis function (e.g., configuration files)
3. Build the *Scanner* Docker image with Docker Compose [63] by running the command `docker-compose build -build-arg DEPLOY_PACKAGE_PATH=<F> scanner`, (where `F` is the name of the folder created at step 2).

[3]Python Docker image `https://hub.docker.com/_/python/`
[4]MongoDB Docker image `https://hub.docker.com/_/mongo/`
[5]NodeJS Docker image `https://hub.docker.com/_/node/`
[6]The file `requirements.txt` is empty if the implemented function does not have dependencies.

It is worth noting that step 3 builds the Docker image of the *Scanner* with the customised analysis function (contained in the `analysis.py`) that replaces old *Scanner* code. The option `-build-arg DEPLOY_PACKAGE_PATH=<F>` at step 3 copies the folder `F` (containing the custom `analysis.py` file, the `requirements.txt` file and any other files needed to the analysis) into the (old) Docker image of the *Scanner* Docker image hence allowing to create a new image running the new analysis function. The new image of the *Scanner* is indeed built by importing the custom `analysis.py` function and by installing all dependencies listed in the file `requirements.txt` (if any).

```
1  def analysis(image_json, context):
2      logger = context['logger']
3      client_images = context['images']
4      # return True or False
```

LISTING 5.1: Signature of the function defined in the `analysis.py`

The `analysis.py` file stored in the deploy package `F` contains the code of the custom analysis function (written in Python) and it must follow the signature illustrated in Listing 5.1.

- The `image` parameter is a JSON object containing the `name` of the image to be analysed along with other basic fields taken from the registry (some of the most important fields of the JSON object are `is_automated`, `is_official`, `star_count`, and `pull_count`).

- The `context` parameter is a dictionary containing the objects `images` and `logger` (lines 2, 3). The `images` object can be used for interacting with the *Images Service* API. More precisely, the `images` object offers the methods `get_image(name)`, `post_image(json)`, `put_image(json)`, `delete_images(id)` for getting, adding, updating, and deleting an image into the *Images Database*, respectively. The `logger` is the standard *logging.Logger* class of Python and it provides a set of methods for logging the actions during the execution of the code (e.g., `info()`, `warning()`, `error()`, `critical()`, `log()`.

- The return code is a boolean value. `True` must be returned by the function if the image has been processed correctly. `False` must be returned for discarding and deleting the image from the *Message Broker*.

Sect. 5.4 presents two examples of analysis functions that we used to create two different analysers of Docker images.

The second step required to obtain a custom analyser is to set the crawling policy of the *Crawler* microservice for crawling the images from the Docker Hub. The crawling options of the *Crawler* can be found in the `docker-compose.yml` file in the `crawler` service definition. The available configuration options are the following:

`-random` By setting `-random=True` the *Crawler* crawls the images from the Docker registry by randomly selecting them, otherwise it crawls images sequentially.

`-policy` By setting `-policy=stars_first` the *Crawler* crawls the images starting from those with a higher number of stars. Otherwise, by setting `-policy=pulls_first` it crawls first the images with more number of pulls.

`-min-stars` This option permits setting the minimum number of stars (`-min-stars=<integer>`) that an image must have in order to be crawled. All the images with a number of stars less than `-min-stars` are not crawled.

`-min-pulls` This option permits setting the minimum number of pulls (`-min-pulls=<integer>`) that an image must have in order to be crawled. All the images with a number of pulls less than `-min-pulls` are not crawled.

`-only-official` If this option is set, then only the official images stored into the Docker registry are crawled.

`-only-automated` If this option is set, then only the images that are automatically created from a GitHub repository are crawled.

Finally, the user can configure the scaling options of the architecture by setting the number of replicas of the *Scanner* microservice. Running more *Scanner*s in parallel may reduce the time needed to analyse the crawled images. The `deploy` option of the `scanner` in the *docker-compose.yml* file permits specifying the number of parallel *Scanner*s to be started. Listing 5.2 shows an example of a configuration that starts 10 *Scanner*s in parallel when the analyser is started.

```
1  scanner:
2      ...
3      deploy:
4        mode: replicated
5        replicas: 10
```

LISTING 5.2: An example of configuration of the *Scanner* microservice.

In addition, the *Scanner* can be scaled up or down at run time by using the command `docker-compose scale [SERVICE=NUM...]`. For example, the command `docker-compose scale scanner=5` updates the number of *Scanner* replicas to 5.

5.3.3 How to deploy DOCKERANALYSER

DOCKERANALYSER is a multi-container Docker application which can be deployed using the Docker platform. It can be deployed in two different configurations, depending on whether the

target infrastructure is a single host or a cluster of multiple hosts. The *single host deployment* configuration runs all the containers of DOCKERANALYSER in a single node while the *multi host deployment* runs the containers in a cluster of distributed nodes. While former is suitable for running simple and low load analyser, the latter is recommended whenever the analyser requires higher amount of physical resources (e.g., network traffic or storage space) because it permits distributing the load on multiple machines rather than just one.

Single host deployment. Docker Compose [63] permits deploying a multi-container application on a single host if such application is equipped with a *docker-compose.yml* that describes the application deployment. DOCKERANALYSER is equipped with its own *docker-compose.yml* file, and it can hence be deployed on any host supporting Docker Compose. In order to start a newly created analyser, users should submit the command `docker-compose up`.

Multiple host deployment. Docker Swarm [62] permits defining a cluster of Docker engines (called a *swarm*) where to schedule the containers forming a multi-container application. DOCKERANALYSER is equipped with a shell scripts (called *start_swarm.sh*) that allows to start the analyser in a swarm. The script assumes that the user has already configured the swarm where the analyser will be actually executed (as the script itself will have to be executed within the Docker engine managing the swarm). The *start_swarm.sh* script takes as input the name of the deploy package and the name to be assigned to the analyser. The script first creates the *Scanner* image with the deploy package, and it then runs the analyser into the swarm. By default, the script distributes the containers of the analyser among all nodes of the swarm.

5.4 Use cases

In this section, we illustrate how different analysers of Docker images can be created by instantiating DOCKERANALYSER with different user-defined analysis functions. In particular, we present DOCKERFINDER and DOCKERGRAPH, two use cases that run different analysis functions.

DOCKERFINDER analyses an image by running it in a container and checking whether the image provides a list of software distributions. DOCKERGRAPH, instead, creates a graph of Docker images where each node is a name of the repository of an image and where every node is connected to its parent image. The use cases are obtained by replacing the scanner microservice of DOCKERANALYSER (that consist of replacing the *Scanner* Docker image) while the other microservices in the architecture remain untouched. As presented in Sect. 5.3.2 replacing the scanner microservice corresponds to building a new *Scanner* Docker image starting from a user-defined deploy package folder (containing the `analysis.py` file, the `requirements.txt` file, and any other files needed by the analysis function).

5.4.1 DOCKERFINDER

Docker images stored in Docker Hub provide virtually almost any software distributions (e.g., libraries, programming languages, frameworks) to the users. However, the current support for searching such images based on the software distributions they support is missing. Users may want to deploy an application component in a Docker image and that such application requires some specific versions of software distributions (e.g., *Python 2,7*, *Java 1.8*). DOCKERFINDER permits searching for Docker images based on the versions of software distributions they support.

The *deploy-package* folder of DOCKERFINDER contains three files: the `analysis.py` (Listing 5.3), the `requirements.txt` file that contains only the `docker==2.2.1` python library dependency (used by the analysis function for interacting with Docker daemon), and the `software.json` JSON (Listing 5.4) file containing the list of software distributions to be searched in each image. The JSON file contains a list of triples, where each triple is composed of the name of the software distribution, the command to be executed in order to know the version of the software, and a regular expression used to search the matching version (if it exists).

The analysis function of DOCKERFINDER is detailed in Listing 5.3. Lines 1-4 import the Python libraries `json`, `docker`, `re`, and `os` used by the function[7]. Line 6 creates the Docker client object exploited for interacting with Docker daemon. Line 8 defines the `analysis` function that takes as input the image to be analysed and the context. Lines 12-14 pull the image locally using the docker client, then create and start an infinite sleeping container. Lines 16-20 open `software.json` file contained in the deploy package folder and for each software distribution (line 18) takes the command (e.g., *python version*) to be executed and run the command into the already running container. Line 19 the `output` variable that contains the result of execution of the command inside the container. Lines 20-26 use the regular expression to search the version of the software in the output variable (if it exists). Lines 27 adds the software distribution found (if any) in the JSON that will then sent to the images server. Line 28 uses the `client_images.post_image(json)` to post the JSON object containing the results of the analysis into the *Images Service*. Line 29-30 stop the sleeping container and remove it. Line 31 removes also the Docker image analysed. Both the container and the image are removed for freeing storage space.

Users can search Docker images based on the software distributions they support calling the RESTful API of the *Images Service*. The parameters of the *Images Service* API are obtained looking at the fields contained in the JSON object that describe the images analysed. For example, in order to retrieve the Docker images supporting both *Java* and *Python* users can query the *Images Service* with the `GET api/images?python=2.7&java=1.8` method. DOCKERFINDER can be exploited by other tools for obtaining the list of Docker images that satisfy the software distributions required by an application component that needs to be deployed in Docker

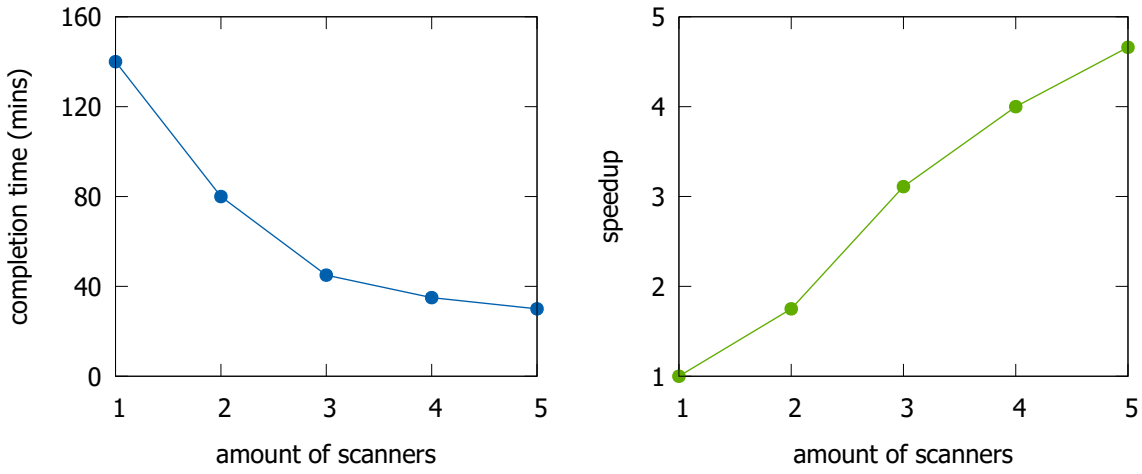
[7]The analysis function can import (in addition to the library present into the `requirements.txt`) any of the standard library provided by Python (e.g., `json`, `re`, `os`).

```python
import json
import docker
import re
import os

client_docker= docker.DockerClient(base_url="unix://var/run/docker.sock")

def analysis(image_json, context):
    logger = context['logger']
    client_images = context['images']
    try:
        image = client_docker.images.pull(images_json['name'])
        container = client_docker.containers.create(images_json['name'],
                                                    entrypoint="sleep  infinity")
        container.start()
        softwares = {}
        with open(os.path.join(os.path.dirname(__file__), 'softwares.json')) as softwares_json:
            software = json.load(softwares_json)
            for sw in software:
                output = container.exec_run(cmd=sw['cmd']).decode()
                match = re.search(sw['regex'], output)
                if match:
                    version = match.group(0)
                    softwares[sw['name']] = match.group(0)
                else:
                    logger.debug("[{0}] NOT found in ".format(sw['name']))

        images_json['softwares'] = softwares
        client_images.post_image(images_json)
        container.stop(timeout=2)
        container.remove()
        client_docker.images.remove(images_json['name'], force=True)
    except docker.errors.ImageNotFound as e:
        logger.exception("{} image not found".format(images_json['name']))
        return False
    return True
```

LISTING 5.3: `analysis.py` function of DOCKERFINDER.

```
[{
   "name": "python",
   "cmd": "python --version",
   "regex": "[0-9]+[.][0-9]*[.0-9]*"
}, {
        "name": "java",
        "cmd": "java -version",
        "regex": "[0-9]+[.][0-9]*[.0-9]*"
}]
```

LISTING 5.4: Some of the software distributions listed in the `software.json`



(a) (b)

FIGURE 5.3. Time performances registered for analysing a set of 100 images randomly sampled from the Docker Hub, where each image was analysed by *Scanner*s by checking the availability of 16 different software distributions. In both plots, the x-axes represent the amount of replicas of *Scanner*s actually running in the running instance of DOCKERFINDER. The y-axes instead represent the (a) completion time and the (b) corresponding speed-up.

containers (e.g., TOSKERISER [26]).

By running DOCKERFINDER, we discovered that the most time consuming task is that of *Scanner*, which have to spend time in downloading images and in analysing them to produce their descriptions. Images can be scanned independently and *Scanner* can hence be easily scaled out to improve the time performances of DOCKERFINDER (as shown[8] in Fig. 5.3). Exploiting the scalability property of microservice-based architecture, and given the fact that DOCKERFINDER is a multi-container Docker application scaling *Scanner*s just corresponds to manually increasing/decreasing the amount of corresponding Docker containers running.

5.4.2 DOCKERGRAPH

Docker permits reusing an already existing image for building other images. An image reused by another image is called parent image. Most of the Docker images stored in Docker Hub are built by reusing already existing images. However, the support for knowing the parent relationship occurring between images is missing. Knowing which are the images that are more used by other images or knowing the images use a single parent image can be exploited for many applications. For example, if a parent image is affected by a security flaw, having the graph of all the images that reused the affected image as parent image can be useful for patching the flaw in such images.

[8]The results displayed in Fig. 5.3 have been obtained by running DOCKERFINDER on a Ubuntu 16.04 LTS workstation having a AMD A8-5600K APU (3.6 GHz) and 4 GBs of RAM.

DOCKERGRAPH can be also used by other tools that require to implement a smart caching policy of images. For instance, the graph can be exploited for maintaining the images locally that are more used as parent image without deleting them.

DOCKERGRAPH constructs a directed graph of images where the nodes are the repository names of images and a link from an image `s` to an image `p` is added if the image `p` is the parent image of *s*. DOCKERGRAPH retrieves the repository name of the parent image by looking at the *FROM* option in the Dockerfile used to create an image.

The deploy package folder of DOCKERGRAPH is composed by the `analysis.py` file and an empty `requirements.txt` because the analysis function requires no external libraries. The `analysis` function of DOCKERGRAPH is shown in Listing 5.5. Lines 1-2 import the *requests* and *re* Python libraries used to interact with the GitHub API and for handling regular expression, respectively. Lines 4-24 defines the `analysis` function of DOCKERGRAPH. Lines 5-6 gets the `logger` and the `client_images` objects. Line 10 checks whether the image has been already analysed by calling the `is_new()` method of the `client_images` object provided by the context. If the image has not been already analysed, line 13 calls the `get_dockerfile(repo)` method that retrieve the image's Dockerfile stored into Docker Hub (if it is present). Line 14 calls `extract_FROM(dockerfile)` method that use a regular expression for extracting the *FROM* option present into the Dockerfile. It returns a couple of strings where the first is the repository name and the second is the tag of the parent image. Lines 15 adds the repository name (`from_repo`) and the tag (`from_tag`) of the parent image name Python dictionary describing the image. In line 17 the `client_images.post_image(JSON)` method is called to add the `node_image` dictionary to the *Images Service*.

DOCKERGRAPH has been executed[9] to crawl sequentially the repository stored in Docker Hub. At the time of executing DOCKERGRAPH, Docker Hub contained approximately 600000 repositories. The graph constructed by DOCKERGRAPH counts 87570 repository names because DOCKERGRAPH discarded all the repositories in Docker Hub whose Dockerfiles are not present or badly formatted. Fig. 5.4 shows only the top 10 images used as parent images by other images. The most used image is *ubuntu* with 15208 images using it as parent image, while *nginx* is used as parent image by 1697 images.

It is worth nothing that the image descriptions obtained by an analyser are returned as raw data (i.e., JSON objects). It is left to the users to post-process the raw data and to visualise it (like in the chart in Fig. 5.4) using data visualisation tools. The data visualisation of the obtained images descriptions is outside of the scope of this work.

[9] DOCKERFINDER has been executed on a Ubuntu 16.04 LTS workstation having a AMD A8-5600K APU (3.6 GHz) and 4 GBs of RAM.

```
import requests
import re

def analysis(image_json, context):
    logger = context['logger']
    client_images = context['images']

    repo = image_json["repo_name"]
    logger.info("Received image to be analysed: {} ".format(repo))
    if client_images.is_new(repo):
        node_image = {'name': repo}
        try:
            dockerfile = get_dockerfile(repo)
            from_repo, from_tag = extract_FROM(dockerfile)
            node_image['from_repo'] = from_repo
            node_image['from_tag'] = from_tag
            client_images.post_image(node_image)
        except ValueError as e:
            logger.error(str(e))
            return False
        return True
    else:
        logger.info("{}  already present into local database ".format(repo))
        return False

def extract_FROM(dockerfile):
    search = re.search('FROM ([^\s]+)', dockerfile)
    if search:
        from_image = search.group(1)
        if ":" in from_image:
            from_repo, from_tag = from_image.split(":")
        else:
            from_repo = from_image
            from_tag = None
        return from_repo, from_tag
    else:
        raise ValueError("FROM value not found in DockerFile")


def get_dockerfile(repo_name):

    docker_url = "https://hub.docker.com/v2/repositories/{}/dockerfile/"
    try:
        response = requests.get(docker_url.format(repo_name))
        dockerfile = response.json()['contents']
        return dockerfile
    except ConnectionError as e:
        raise e
```

LISTING 5.5: analysis.py function of DOCKERGRAPH

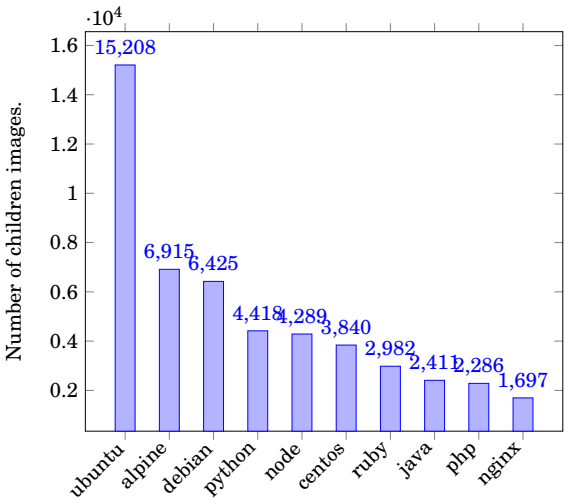


FIGURE 5.4. Top ten Docker images used as parent images.

5.4.3 Discussion

In this section, we discuss the advantages of using DOCKERANALYSER versus building Docker images analysers from scratch. We evaluated the usefulness of DOCKERANALYSER by considering (i) the number of functionalities to be developed, (ii) the reusability of the code, and (iii) the time required to obtain new analysers starting from already existing analysers.

DOCKERANALYSER reduces the number of functionalities to be developed with respect to build the analyser from scratch. Table 5.1 reports four of the main functionalities that an analyser should support: *Image crawling* is how the images are crawled from a Docker registry, *Image analysis* is how the images are analysed, *Storage of results* is how the results are stored, and *Scalable architecture* is how to implement a scalable architecture. As shown by Table 5.1, DOCKERANALYSER considerably simplifies the building of analysers by requiring to only account for the *Image analysis*, as the others functionalities are provided by the architecture. *Image crawling* is carried out by the *Crawler*, *Storage of results* is provided by the local storage database, and the *Scalable architecture* is permitted by scaling the *Scanner* microservice. Instead, building an analyser from scratch would require to develop all the four functionalities.

We also evaluated the reusability of DOCKERANALYSER by considering both the number of reusable components and the amount of reusable code of the architecture. DOCKERANALYSER is composed by five components (*Crawler*, *Message Broker*, *Scanner*, *Images Service*, and *Images Database*). A new analyser is obtained from DOCKERANALYSER by only replacing the *Scanner* component, hence reusing the other four components. We also evaluated the amount of reusable code with the following metric (taken from [68]):

$$R = \frac{\texttt{lines of reused code}}{\texttt{total lines of code}}$$

We exploited such metric for evaluating the percentage of reused code for both the analysers we

	Using DockerAnalyser	From Scratch
Image crawling	no	yes
Image analysis	yes	yes
Storage of results	no	yes
Scalable architecture	no	yes

TABLE 5.1. Functionalities to implement during the design of new Docker images analysers.

developed, viz., DOCKERFINDER (R_{df}) and DOCKERGRAPH (R_{dg}). For both DOCKERFINDER and DOCKERGRAPH the reused code is around 94%:

$$R_{df} = \frac{loc_{da}}{loc_{df}} = \frac{1861}{1971} = 0.944$$
$$R_{dg} = \frac{loc_{da}}{loc_{dg}} = \frac{1861}{1976} = 0.941$$

Notice that, in both formulas, the value of `lines of reused code` is the amount loc_{da} of lines of code of DOCKERANALYSER (which are all included also in both the analysers we developed). The values of `total lines of code` are instead the amounts loc_{df} and loc_{dg} of all lines of code of DOCKERFINDER and DOCKERGRAPH, respectively.

Finally, DOCKERANALYSER can reduce the time required to obtain new analysers starting from already existing analysers. For example, by reusing the code of DOCKERFINDER, a user may create a new analyser by only modifying the files `analysis.py` and `software.json` that we provided. She can customise the commands executed by DOCKERFINDER by modifying the `software.json` file or she can modify the lines 15-25 of Listing 5.3 in order to execute a different type of analysis on the images.

5.5 Related work

MicroBadger [117] is an on-line service that shows the contents of public Docker images, including metadata and layer information. Using MicroBadger a user can also add personalized metadata to images in order to retrieve them successively. MicroBadger differs from DOCKERANALYSER because it only permits to assign metadata to images but it does not provide a way to run customised analysis of Docker images.

Another approach that allows assigning custom properties to Docker images is JFrog [97]. JFrog's Artifactory is a universal Artefact Repository working as a single access point to software packages including Docker. JFrog can search Docker images by their name, tag or digest. Users can also assign custom properties to images, which can then be exploited to specify and resolve

queries. JFrog differs from DOCKERANALYSER since permits only to assign manually custom metadata to images. DOCKERANALYSER architecture fully automates the process of assigning properties to the images based on what they feature.

Works in [86, 93, 105, 149] are frameworks that follow the serverless architecture [140] for running custom functions. The serverless functions are functions written in any language that are mapped to event triggers (e.g., HTTP requests) and scaled when needed.

Snafu [149], or Snake Functions, is a modular system to host, execute and manage language-level functions offered as stateless microservices to diverse external triggers. Functions can be executed in-memory/in-process, through external interpreters (Python 2, Java), and dynamically allocated Docker containers.

OpenLambda [86] is an Apache-licensed serverless computing project, written in Go and based on Linux containers. One of the goals of OpenLambda is to enable exploration of new approaches to serverless computing.

kubeless [105] is a Kubernetes-native serverless framework. Kubeless permits creating functions and run them on a in-cluster controller that watches and launches the functions on-demand.

IronFunctions [93] is an open source serverless platform. It can run any languages as functions and it supports AWS lambda format. Prerequisites: Docker 1.12 or later installed and running.

The common characteristic of DOCKERANALYSER and serverless architectures is that both approaches allow users to provide only an analysis function while the architecture is responsible to run, scale, and manage the execution of such function. DOCKERANALYSER differs from serverless architecture because it provides also an internal storage where the description of the images produced by the analysis function are stored. The previous approaches of serverless architecture, instead, do not provide any support for storing the results of the functions.

Our work shares with Wettinger et al. [161] the general objective of contributing to ease the discovery of DevOps "knowledge" (which includes Docker images). [161] proposes a collaborative approach to store DevOps knowledge in a shared, taxonomy-based knowledge base. More precisely, [161] proposes to build the knowledge-base in a semi-automated way, by (automatically) crawling heterogeneous artefacts from different sources, and by requiring DevOps experts to share their knowledge and (manually) associate metadata to the artefacts in the knowledge-base. DOCKERANALYSER instead focuses only on container images, and it permits building analyser that creates description of such images in a fully-automated way.

Finally, is worth noting that there exist solutions that try resolve the problems addressed by the two use cases presented in Sect. 5.4.

Docker Store [61] is a repository containing trusted and verified Docker images. Similar to Docker Hub, Docker store offers a search web-based interface that returns the images that match the image name, description, or the publisher name. In addition, Docker Store permits limiting the results by category (e.g., programming languages, base images, Operating System).

With Docker Store it is not possible to distinguish, for instance, whether an image support a software distribution (e.g., Python, Java) since all images supporting such languages fall in the same category. DOCKERFINDER does not suffer of the same limitation, as it permits explicitly searching for images supporting either Java or Python, or both.

ImageLayers [116] is an on-line service that analyses Docker images stored in Docker Hub and shows the layers that compose them and layers that are shared by multiple images. While *ImageLayers* considers the layers composing an image, DOCKERGRAPH instead considers the parent image of an image. DOCKERGRAPH permits analysing all the images contained in a Docker Registry and constructs a graph of images. *ImageLayers* permits only analysing the layers a single image at the time and returns a flat description of a single image.

5.6 Conclusions

Docker images are stored in Docker Registries that allow to add, remove, distribute, and search such images. Images stored inside Docker registries are described by fixed attributes (e.g., name, description, owner of the image), which may not be enough to permit users to select the images satisfying their needs. Currently, users are required to manually download the images from remote registries and look for images satisfying the desired functionalities.

In order to solve the aforementioned problem, we presented DOCKERANALYSER a tool to build customised analysers of Docker images in a fully automated way. Users are required to provide only the analysis function and any other files needed by the analysis function, whilst DOCKERANALYSER disposes of the functionalities for crawling the images from a Docker registry, running the provided analysis on every image, storing the results of the analysis in a local storage, and searching the obtained results.

We believe that the actual value of DOCKERANALYSER is that it can be exploited by users (e.g., researchers, developers and data miners) interested in building their own analysers of Docker images. Users are only required to provide the analysis function, in the form of a Python function that, given the name of a Docker image, scans such image to extract some metadata for generating the description of the image.

We identified three main classes of analysers that can be obtained from DOCKERANALYSER, namely:

1. Analysers that execute commands inside Docker images for extracting features,
2. analysers that inspect the source code of Docker images, and
3. analysers that scan the compiled/binaries version of Docker images.

In this chapter, we have shown a concrete example of analyser for class (1) and a concrete example for class (2). For (1) we presented DOCKERFINDER, an analyser that extracts the

versions of the software supported by the images. For (2) we presented DOCKERGRAPH that analyses the source code stored in the GitHub repository (the Dockerfile of the image) in order to construct a graph of *parent* images. The development of an analyser in class (3), and (more generally) the development of other analysers and the identification of other classes of analysers that can be defined is left for future work.

The use cases also showed that the choice of implementing DOCKERANALYSER with a microservice-based architecture eases building customisable and scalable analysers. We indeed experimented the benefits of the scalability and replaceability properties of microservice-based architectures. In particular, replaceability allows obtaining DOCKERFINDER and DOCKERGRAPH by only replacing the Docker image of the scanner microservice. Instead, by exploiting the scalability property, we scaled the number of *Scanner* microservices of DOCKERFINDER in order to reduce the time needed to analyse the images.

We believe that DOCKERANALYSER can also take advantage of the extensibility property of microservice-based architectures that permit adding new microservices. For instance, DOCKERANALYSER can be extended with a *checker* microservice (such as in [27]) that maintains the consistency of the images stored in the local storage of DOCKERFINDER and those in Docker Hub.

As part of our future work we want to build DOCKERANALYSER as a web-based service where users can upload the *deploy package* folder F through a GUI and the web-based service creates the analyser with the provided *deploy package*, starts the analyser, and visualises the obtained images descriptions in a dashboard (e.g., with customisable charts, like that in Fig. 5.4).

In addition, we plan to extend DOCKERANALYSER in such a way, that (i) it permits analysing other container-based technologies (such as [50, 87]), and (ii) it permits specifying the analysis function in other programming languages (e.g., Java, Go, Bash), Finally, we plan also to implement other analysers of Docker images. For instance, a security analyser can be built by using one of the existing static analyser of Docker images (such as [49]) in order to analyse the images stored in Docker Hub discovering the images affected by security flaws.

CHAPTER 6

Component-aware Orchestration of Cloud-based Enterprise Applications, from TOSCA to Docker and Kubernetes

Enterprise IT is currently facing the challenge of coordinating the management of complex, multi-component applications across heterogeneous cloud platforms. Containers and container orchestrators provide a valuable solution to deploy multi-component applications over cloud platforms, by coupling the lifecycle of each application component to that of its hosting container. We hereby propose a solution for going beyond such a coupling, based on the OASIS standard TOSCA and on Docker. We indeed propose a novel approach for deploying multi-component applications on top of existing container orchestrators, which allows to manage each component independently from the container used to run it. We also present prototype tools implementing our approach, and we show how we effectively exploited them to carry out two concrete case studies.

The results in this Chapter are currently submitted for publication [20].

6.1 Introduction

Cloud computing is a flexible, cost-effective and proven delivery platform for running on-demand distributed applications [65]. To fully exploit the potentials of cloud computing and facilitate the scalability, reliability and portability of applications, various cloud-native architectural styles have emerged (with microservices being one of the most prominent examples). This has resulted in a growth of the complexity of applications, which nowadays integrate dozens (if not hundreds) of interacting components [147]. The problem of automating the deployment and management of such complex, multi-component applications over heterogeneous cloud platforms has hence

become one of the main challenges in enterprise IT [45, 96].

The components forming an application are typically deployed on cloud platforms by relying on virtualisation technologies. Container-based virtualisation [148] is getting more and more momentum in this scenario, as it provides an isolated and lightweight virtual runtime environment [127]. Docker (`https://www.docker.com`) constitutes the *de-facto* standard for container-based virtualisation, and it permits packaging software components (together with all software dependencies they need to run) in Docker images, which are then exploited as read-only templates to create and run Docker containers. Container orchestrators are then used to automate the deployment and management of containerised applications at a large scale. Docker Swarm (`https://docs.docker.com/engine/swarm`) and Kubernetes (`https://kubernetes.io`) are currently the most popular solutions for orchestrating Docker containers, providing all necessary abstractions for scaling, discovering, load-balancing and interconnecting Docker containers over single and multi-host systems [88, 114].

Docker containers are however treated as a sort of "black-boxes", since they constitute the minimal entity that can be orchestrated. Container orchestrators can indeed create, start, stop and delete containers, but they do not provide support for coordinating the management of the components running within containers. The lifecycle of the software components forming a containerised application is hence tightly coupled to that of their hosting containers. For instance, when the orchestrator creates and starts a container, all the software components it contains have to be created and started as well, as the orchestrator does not provide a support for creating or starting them afterwards. The same holds when containers are stopped or deleted, as the components they are hosting get stopped and deleted as well. This is because currently existing container orchestrators do not provide a support for coordinating the management of software components independently from that of their hosting containers [30].

Decoupling the management of application components from that of the containers hosting them can anyway bring various advantages. For instance, this allows to employ Docker containers as so-called *system containers*, i.e., a lightweight portable alternative to virtual machines [158]. It also enables inter-process communication within the components running in the same container, without requiring them to necessarily communicate over the network [46]. This helps saving resources, thus containing costs or enabling a fine-grained orchestration on infrastructures where computing and networking resources are costly and limited, e.g., in edge clusters [129]. Other known advantages of decoupling application components from their hosting containers are maintainability and reuse [26]. Deployment requirements of multi-component applications change over time. If components are coupled to their hosting containers, this requires to repackage them from scratch whenever their deployment requirements change.

Following this idea, we hereafter propose a solution for managing the software components forming an application independently from the containers used to run them. The proposed solution is intended to allow a more flexible, component-aware management of multi-component

applications on top of existing Docker-based container orchestrators, and it does so by relying on the OASIS standard TOSCA for specifying and orchestrating multi-component applications. More precisely, starting from an existing representation of multi-component applications in TOSCA (taken from our previous work [30]), we provide the following contributions:

- We propose a novel approach for managing the lifecycle of software components forming a multi-component application independently from that of the containers used to host such components.
- We present the prototype tools implementing our approach. These include a service enabling the component-aware runtime management of multi-component applications and a packager for generating the deployment artifacts needed to ship and manage applications on existing Docker-based container orchestrators.
- We showcase the effectiveness of our approach and prototype tools by reporting on how we exploited them to run two concrete case studies based on existing benchmark applications.

It is worth highlighting that our solution is not intended to implement a new orchestrator from scratch, as for instance we did with the TosKer orchestration engine [30]. We instead aim at enabling a component-aware management of multi-component applications on top of existing, production-ready container orchestrator (e.g., Docker Swarm or Kubernetes), in order to exploit all features they already provide, e.g., multi-host deployment and network overlays. In this way we do not need to re-implement such features from scratch, and we can hence focus on seamlessly extending the orchestration support they provide.

6.2 Background

6.2.1 TOSCA

TOSCA [124] (*Topology and Orchestration Specification for Cloud Applications*) is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based and machine-readable modelling language that allows to describe cloud applications. Obtained specifications can then be processed to automate the deployment and management of the specified applications.

TOSCA allows to specify a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 6.1). The topology template is a typed directed graph that describes the topological structure of a multi-component application. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relations occurring among such components.

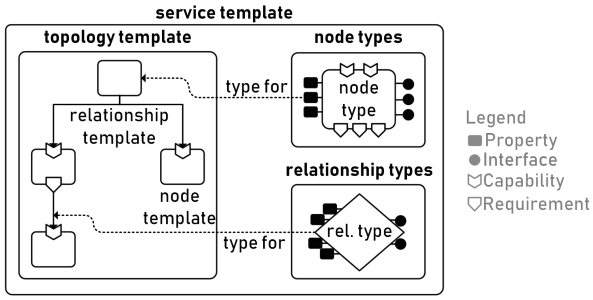


FIGURE 6.1. The TOSCA metamodel [124].

Node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components' requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components. As the TOSCA type system supports inheritance, a node/relationship type can be defined by extending another, thus allowing the former to inherit the latter's properties, requirements, capabilities, interfaces, and operations (if any).

Node templates and relationship templates also specify the artifacts needed to actually realise their deployment or to implement their management operations. As TOSCA allows artifacts to represent contents of any type (e.g., scripts, executables, images, configuration files, etc.), the metadata needed to properly access and process them is described by means of artifact types.

Finally, to enable their actual deployment, TOSCA applications are packaged and distributed in CSARs (*Cloud Service ARchives*). A CSAR is a zip archive containing an application specification along with the concrete artifacts realising the deployment and management operations of its components.

6.2.2 Modelling Multi-component, Docker-based Applications with TOSCA

In our previous work [31], we defined a TOSCA-based representation for specifying the software components forming an application, as well as the Docker containers and Docker volumes used to form their runtime infrastructure. More precisely, three different TOSCA node types (Fig. 6.2) allow to distinguish among the Docker containers, Docker volumes and software components forming a multi-component application.

- *tosker.nodes.Container* allows to describe Docker containers, by indicating whether a container requires a *connection* (to another Docker container or to an application component), whether it has a generic *dependency* on another node in the topology, or whether it needs some persistent *storage* (hence requiring to be attached to a Docker volume). *tosker.no-*

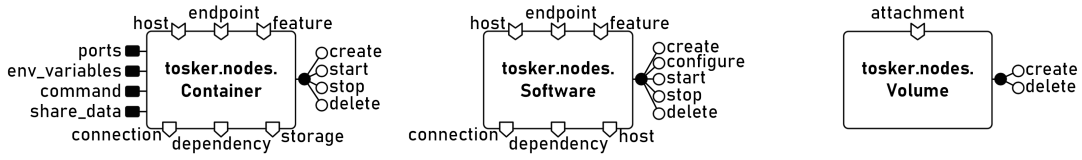


FIGURE 6.2. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

des.Container also permits indicating whether a container can *host* an application component, whether it offers an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of another container/application component). To complete the description, *tosker.nodes.Container* can contain properties (*ports*, *env_variables*, *command*, and *share_data*, respectively) for specifying the port mappings, the environment variables, the command to be executed when running the corresponding Docker container, the list of files and folders to share with the host. Finally, *tosker.nodes.Container* lists the operations to manage a container (which corresponds to the basic operations offered by the Docker platform to manage Docker containers [114]).

- *tosker.nodes.Volume* allows to specify Docker volumes, and it defines a capability *attachment* to indicate that a Docker volume can satisfy the *storage* requirements of Docker containers. It also lists the operations to manage a Docker volume (which corresponds to the operations to create and delete Docker volumes offered by the Docker platform [114]).

- *tosker.nodes.Software* allows to represent the software components forming a multi-component application. It allows to specify whether an application component requires a *connection* (to a Docker container or to another application component), whether it has a generic *dependency* on another node in the topology, and that it has to be *host*ed on a Docker container or on another component. *tosker.nodes.Software* also permits indicating whether an application component can *host* another application component, whether it provides an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of a container/application component). Finally, *tosker.nodes.Software* lists the operations to manage an application component by exploiting the TOSCA standard lifecycle interface [124] (viz., *create*, *configure*, *start*, *stop*, *delete*).

The interconnections and interdependencies among the nodes forming a multi-component application can instead be specified by exploiting the TOSCA normative relationship types [124]. The relationship type *tosca.relationships.AttachesTo* allows to attach a Docker volume to a Docker container. *tosca.relationships.ConnectsTo* allows to describe the network connections to establish between Docker containers and/or application components. *tosca.relationships.HostedOn* allows to indicate that an application component is hosted on another component or on a Docker container

(e.g., to indicate that a web service is hosted on a web server, which is in turn hosted on a Docker container). Finally, *tosca.relationships.DependsOn* allows to represent generic dependencies between the nodes of a multi-component application (e.g., to denote that a component must be deployed before another, as the latter depends on the availability of the former to properly work).

Concrete examples of modelling of multi-component applications with the above recapped TOSCA representation can be found in Sect. 6.7.

6.3 Bird-eye view of our approach

Our ultimate objective is to enable a component-aware management of multi-component applications by piggybacking on existing container orchestrators (such as Docker Swarm and Kubernetes). We indeed aim at seamlessly extending the support they provide for orchestrating containers, so that the lifecycle of application components is not entangled to that of their hosting containers, but rather allowing components to be managed independently.

Fig. 6.3 provides an high-level overview on the orchestration approach we propose. The input is the TOSCA specification of a multi-component application. In the figure, the considered application is composed by three services (i.e., *s1*, *s2* and *s3*) and three containers (i.e., *c1*, *c2* and *c3*). Containers *c1* and *c2* are used as system containers [158], i.e., as lightweight virtual environments for hosting services *s1* and *s2* and service *s3*, respectively. The container *c3* is instead a standalone container running its default main process. Services *s1* and *s3* also connect to *s2* and to *s2* and *c3* to deliver their businesses.

Given such a TOSCA application specification, a *Packager* generates the Docker Compose file allowing to deploy the application on top of Docker-compliant container orchestrators. The obtained artifact not only packages the components forming an application within the containers hosting them, but also seamlessly extends the application deployment by introducing some additional components enabling the desired component-aware orchestration.

- A *Unit* is included in each container packaging some application component (i.e., *c1* and *c2*). Each *Unit* is responsible of managing the lifecycle of the components packaged within the container it appears in, by launching the concrete artifacts (e.g., bash scripts) implementing their management operations.

- A *Manager* is included and packaged within a newly added container (i.e., *c4*). The *Manager* is responsible of interacting and coordinating the *Unit*s, in order to manage the components forming an application, and based on the commands issued by an application administrator.

With our approach, an application administrator can continue to deploy and manage the containers in an application by exploiting existing container orchestrators. She can then deploy and manage each software component forming her application by issuing commands to the

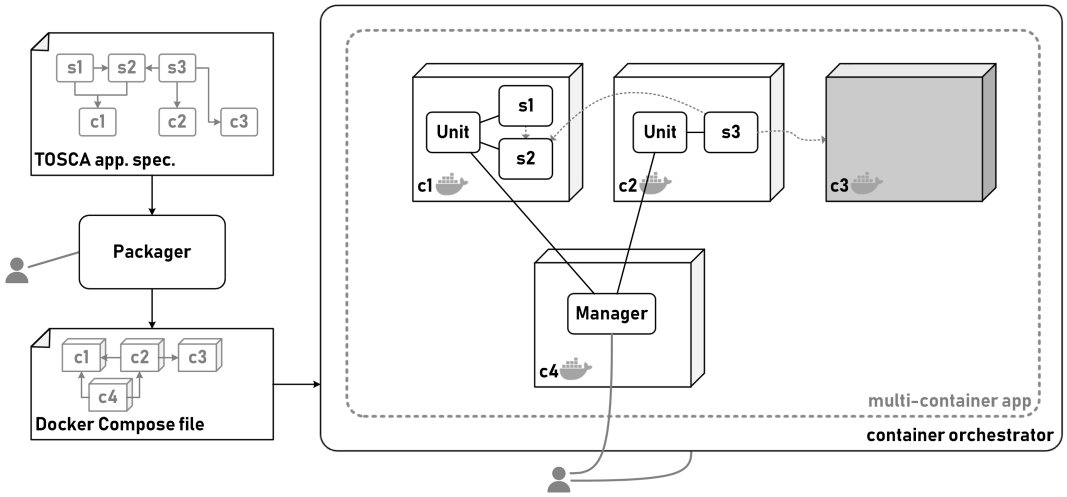


FIGURE 6.3. Bird-eye view of our approach for enabling a component-aware orchestration of the management multi-component applications on top of existing container orchestrators.

containerised *Manager*, which will then properly forward it to the *Unit*s to enact the corresponding management operation. For instance, once the containers in Fig. 6.3 are all running, an application administrator may issue the commands to install and start service *s1* to the *Manager*. The latter will forward the corresponding requests to the *Unit* in the container *c1*, which will launch the bash scripts implementing the to-be-enacted management operations.

It is worth noting that our approach seamlessly integrates with any Docker-compliant container orchestrator, as the newly introduced components (i.e., the *Manager* and the *Unit*s) are themselves packaged within Docker containers. In addition, by piggybacking on existing container orchestrator, our approach allows to uniformly manage containers, independently on whether they are used as lightweight virtual hosting environments for application components (as *c1* e *c2* in the figure) or whether they are used as standalone services (as *c3* in the figure). This allows application administrators to choose whether to enable the fine-grain management of components or to couple their lifecycle to the containers they appear in. In the former case, she has to distinguish components and containers in the TOSCA application specification (and provide the artifacts implementing the lifecycle operations of such components), while in the latter case she just packages the components within the corresponding containers (by writing proper Dockerfiles — as she would be doing with current Docker-based orchestrators).

In the following, we illustrate how we concretely obtained the above illustrated orchestration solution, by designing and developing the TOSKOSE open-source toolset. We first show the issues and design choices for allowing *Units* to independently manage the software components packaged within a container (Sect. 6.4). We then show how we designed and developed the *Manager* to enable the orchestration of multi-component applications on top of existing orchestrators

(Sect. 6.5), and how we implemented the *Packager* for generating a Docker-based artifacts from the TOSCA specification of an application (Sect. 6.6).

6.4 Managing containerised components

6.4.1 Managing Multiple Components in a Container

The first challenge to tackle for enabling an approach like ours consists in allowing multiple components to coexist within a same Docker container, with each component also being independently manageable from the other components and from the container hosting them. Docker envisions the possibility of running multiple components in the same container [3], by also recommending two possible solutions for doing so.

A naïve solution consists in wrapping all commands to install, configure and start the components to be hosted on a Docker container in a shell script. Bash job control can also be exploited to write down the shell script, e.g., to delay the starting of some components, or to execute processes implementing some of their management operations. The obtained shell script can then be executed as the main process of the container, hence meaning that the container continues to run until the shell script continues to run. This means that the shell script must not terminate until at least one of the components running in the container has to continue to run. Hence, even if the naïve solution allows to run multiple components in the same container, their management lifecycle is still coupled to that of their hosting container. Another example supporting this statement comes from the restarting of a component that got stopped, which requires an application administrator to force a new execution of the shell script by tearing down its hosting container and starting a new container. The naïve approach indeed does not support remotely orchestrating the lifecycle of application components, in a way that is independent from the lifecycle of their hosting containers.

The other suggested solution is to exploit a process management system, like *Supervisor*. *Supervisor* (`http://supervisord.org`) allows users to control multiple processes on Unix-like operating systems, based on a client/server model. A server (called *supervisord*) is indeed responsible of starting sub-processes on demand, under client invocation or for handling some events (e.g., for restarting a child process that crashed or exited). *supervisord* manages each spawned sub-process for the entirety of its lifetime, by also taking care of signal management, logging and configuration (including auto-starting and restarting policies). Clients can then ask *supervisord* to spawn sub-processes through a command-line interface (called *supervisorctl*) or via a XML-RPC API (served by an HTTP server). Users can also customise *supervisord* by exploiting a configuration file, called *supervisord.conf*. The latter is loaded when Supevisor starts, and it is exploited to configure *supervisord*, *supervisorctl* and the HTTP-served XML-RPC API. This includes the definition of so-called `program` section, allowing to define and configure sub-processes that can be spawned on demand.

The *Supervisor*-based solution is hence more suited to our needs. Ad-hoc `programs` can be defined to allow independently executing the artifacts implementing the management operations of the components (with each `program` section devoted to a different lifecycle operation of a different component). Such operations can then be remotely orchestrated using the XML-RPC API natively offered by *Supervisor*. Following this initial idea, we hereafter show how we exploited *Supervisor* to implement the *Unit*s in our orchestration approach (Fig. 6.3). More precisely, we first discuss the issues characterising the management of multiple components within a same container and how *Supervisor* can help solving them (Sect. 6.4.2). We then illustrate how we implement *Unit*s as *Supervisor* instances (Sect. 6.4.3).

6.4.2 Signal Management and Zombie Reaping

Managing multiple components in a same container inherently requires to be able to manage multiple processes in such container. A process indeed runs as the main process of the container (i.e., the process with PID 1), and each operation to manage a component requires to spawn a corresponding sub-process, e.g., executing the shell script implementing such operation. Two subsequent, potential issues must be taken into account while managing multiple processes within the same container, i.e., *signal management* and *zombie reaping* [47].

6.4.2.1 Signal Management

Signals sent to a Docker container are forwarded to its main process, which has hence to be configured so as to allow it to decide whether and how to forward them to its sub-processes. A striking example in this direction is the following: Suppose that a Docker container is stopped, by issuing the `docker stop` command. The latter sends a `SIGTERM` to the main process of the container for terminating its execution [4]. If the main process has not been configured to handle `SIGTERM`, it does not forward such signal to its child processes, which are hence not aware that the container is going to be dismissed. Afterwards, the Docker runtime dismisses the container by sending a `SIGKILL` signal, which results in killing uncleanly all processes running within the container. `SIGKILL` cannot be trapped, blocked or ignored and the processes are interrupted abnormally, possibly causing inconsistencies or data corruption [118].

The main process of the container has hence to be configured to handle the signals it receives, by properly forwarding them to the processes running in the container. If adopting the naïve solution, this drastically complicates the writing of the shell script running as the main process of the Docker container, for which we would still have the issue of not being able to remotely orchestrate the lifecycle of the components running within the container. *Supervisor* instead natively supports signal management, hence making it more suitable to our needs.

6.4.2.2 Zombie Reaping

Another challenge while trying to manage multiple processes within the same Docker container comes from the well-know PID 1 zombie reaping problem. In Unix-like systems, zombie processes are processes in a terminated state, waiting for their parent to exit completely and get their descriptor removed from the process table. The descriptor of a terminate process is kept in the process table until its parent reads its exit status and remove its descriptor from the table, hence "reaping the zombie" process [118]. Unix-based systems are typically provided with full-fledged PID 1 processes (e.g., *systemd* in Debian), which support routines for reaping zombie processes. However, the PID 1 process of a Docker container is user-defined, and typically consists in the main process of the application running within the container. The latter typically does not feature any routine addressing the zombie reaping problem [47].

Zombie processes are however harmful. Even if they are only consuming a little amount of memory to store their process descriptor, they keep their PID occupied. As Unix-like systems have a finite pool of PIDs, if zombies are accumulated at a very quick rate, the pool of available PIDs can be rapidly exhausted. This would result in preventing the spawning of other processes [118]. Given that we aim to managing multiple components within the same container, and since each call to lifecycle operation of a component results in new processes getting launched, zombie reaping has to be addressed [47]. However, neither the naïve solution nor that based on *Supervisor* are supporting zombie reaping [3].

6.4.2.3 Existing Approaches

Several solutions are addressing signal management and zombie reaping in Docker containers, with *dumb-init* (`https://github.com/Yelp/dumb-init`), *baseimage-docker* (`https://github.com/phusion/baseimage-docker`) and *tini* (`https://github.com/krallin/tini`) perhaps being the most prominent examples. They all share the baseline idea of wrapping the main process of a Docker container with a process acting as a proxy for signals and featuring a routing addressing zombie reaping. From developers' viewpoint, they provide a much lighter and usable solution with respect to including full-fledged init process systems (e.g., *sysvinit*, *upstart* or *systemd*) within Docker containers [3].

The usage of *dumb-init* and *tini* is analogous. Both must be packaged within a Docker image and they must be used as main processes (wrapping the main process of the application) when a container is spawned from such image. *dumb-init* and *tini* will then take care of zombie reaping in a seamless way, by also forwarding signals to the process they wrap. Differently from *dumb-init*, *tini* is integrated with the Docker platform (since version 1.13). A boolean flag `--init` is supported by the commands `dockerd` and `docker run`, which allows to seamlessly feature signal forwarding and zombie reaping in a container spawned from an existing image, backed by *tini*.

Solutions like *dumb-init* and *tini* allow to resolve issues related to signal management and zombie reaping, but they however still lack of other essential features, e.g., remote control and

logging. *baseimage-docker* is a step forward in this direction, as it offers an all-in-one Docker image based on Ubuntu, featuring an init process for signal management and zombie reaping.

6.4.3 Our Solution

For addressing both signal management and zombie reaping, we exploited the *Supervisor*-based solution recommended by Docker [3], in conjunction with *tini* (as the latter is already fully integrated with Docker). More precisely, we use *tini* as the main process of Docker containers, wrapping a *Supervisor* instance implementing the *Unit*s of our approach. In this way, *tini* cares about zombie reaping, and *Supervisor* cares about signal management, while at the same time enabling to remotely manage multiple processes within a same container. Such an approach brings other valuable advantages with respect to the main competitor among other existing approaches, i.e., *baseimage-docker*. Among such advantages, two are worth highligthing:

- *baseimage-docker* is coming only with a given distribution of Ubuntu, while *Supervisor* and *tini* work with any operating system distribution featured by a Docker container. This hence makes our approach applicable to a wider set of scenaria.
- `baseimage-docker` only support SSH to remotely access the internals of a container. *Supervisor* instead exposes a customisable XML-RPC API on top of a HTTP server. The API exposes methods for managing the lifecycle of both `supervisord` and its child processes, and it can be customised by exploiting an external INI-style configuration file. In particular, it is possible to define `program` sections, which result in offering remotely accessible methods that can be invoked on demand. The latter acts as an enabler for our orchestration approach, as the management operations associated with the components of an application can be implemented as *Supervisor* `programs`.

*Unit*s are implemented by packaging a (*tini*-wrapped) standalone instance of *Supervisor* in each container running one or more application components. It runs as the main process of the container, and it is configured to allow executing (on demand) the artifacts implementing the management operations of the components hosted by the container. The latter is obtained by providing the *Supervisor* instance with a configuration file containing a different `program` section for each management operation supported by the components hosted by the container, hence configuring the XML-RPC API of the *Supervisor* instance to feature a remotely accessible operation for each management operation of hosted components. The configuration file is automatically generated from the TOSCA specification of an application, and it is automatically packaged within each container of an application together with *Supervisor* (Sect. 6.6).

To enable the packaging of a standalone instance of *Supervisor*, we developed TOSKOSE UNIT (`https://github.com/di-unipi-socc/toskose-unit`). TOSKOSE UNIT is a Docker image bundling a standalone instance of *Supervisor*, which is publicly available on the Docker Hub (`https://hub.docker.com/r/diunipisocc/toskose-unit`). Its purpose is to allow including a

suitably configured instance of *Supervisor* in any Docker image of the containers forming the infrastructure of an application, which can be obtained by means of multi-stage Docker builds.

While developing TOSKOSE UNIT, we had to address an issue inherent to the usage of *Supervisor* within a Docker container. *Supervisor* is a Python application, hence requiring Python runtime to be available in the container running it. However, installing a Python interpreter in a Docker image can generate conflicts, if the Docker image is already featuring some Python interpreter. To address this issue, we exploited the PyInstaller "freezing" tool. PyInstaller "freezes" an existing Python program by creating a single-file executable that contains the application code and the Python interpreter to run it. In this way, we "freezed" *Supervisor* and created a bundle not needing any Python interpreter or module to be installed in the Docker containers running it, hence avoiding the risk for conflicts.

6.5 Orchestrating multi-component applications

The second challenge to tackle consists in allowing to seamlessly manage the lifecycle of the components forming a TOSCA application, by enabling the remote invocation of their management operations, and by running them on top of existing Docker container orchestrators. Of course, the latter is because we wish to avoid reinventing the wheel, i.e., instead of re-designing a container orchestrator from scratch, we wish to piggyback on top of existing, production-ready orchestrators, also for allowing to reuse the capabilities they feature.

Existing container orchestrators already allow deploying and managing containers. For instance, given a specification of the containers to run and of their configuration, both Docker Swarm and Kubernetes can deploy such containers on a cluster of hosts by also configuring them as indicated. Docker Swarm and Kubernetes then proceed by orchestrating deployed containers, e.g., by applying them load balancing and scaling policies, for recovering failed instances, and to manage overlay networks or provisioning resources [47].

At the same time, Docker containers are treated as "black-boxes" by existing container orchestrators, as the minimal entity that can be orchestrated is the container itself [31]. Our objective is to go a step forward, by enabling a component-aware orchestration of containerised applications, i.e., we aim at allowing to orchestrate the management of components running within the same container. To this end, we introduced *Unit*s (i.e., suitably configured Supervisor instances — Sect. 6.4) in each container running some component, so as to offer an XML-RPC API allowing to remotely invoke the operations for managing the lifecycle of the components it hosts. The next step is hence to find a suitable implementation of the *Manager* in our orchestration approach (Fig. 6.3), i.e., to introduce a containerised component in an application, which allows coordinating the management of each of its components by suitably interacting with the corresponding *Unit*.

Our baseline idea for doing so is illustrated in Fig. 6.4. Whenever the user wishes to execute a management operation on a component of her application, she invokes the *Manager* asking it

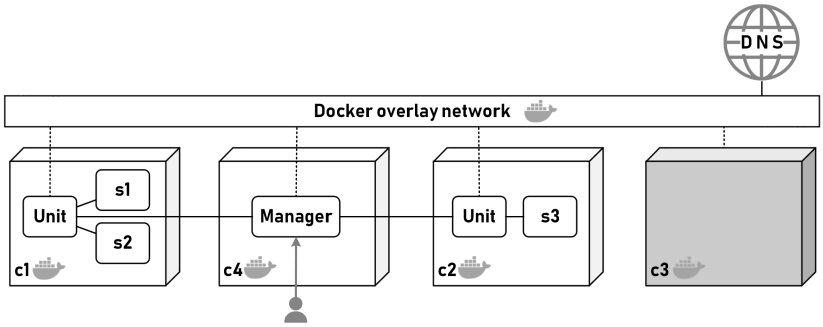


FIGURE 6.4. Interactions among *Manager* and *Unit*s for managing the lifecycle of containerised application components.

to execute such operation. The *Manager* then forwards the request to the *Unit* managing such component (i.e., it invokes the XML-RPC API of the Supervisor instance running within the container hosting the component). The interaction between the *Manager* and the *Unit* occurs through a Docker overlay network, and as soon as the *Unit* receives the request, it executes the required management operation (i.e., it runs the corresponding `program` section of the *Supervisor* configuration file). We hereafter illustrate a possible realisation of such an idea, given by the TOSKOSE MANAGER.

6.5.1 The Architecture of the TOSKOSE MANAGER

The TOSKOSE MANAGER realises the *Manager* in our orchestration approach (Fig. 6.3), hence being responsible of coordinating *Unit*s to allow remotely managing the containerised components of an application, based on its TOSCA specification. The TOSKOSE MANAGER is intended to be included in an application as an additional containerised component, as this will allow managing it with Docker-based container orchestrators, together with all other containers forming an application.

Given that we are piggybacking on top of existing container orchestrators, we can exploit their capabilities for setting up the overlay network where the containers of an application will run (both for single-host and multi-host settings). This also means that, by properly setting network aliases, containers can intercommunicate by simply exploiting their hostnames on the overlay network, which will be automatically resolved by the network DNS. In addition, Docker-based container orchestrators allow running different applications in different virtual private networks. This has two main advantages for our purposes, namely (i) it permits securing the interactions among the components of an application, including the TOSKOSE MANAGER, and (ii) even if multiple instances of TOSKOSE MANAGER run on the same overlay network, they will not interfere one another.

With the above setting, TOSKOSE MANAGER only requires to know which container is hosting which components (to be able to interact with the *Unit* running in the same container) and the

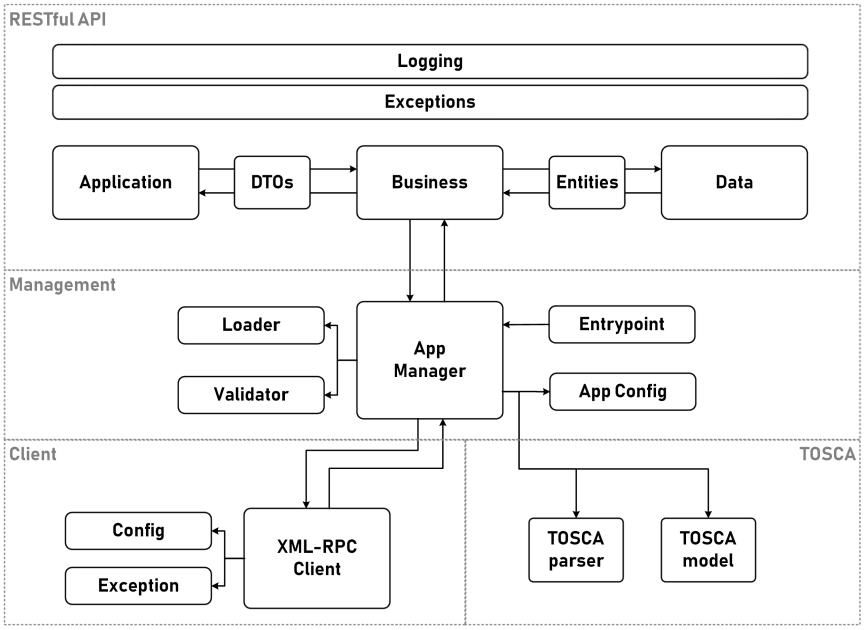


FIGURE 6.5. The architecture of the TOSKOSE MANAGER.

network aliases associated to the containers of an application. Such information is provided to TOSKOSE MANAGER by feeding it with the TOSCA application specification (from which it can retrieve the relations among components and containers) and an additional configuration file containing the network aliases associated to the container. The configuration file can be automatically generated, and both files are automatically injected to the TOSKOSE MANAGER by the TOSKOSE PACKAGER (Sect. 6.3).

To concretely implement the baseline idea shown in Fig. 6.4, the TOSKOSE MANAGER must hence feature (i) a remotely accessible API for allowing an application administrator to invoke the operation to manage the components of her application, (ii) a core processing module for identifying the *Unit*s where to forward requests, and (iii) a client for the XML-RPC API offered by the *Unit*, to concretely forward requests. It must also be capable of (iv) processing TOSCA application specifications. Fig. 6.5 shows the architecture of TOSKOSE MANAGER, designed to feature (i-iv), as well as to comply with the separation of concerns design principles to make it modular and extensible [106].

RESTful API. A *RESTful API* allows the application administrator to invoke the execution of a management operation on a component of her application. The API is designed by following the Web API design paradigm [2], by partitioning it in three main logical layers, i.e., *Application*, *Business* and *Data*. The *Application* layer contains the controllers for translating HTTP incoming requests and outgoing responses, and for encoding and decoding their payloads, by also validating them. The *Business* layer is where the business logic of the API resides, with business rules and workflows defined to suitably interact with the *Data* layer and with the core of TOSKOSE MANA-

GER. The *Data* layer provides a storage of all information needed to orchestrate an application (i.e., component and container names, network aliases, etc.).

To enhance data encapsulation in inter-layer communications, *DTOs* and *Entities* are defined. These are used in the communication between the *Application* and *Business* layers, and between the *Business* and *Data* layers, respectively. Two other standalone modules are used for logging and error handling of the API, i.e., *Logging* and *Exceptions*, which are organised as cross-cutting concerns.

Management. The *Management* area contains the core module of TOSKOSE MANAGER, i.e., *App Manager*. The latter acts as a proxy between the *RESTful API* and the *Client* modules. It indeed receives requests from the API (e.g., executing a management operation on a component, or getting the status of all components), and it suitably interacts with the *Client* modules so that they interact and coordinate *Unit*s to carry out the requests.

The *Management* is also responsible of configuring the environment for allowing *App Manager* to run. It indeed contains the *App Config* and *Entrypoint* modules, which configure the runtime environment and run a web server used to host the *App Manager* (with the web server also being the main process run in the container of the TOSKOSE MANAGER). It also contains the the *Loader* and *Validator* modules, used for loading and validating the TOSCA application specification and the Toskose configuration file, which contain the information needed to orchestrate the specified application.

Client. The *Client* is responsible of communicating with the *Unit*s on the containers hosting the components of an application. It hence contains a *XML-RPC Client* allowing to invoke the XML-RPC API offered by the *Supervisor* instances implementing the *Unit*s. Whenever the *XML-RPC Client* is required by the *App Manager* to invoke an operation offered by a *Unit*, it builds and sends a HTTP request to the API of such *Unit*, which payload is structured according to the XML structure expected by the API of *Supervisor*. The *XML-RPC Client* is returned XML data representing the outcome of its request from the *Unit*, and it communicates the *App Manager* such an outcome.

For enforcing fine-grained failure management, and in accordance to separation of concerns design principles, error handling is kept separate from the rest of the application [106]. Errors are indeed handled by the *Exceptions* module, which is also part of the *Client* area.

TOSCA. The *TOSCA* area is responsible of the processing of TOSCA application specifications. It indeed features two modules, i.e., *TOSCA parser* and *TOSCA modelling*, intended to allow parsing TOSCA application specifications and to build an in-memory representation of specified applications.

6.5.2 A Prototype Implementation of the TOSKOSE MANAGER

An open-source prototype implementation of the TOSKOSE MANAGER is publicly available on GitHub (`https://github.com/di-unipi-socc/toskose-manager`), and it is also shipped as a

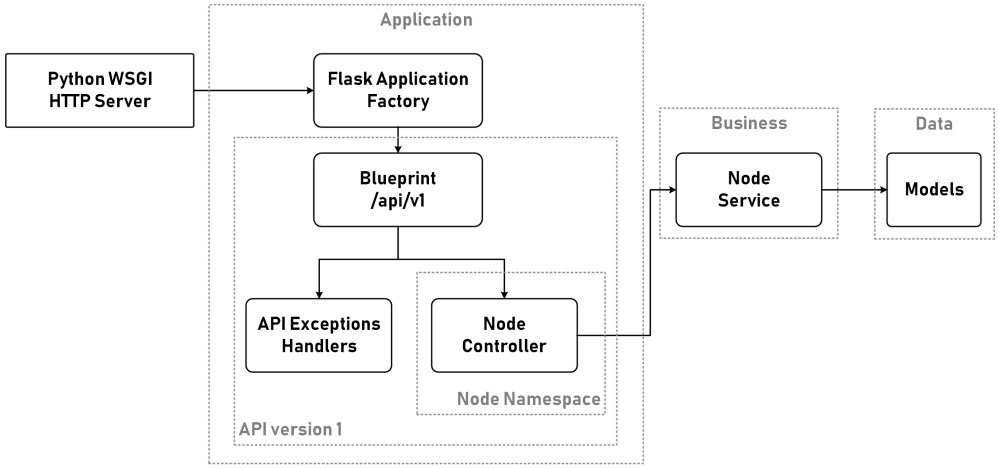


FIGURE 6.6. Architecture of the prototype implementation of the *RESTful API* of TOSKOSE MANAGER.

Docker image publicly available on the Docker Hub (`https://hub.docker.com/r/diunipisocc/toskose-manager`). The prototype of TOSKOSE MANAGER is written in Python (v3.7.1), and we hereafter detail its implementation. More precisely, given that the *TOSCA* modules have been obtained by suitably extending the OpenStack TOSCA parser (`https://github.com/openstack/tosca-parser`), we shall focus on the implementation of the *RESTful API* and of the *Management* and *Client* areas.

RESTful API. Fig. 6.6 illustrates the architecture of the *RESTful API* featured by the prototype implementation of TOSKOSE MANAGER. The topmost component is a *Python WSGI HTTP Server*, where WSGI stands for "Web Server Gateway Interface" (which is a specification describing how a web server communicates with the web applications it hosts, and how they can be chained together to process a request). The HTTP Server is powered by Gunicorn (`https://gunicorn.org`), it implements the WSGI interface, and it permits running the web application implementing the API to be offered.

The *Application* layer of the API is then implemented by exploiting the Flask Python framework (`https://palletsprojects.com/p/flask`), extended with Flask-RESTPlus (`https://flask-restplus.readthedocs.io`). The latter has been included as it provides a collection of Python decorators and tools for quickly building RESTful APIs and exposing their documentation using the Swagger UI (`https://swagger.io`). In addition, Flask-RESTPluse enforces modularity of built APIs, hence making the current implementation of the *Application* layer of the *RESTful API* extensible for further developments.

The above setting has been obtained by suitably configuring the *Flask Application Factory*, which acts as the entrypoint of an application, by initialising the Flask environment where the application will run. This includes mounting extensions (such as Flask-RESTplus), as well as registering blueprints (i.e., logical groups partitioning the modules of an application based

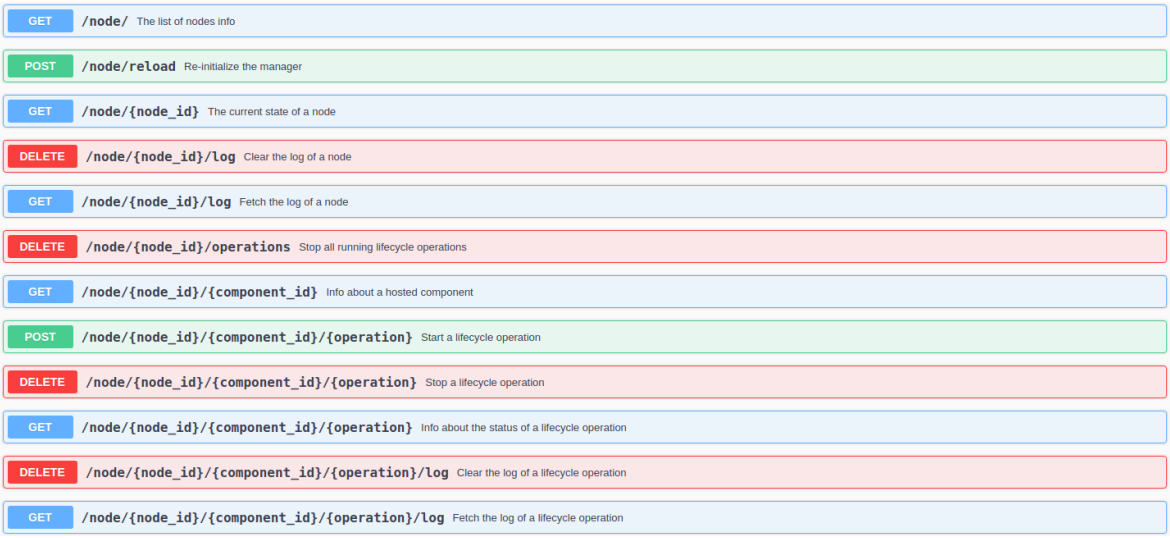


FIGURE 6.7. Snapshot of the HTTP methods offered by a running instance of the *RESTful API* of TOSKOSE MANAGER, obtained from the Swagger UI featured by Flask RESTPlus.

on the concerns they relate to, to enforce separation of concerns [106]). For instructing the *Flask Application Factory* to loading the current prototype of the API, we hence developed a *Blueprint* */api/v1*. Notice that adding a different version of the API, or running different versions simultaneously, simply require to change or add another blueprint among those registered.

In addition, by exploiting the Flask RESTPlus extension, we logically organised the Blueprint using so-called "namespaces". A *Node Namespace* is indeed used, which is exploited to mark the *Node Controller* related to the resource `/node`. The latter is the root resource of *RESTful API*, and the *Node Controller* has been implemented so as to offer all methods shown in Fig. 6.7 The *Node Controller* is hence responsible of accepting incoming requests, which it decodes and validates by checking their payload against an expected schema. Non-valid requests are refused, while valid requests are passed to the business layer for processing. Such a forwarding involves exchanging complex structured data, which is done by exploiting DTOs for enforcing data encapsulation.

The *Business* layer is implemented by the *Node Service*, which business rules allow to process incoming requests. Intuitively, it processes each request by retrieving information on involved application components, aggregating such data in the form of DTOs, and passing the request and retrieved data to the core of TOSKOSE MANAGER. In addition, the *Node Service* fetches and manipulates the logs of the *RESTful API*.

Management. The *App Manager* module in the *Management* area (Fig. 6.5) is the core module for managing multi-component application with TOSKOSE. Such a module maintains an in-memory representation of the managed application built from its TOSCA specification and enriched according to the Toskose configuration file, both loaded and validated during the initialization of

the module. The TOSCA parsing is done by exploiting another Python module (i.e., *TOSCAParser*), which implementation is essentially wrapping the OpenStack TOSCA Parser library. Logging and error handling are also configured during the initialization of *App Manager*, by exploting the standalone *Loader* module.

After the initialization, the *App Manager* starts waiting for incoming requests for the (*Node Service* of the) *RESTful API*. Upon the receiving of a request, it instructs the *XML-RPC Client* to contact the *Unit* managing the component involved by the request.

Client. The *Client* area is implemented following the Factory method pattern, with the *App Manager* delegating to the *Client Factory* the decision to determine the concrete implementation of the client that must be used to interact with a *Unit*. Currently, the only available implementation is that for the XML-RPC API of *Supervisor*, which is obtained by exploiting the Python built-in `xmlrpc.client`. We anyway decided to implement clients using the Factory method pattern to allow TOSKOSE MANAGER to be extended to support multiple interaction protocols, in view of further developments.

6.6 Generating deployable artifacts

The last brick needed for enabling our orchestration approach is the *Packager*, i.e., a solution that, given the TOSCA specification of an application, automatically generates the deployable artifacts needed to actually enact its deployment on top of existing, production-ready container orchestrators. The deployable artifacts must be such that *Unit*s (i.e., suitably configured Supervisor instances) are included within the containers hosting application components, and that a containerised TOSKOSE MANAGER is added to the application. The *Unit*s and the TOSKOSE MANAGER are indeed needed to enable a component-aware orchestration on top of the targeted container orchestrator. We hereafter present our solution for doing so, by first illustrating a solution for automating the generation of deployable artifacts, and by then presenting its prototype implementation, i.e.,TOSKOSE PACKAGER.

6.6.1 Automating the Generation of Deployable Artifacts

Fig. 6.8 illustrates a workflow that, given the TOSCA specification of an application, automatically generates the deployable artifacts for enabling a component-aware orchestration of multi-component applications on top of an existing container orchestrator. The workflow begins with the activities *TOSCA validation* and *Model generation*, which validate the TOSCA application specification and generate an in-memory representation of the specified application, respectively. If the given TOSCA specification is not valid, or if the *Model generation* fails, the workflow exits by returning error information.

The workflow then proceeds by dealing with the *Configuration management*, i.e., with the file indicating the configuration of the containers forming the application, including the network

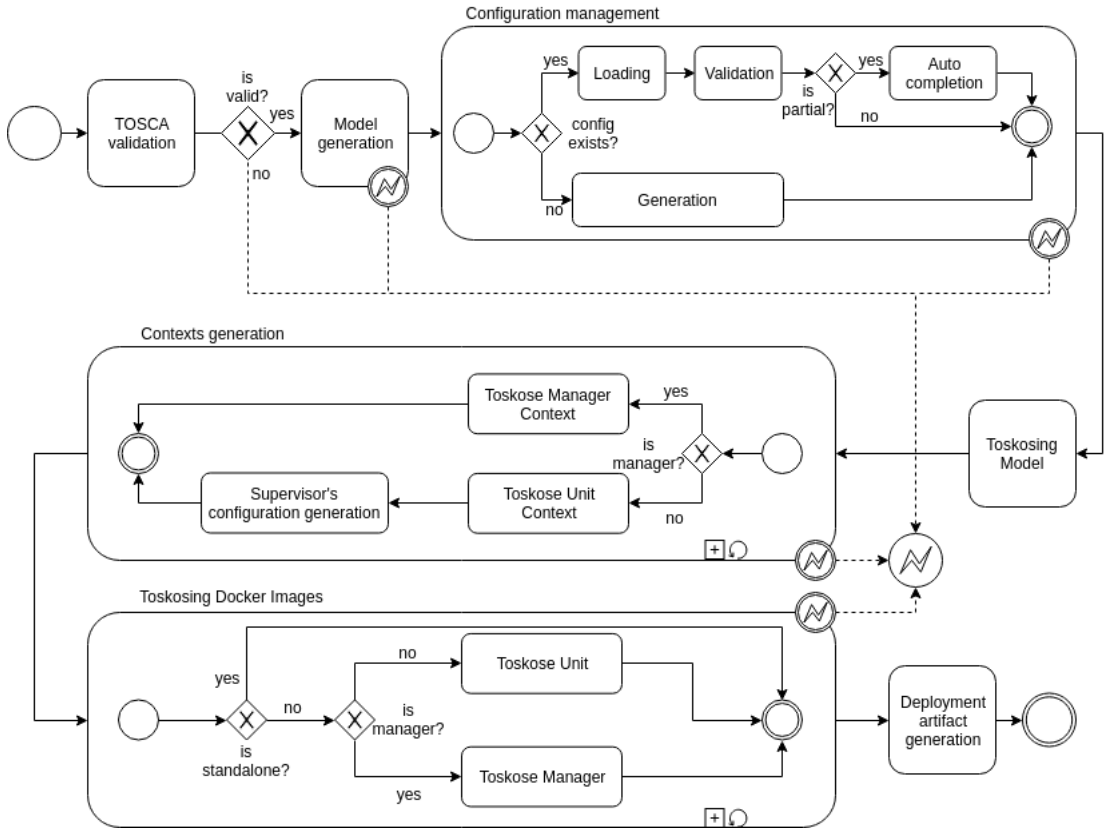


FIGURE 6.8. Workflow for automatically generating deployable artifacts, depicted according to the BPMN graphical notation [1].

aliases and ports for reaching the components they host. If such a configuration file is missing or partial, the workflow automatically sets the missing fields to default values. The obtained configuration file is then used by the *Toskosing model* activity, which enriches the in-memory application representation with the information contained in the configuration file, as well as by adding the additional container packaging the TOSKOSE MANAGER.

The subsequent phase (*Contexts generation*) is repeated for each container of the application hosting some component (either being some original application component or the newly introduced TOSKOSE MANAGER). For each of such container, a Docker build context is prepared, by setting up a folder containing all files needed to build a Docker image. If a container is hosting application components, a so-called *Toskose Unit Context* is generated. The latter contains all the artifacts needed for deploying and managing the application components hosted on the container, i.e., the artifacts implementing an application component and those implementing its management operations. An automatically generated *Supervisor* configuration file is also included within the build context, which will then be used to configure the *Supervisor* instance implementing the *Unit* on the container (e.g., for ensuring that its XML-RPC API will offer methods for remotely invoking the management operations of the hosted components). If the

processed container is instead that hosting the TOSKOSE MANAGER, the build context only contains the TOSCA specification and the Toskose configuration file of the application under processing.

The workflow then proceeds by *Toskosing Docker images*, i.e., preparing the Docker images of each containers of the application. If the container is a standalone container, the original image is kept. If instead the container is hosting some application component, two different activities are enacted, depending on whether the hosted components are original application components or the TOSKOSE MANAGER. In the former case, a new Docker image is built by combining the corresponding build context and the *Supervisor* bundle fetched from the TOSKOSE UNIT Docker image, by means of a multi-stage Dockerfile. In the latter case, the build context is instead combined with the Docker image packaging the TOSKOSE MANAGER, still by means of a multi-stage Dockerfile.

The deployable artifact generation process then completes with the *Deployment artifact generation* activity, which essentially takes the newly generated images of Docker containers (hereafter called *toskosed* images, for brevity) and combines them in a multi-container application deployment, e.g., a Docker Compose file. The *toskosed* images are configured in accordance with the in-memory application representation, so as to allow them to properly intercommunicate (both for running the application business and for allowing the TOSKOSE MANAGER to interact with the *Supevisor* instances implementing the *Unit*s).

6.6.2 The Architecture of the TOSKOSE PACKAGER

Fig. 6.9 illustrates the architecture of TOSKOSE PACKAGER, our solution for generating the artifacts enabling the component-aware orchestration of an application on existing Docker-based container orchestrators, based on the workflow in Fig. 6.8. Still pursuing the aim of obtaining a modular and extensible solution, the architecture is designed in accordance with the separation of concerns design principles [106]. The architecture is indeed partitioned by devoting different sets of modules to different stages of the workflow.

A *CLI* module allows users to provide the TOSKOSE PACKAGER with the necessary input for starting the process for automating the generation of deployable artifacts, i.e., a TOSCA application specification and (optionally) a Toskose configuration file. The input is then passed to the *Main module*, which is in charge of coordinating the modules forming the architecture of TOSKOSE PACKAGER to suitably execute the activities of the workflow. It then first processes the input, by invoking the *Loader* module for actually importing the input files, and the modules in the *TOSCA* area for validating the TOSCA application specification and generating an in-memory representation the application.

The *Main Module* then interacts with the modules in the *Configuration management* area. The *Validator* module allows the *Main module* to validate the input Toskose configuration file, if any. The *Completer* module instead allows the *Main module* to complete the Toskose

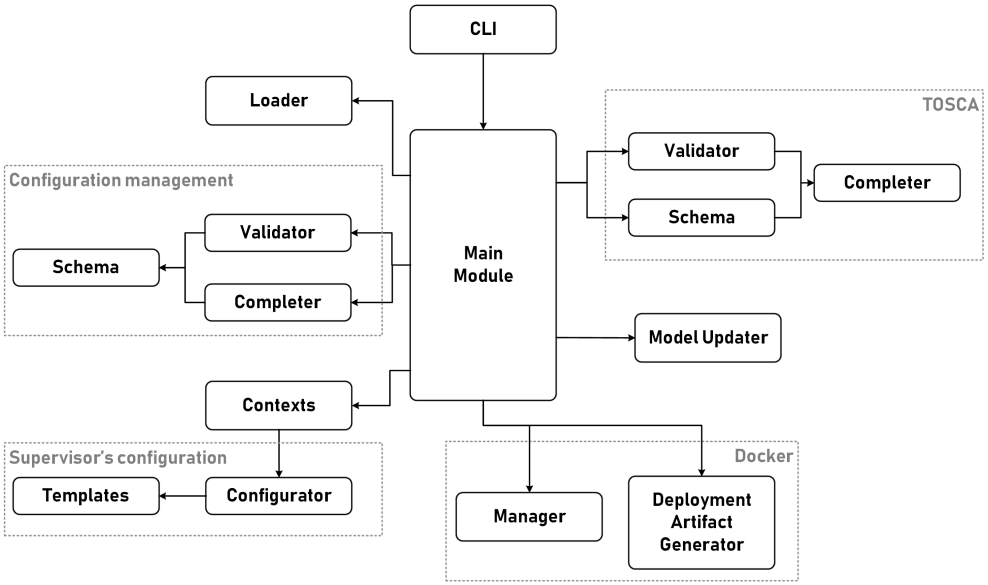


FIGURE 6.9. The architecture of the TOSKOSE PACKAGER.

configuration file with default configurations, or to generate it from scratch if no configuration file is provided. Both the *Validator* and the *Completer* relies on a *Schema* module, indicating how the Toskose configuration file has to be structured, and which default values to employ for missing configurations. Once the configuration is ready, the *Model Updater* is used by the *Main Module* to enrich the in-memory application representation with the configuration information.

The *Main Module* continues by interacting with the *Contexts* module, which creates the build contexts for each container hosting some component, including that hosting the TOSKOSE MANAGER. For the containers hosting original application components, *Contexts* interacts with the *Configurator*, which allows to create the file for configuring the *Supervisor* instances implementing the *Unit*s, based on existing *Templates*.

Finally, the *Main Module* interacts with the modules in the *Docker* area. It first requires to the *Manager* to build the Docker images for the containers forming the application, which the *Manager* carries out by relying on the build features of the Docker Engine. The *Main Module* then requires the generation of the final *Deployment artifact* to the *Deployment Artifact Generator*.

6.6.3 A Prototype Implementation of the TOSKOSE PACKAGER

An open-source prototype implementation of the TOSKOSE PACKAGER is publicly available on GitHub (`https://github.com/di-unipi-socc/toskose-packager`). The prototype is written in Python (v3.6) and it has been released on the Python Package Index (`https://pypi.org/project/toskose`), to allow installing it with the command `pip install toskose`.

The prototype provides a command-line interface that takes as input a Cloud Service ARchive and (optionally) a Toskose configuration file, and it returns a Docker Compose artifact. The latter

allows to deploy the application on Docker-based container orchestrators, with the latter being used to orchestrate the components of the application, while the orchestration of the components hosted on the containers being enabled by the RESTful API of TOSKOSE MANAGER (Sect. 6.5). Currently, the generated Docker Compose artifact is tested and fully working on Docker Swarm and on Kubernetes, with the latter requiring to first run Kompose (`https://kompose.io`) or Compose Object (`https://github.com/docker/compose-on-kubernetes`) to actually enact the deployment of the application.

We hereafter detail our prototype implementation of the TOSKOSE PACKAGER, by showing how each component of the architecture in Fig. 6.9 has been implemented.

CLI. The command-line interface (*CLI*) offers the following interface:

```
toskose [OPTIONS] CSAR_PATH [CONFIG_PATH]
```

where the optional argument `OPTIONS` is a list of options for customising the run of `toskose`. In particular, `-o` and `--output-path PATH` allow to specify the path where to place the output deployment artifacts, `-p` and `--enable-push` activate the automatic pushing of *toskosed* images on a Docker registry, `--docker-url URL` allows to define a custom entrypoint for the Docker Engine API, `-q` and `--quiet` reduce the output information messages, and `--debug` activates the debug mode.

The other arguments instead indicate the paths to the input files for the TOSKOSE PACKAGER. More precisely, `CSAR_PATH` indicates the path to a Cloud Service ARchive (CSAR), containing the TOSCA specification of an application and the artifacts realising the application and its management operations. The optional argument `CONFIG_PATH` instead provides the path to a Toskose configuration file.

Main Module. The *Main Module* is implemented as a Python module that is invoked by the command-line interface, which passes it the paths to the input files to be processed, and the processing options. It then starts coordinating the other modules of our prototype implementation of TOSKOSE PACKAGER to carry out the activities of the workflow in Fig. 6.8, in the given order.

The *Main Module* also generates to temporary directories using the Python `tempfile` library. A directory is used for storing the content of the (unpacked) CSAR archive, while the other one is used for storing the Docker build contexts. Both the directories are stored under `/tmp` and they are removed once the *Main Module* reaches the end of the workflow.

TOSCA. The implementation of the *Validator* provides the necessary for checking whether the input CSAR archive complies with the TOSCA standard [124]. It indeed allows to check whether the extension of the archive complies with admitted ones (i.e., `.csar` or `.zip`), as well as the directories composing the archive are organised as indicated by the TOSCA standard.

The *Parser* and *TOSCA Model* instead allow parsing the YAML file defining the TOSCA specification of an application and building an in-memory representation of the application. They are implemented by extending the OpenStack TOSCA parser (`https://github.com/`

`openstack/tosca-parser`), in order to allow processing the TOSCA-based representation given by TosKer [31].

Configuration. Toskose configuration files are specified in YAML, and they are structured in two YAML objects, i.e., `nodes` and `manager`. The object `nodes` is devoted to the configuration of the containers hosting application components, with one nested YAML object for each of such containers. Each nested object is given the same name as that of the container in the TOSCA application specification, and it allows to specify the `alias` associated to the container and the `port` where the XML-RPC API of the *Supervisor* instance implementing a *Unit* is offered. These are then used by the used by the TOSKOSE MANAGER for communicating with the *Unit* managing the components running in the container.

The object `manager` instead allows to provide the configuration information for the Docker container running the TOSKOSE MANAGER. It indeed allows specifying its `alias` on the Docker network, as well as the `port`, `username` and `password` for accessing the *RESTful API* of the TOSKOSE MANAGER

A Toskose configuration file can be optionally provided to our prototype implementation of the TOSKOSE PACKAGER. If provided, it is validated against the above illustrated schema by the implementation of the *Validator*. If something is missing, or if no configuration file is given, an auto-completion routine implementing the *Completer* allows to fill it with default values.

Model Updater. The implementation of the *Model Updater* adds the information contained in the Toskose configuration file to the in-memory representation of the processed application. More precisely, it first extends the application representation by setting environment variables to be defined in each container hosting a component, in such a way that the instance of *Supervisor* it runs is configured to listen on the indicated `port`. It also extends the representation of each container by setting properties needed for naming and tagging the correspondingly generated *toskosed* image (and optionally pushing it to a Docker registry).

The implementation of the *Model Updater* also includes an additional container to the in-memory application representation, devoted to hosting the TOSKOSE MANAGER. Such a container is then configured according to the specified configuration information, in a way similar to that described above. The choice of injecting the addition container during this activity simplifies the subsequent steps, which have to process only the in-memory application representation instead of fetching information from different sources.

Contexts generation. The *Contexts* module of our prototype implementation of the TOSKOSE PACKAGER fills a temporary folder devoted to Docker build contexts with a *Toskose Manager Context* for the container hosting the TOSKOSE MANAGER and a *Toskose Unit Context* for each container hosting some original application component. The *Toskose Manager Context* is devoted to storing the Toskose configuration file and the TOSCA specification of the application under processing, which are needed by the TOSKOSE MANAGER for enacting the management of the application.

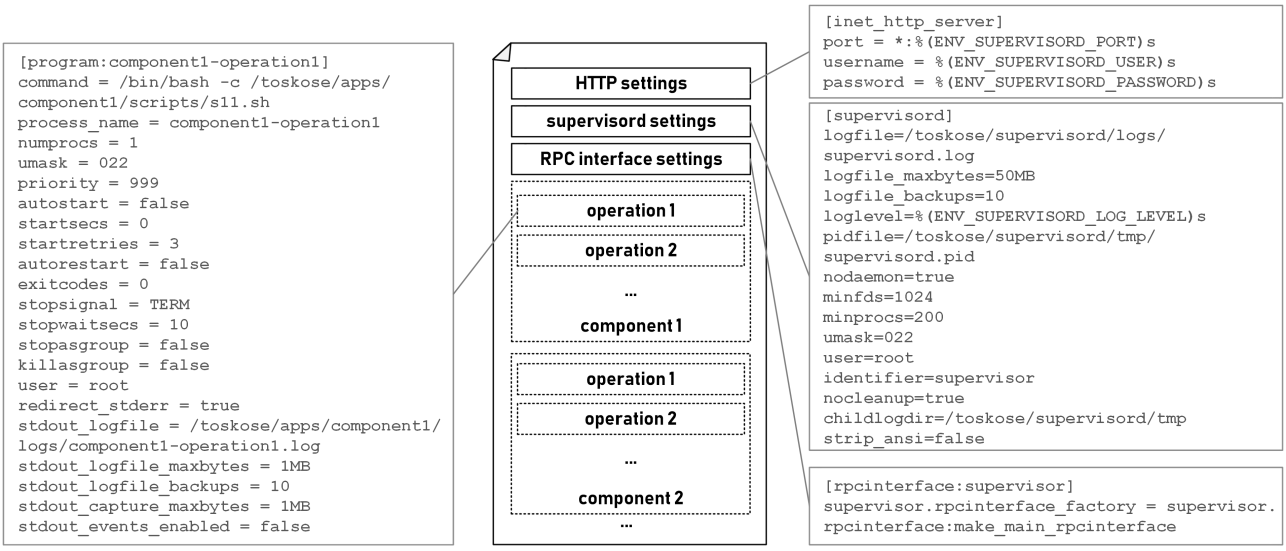


FIGURE 6.10. Template of the configuration of *Supervisor* injected in Docker containers to implement *Unit*s.

Each *Toskose Unit Context* is assigned the name of the corresponding container. It is devoted to contain the artifacts realising the components hosted on the container, as well as the scripts implementing their management operations. The folder also contains a file `supervisor.conf`, providing all configuration information needed by the *Supervisor* instance running on the container for suitably implementing a *Unit*. The file `supervisor.conf` is generated by automatically extending a base template, which schema is in Fig. 6.10. The *HTTP settings* section configures the HTTP server used for offering the XML-RPC API of *Supervisor* by expanding the enviroment variables defining its configuration in the in-memory application representation. The *supervisord settings* section configures *supervisord* (the main process of *Supervisor*) with some default configurations, except for the logging level, which is fetched from the information stored in the in-memory application representation. The *RPC interface settings* section initialises the XML-RPC API of the *Supervisor* instance to run. The file is completed by including `program` sections allowing to remotely invoke the management operations of the application component hosted on the container.

Docker. The *toskosed* Docker images are built by the *Manager* module. The latter is implemented by exploiting the `docker-py` Python library (`https://docker-py.readthedocs.io`), which allows to interact with the API of the Docker Engine installed on a host, also for building Docker images. For each container to be *toskosed* (including the TOSKOSE MANAGER), the *Manager* implementation proceeds as follows. Firstly, it pulls the `toskose-unit` or `toskose-manager` Docker image from the Docker Hub and the Docker image associated with the container in the in-memory application from the specified registry, if they are not already available locally. It then

instructs the Docker Engine to build the *toskosed* image, by passing it the application context, the pulled images and a multi-stage Dockerfile. If explicitly required by the user, the *Manager* module also proceeds with pushing *toskosed* images to a Docker registry.

The implementation of the *Deployment Artifact Generator* completes the workflow, by generating the Docker Compose file from the in-memory application representation. The containers of the application (including standalone containers) are added to the Docker compose file as `services` and configured as specified (e.g., by setting their network aliases and environment variables as indicated in the in-memory application representation). The automatically generated *toskosed* images are then used to implement the services corresponding to containers hosting some application component, while the Docker images originally indicated in the TOSCA application specification are used to implement those corresponding to standalone containers. In addition, Docker volumes are included in accordance to the in-memory application representation, and a Docker overlay network is set for allowing the deployment of the application in both single-host and multi-host infrastructure. The resulting Docker Compose file follows the schema shown in Fig. 6.11.

6.7 Case studies

We hereby illustrate two case study based on two different multi-component applications, both designed for testing application orchestrators in practice. We first consider the *Thinking* application (`https://github.com/di-unipi-socc/thinking`), which we developed in the scope of our previous research [22], and we then consider *Sock Shop* (`https://microservices-demo.github.io`), a third-party application developed and maintained by Weaveworks and Container solutions. We show how TOSKOSE enables a component-aware orchestration of such applications on Docker Swarm and Kubernetes, in a multi-host setting.

6.7.1 The *Thinking* Case Study

Thinking is an open-source web application allowing its users to share thoughts on a web-based portal, so that other users can read them. *Thinking* is composed by three main components:

- A MongoDB database storing the collection of thoughts shared by users. The database is obtained by directly instantiating a MongoDB container, which needs to be attached to a volume where shared thoughts are persistently stored.
- A Java-based RESTful Web API to remotely access the database of shared thoughts. The API is hosted on a Maven container, and it requires to be connected to the MongoDB container (for remotely accessing the database of shared thoughts).
- A web-based GUI visualising all shared thoughts and allowing to insert new thoughts into the database. The GUI is hosted on a NodeJS container, and it depends on the availability

```
---
version: '3.7'

networks:
  toskose-network:                      # DOCKER NETWORK DEVOTED TO THE APPLICATION
    driver: "overlay"                   # setting the network to be an overlay network
    attachable: true

services:

  toskose-manager:                      # CONTAINER HOSTING THE TOSKOSE MANAGER
    image: <toskosed_image>             # setting starting image to corresponding toskosed image
    networks:
      toskose-network:                  # attaching the container to the overlay network
        aliases:
        - <alias>                       # setting the alias of the container on the overlay network
    environment:
    - TOSKOSE_MANAGER_PORT=<port>       # setting env. vars. needed by the Toskose Manager
    - TOSKOSE_APP_MODE=<mode>
    - SECRET_KEY=<secret_key>
    ports:
    - <api_port_mapping>                # setting port mapping to expose the RESTful API to users

  <container_node_name>:                # CONTAINER HOSTING SOME APPLICATION COMPONENT
    image: <toskosed_image>             # setting starting image to corresponding toskosed image
    init: true                          # enabling Tini as init process
    networks:
      toskose-network:                  # attaching the container to the overlay network
        aliases:
        - <alias>                       # setting the alias of the container on the overlay network
    volumes:
    - <volume_mapping>                  # attaching container to specified volumes (if any)
    environment:
    - SUPERVISORD_ALIAS=<alias>         # setting env. vars. for the Unit (i.e., Supervisor instance)
    - SUPERVISORD_PORT=<http_port>
    - SUPERVISORD_USER=<user>
    - SUPERVISORD_PASSWORD=<password>
    - SUPERVISORD_LOG_LEVEL=<log_level>
    - ...                               # setting other env. vars. needed by the hosted components
    ports:
    - ...                               # setting specified port mappings (if any)

  <standalone_container_node_name>      # STANDALONE CONTAINER
    image: <base_image>                 # setting starting image to that indicated in the TOSCA spec
    networks:
      toskose-network:                  # attaching the container to the overlay network
        aliases:
        - <alias>                       # setting the alias of the container on the overlay network
    volumes:
    - <volume_mapping>                  # attaching container to specified volumes (if any)

volumes:                                # DOCKER VOLUMES (if any)
  ...
```

FIGURE 6.11. Schema of the Docker Compose file generated by our prototype implementation of the TOSKOSE PACKAGER.

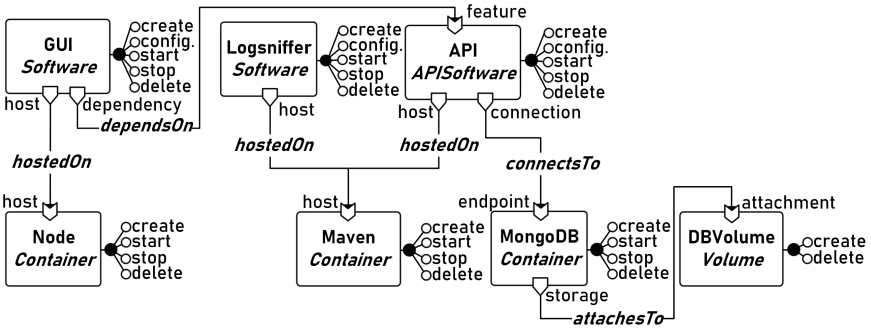


FIGURE 6.12. TOSCA-based representation of *Thinking*, obtained by exploiting the TOSCA types defined by TosKer [31].

of the API to properly work (as it sends HTTP requests to the API to retrieve/add shared thoughts).

For the purposes of this case study, and following the sidecar pattern, the application also includes a Logsniffer in the Maven container running the API of *Thinking*, which provides a web-based GUI for visualising and filtering the logs of the API.

The GUI, API and Logsniffer are also provided with a set of shell scripts implementing their lifecycle operations. The operation to install, configure, start, stop and uninstall each of such components are indeed implemented by the scripts `install.sh`, `configure.sh`, `start.sh`, `stop.sh` and `uninstall.sh`, respectively. The API is also equipped with the script `push_default.sh`, which can be optionally executed when the API is configured (but not running) to add a default set of thoughts to the MongoDB database.

6.7.1.1 Modelling *Thinking* with TOSCA

By exploiting the TOSCA-based representation given by TosKer [31] (and recapped in Sect. 6.2.2 for making this article self-contained), the *Thinking* application can be modelled in TOSCA as shown in Fig. 6.12. *MongoDB* is modelled as a component of type *tosker.nodes.Container* and it is attached to the needed volume (*MongoDB*) through a relationship of type *tosca.relationships.AttachesTo*. *API* and *Logsniffer* are modelled as software components hosted on a component of type *tosker.nodes.Container* (i.e., *Maven*), with *API* being also connected to *MongoDB*. Notice that, while *Logsniffer* is of type *tosker.nodes.Software*, *API* is of type *tosker.nodes.APISoftware*, which extends *tosker.nodes.Software* to include the *push_default* operation featured by *API*. Finally, *GUI* is modelled as a component of type *tosker.nodes.Software*, which is hosted on a component of type *tosker.nodes.Container* (i.e., *Node*). The *GUI* is also indicated as depending on the availability of *API* to suitably serve its clients.

The corresponding TOSCA application specification is publicly available on GitHub, in the CSAR packaging such specification together with the artifacts implementing the components of

Thinking and their management operations (`https://github.com/di-unipi-socc/toskose-packager/blob/master/tests/data/thinking-v2/thinking-v2.csar`). Concerning artifacts, it is worth noting that the artifact type used to implement *MongoDB* differ from those of *Maven* and *Node*, as *MongoDB* is qualified to be a standalone container, while *Maven* and *Node* are used as system containers for hosting application components. In addition, no artifact is provided for implementing the lifecycle operations of the containers and volume in *Thinking*, as our aim is to piggyback on existing container orchestrators, which natively feature such operations.

6.7.1.2 Generating a Deployable Artifact for *Thinking*

We generated a Docker Compose file enabling a component-aware orchestration of the management of *Thinking* on Docker-based container orchestrators by running the TOSKOSE PACKAGER as follows:

```
$ toskose -p thinking.csar toskose.yml
```

where `thinking.csar` was a local copy of the CSAR packaging *Thinking* available on GitHub (see Sect. 6.7.1.1), and `toskose.yml` was the Toskose configuration file shown in Fig. 6.13(a). Fig. 6.13(b) instead shows the Toskose configuration file automatically generated by the TOSKOSE PACKAGER, which we obtained by not specifying any Toskose configuration file while running the TOSKOSE PACKAGER

```
$ toskose -p thinking.csar
```

In both cases, `-p` was set to instruct the TOSKOSE PACKAGER to automatically push *toskosed* images to the Docker Hub.

Both runs of the TOSKOSE PACKAGER successfully generated a Docker Compose file for deploying *Thinking* on a Docker-based container orchestrator. Fig. 6.14 shows the Docker Compose file obtained by running the TOSKOSE PACKAGER with the Toskose configuration file in Fig. 6.13(a). The file shows that the TOSKOSE MANAGER is automatically included among the containers to be deployed and how each container hosting some application component is implemented by a *toskosed* image, with the latter being suitably configured to run a *Supervisor* instance as a *Unit* managing the hosted components.

The Docker compose file in Fig. 6.14 is considered hereafter, as the `docker-compose.yml` file in the rest of our case study. For the sake of completeness, it is worth highlighting that all activities shown hereafter were successfully executed also with the Docker Compose file obtained by running the TOSKOSE PACKAGER with the Toskose configuration file in Fig. 6.13(b).

```
nodes:
  maven:
    alias: maven
    port: 9456
    user: user_21ty5
    password: 1t5mYp4ss
    log_level: INFO
    docker:
      name: giulen/thinking-maven-toskosed
      tag: 0.1.3
  node:
    alias: node
    port: 13450
    user: user_a4bc2
    password: p4ssw0rd
    log_level: DEBUG
    docker:
      name: giulen/thinking-node-toskosed
      tag: 2.1.5
manager:
  alias: toskose-manager
  port: 12000
  user: admin_manager
  password: password_manager
  mode: production
  secret_key: my_secret
  docker:
    name: giulen/thinking-manager-toskosed
    tag: latest
```

(a)

```
nodes:
  maven:
    alias: maven
    port: 9001
    user: admin
    password: admin
    log_level: INFO
    docker:
      name: giulen/thinking-maven-toskosed
      tag: 0.1.3
      registry_password:
  node:
    alias: node
    port: 9001
    user: admin
    password: admin
    log_level: INFO
    docker:
      name: giulen/thinking-node-toskosed
      tag: 2.1.5
      registry_password:
manager:
  alias: toskose-manager
  port: 10000
  user: admin
  password: admin
  mode: production
  secret_key: secret
  docker:
    name: giulen/thinking-manager-toskosed
    tag: latest
    registry_password:
```

(b)

FIGURE 6.13. Toskose configuration files used for generating a deployment of *Thinking*, with (a) being manually created and (b) being automatically generated by the TOSKOSE PACKAGER.

6.7.1.3 Deploying and Managing *Thinking* with Docker Swarm

To deploy the obtained Docker Compose file (i.e., `docker-compose.yml`) with Docker Swarm, we first exploited the Docker Machine tool (`https://docs.docker.com/machine`) to create Swarm cluster composed by four virtual machines (Fig. 6.15).

Following the Docker documentation [5], we then exploited the Docker Stack abstraction for actually enacting the deployment of *Thinking* on the Swarm cluster.

```
$ docker stack deploy --compose-file docker-compose.yml \\
--orchestrator swarm thinking-stack
```

Fig. 6.16 shows the execution and outcomes of the above command. Docker Swarm cared about spawning the containers in the *Thinking* application, and of distributing them on the Swarm cluster. Such containers were however only running their main processes, i.e., the TOSKOSE MANAGER in the case of `thinking-stack_toskose-manager`, the `mongo` application in

the case of the standalone container `thinking-stack_mongodb`, and the *Supervisor* instances implementing the *Unit*s for the remaining two containers. The *GUI*, *API* and *Logsniffer* were instead not deployed yet, as the actual management of their lifecycle was to be orchestrated through the TOSKOSE MANAGER.

We hence completed the deployment of the *Thinking* application by exploiting the cURL command-line tool (`https://curl.haxx.se`) for interacting with the *RESTful API* offered by the TOSKOSE MANAGER. The latter was running on the virtual machine with IP address `192.168.99.115` (Figs. 6.15 and 6.16), and the API was configured to listen on port `12000` (Fig. 6.14). We hence installed the *API* by executing the following command:

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/create"
```

Once installed, we configured the *API* and instructed it to populate the database with default thoughts by issuing:

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/configure"
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/push_default"
```

We then started the *API* by executing

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/start"
```

Similarly, we installed and started *Logsniffer* by executing

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/logsniffer/create"
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/logsniffer/start"
```

and we installed, configured and started the *GUI* by executing

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/node/gui/create"
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/node/gui/configure"
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/node/gui/start"
```

As a result, we were able to reach the web-based portal of *Thinking* both for visualising shared thoughts and for sharing new thoughts (Fig. 6.17).

It is worth highlighting how we came to complete the deployment of *Thinking*. We first fully relied on the capabilities of Docker Swarm to spawn and manage the Docker containers in *Thinking*, and the Docker volume needed by *MongoDB*. The TOSKOSE MANAGER then allowed us to manage the rest of the application at component-level, as we were able to remotely invoke the operations managing the lifecycle of the software components in *Thinking*.

To further experiment the component-aware orchestration enabled by our approach with the deployed instance of *Thinking*, we wished to stop and restart its *API*, by also observing the changes actually happening to the application. We hence stopped the *API* of *Thinking* by remotely invoking its *stop* management operation through the TOSKOSE MANAGER:

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/stop"
```

As a result, if connecting to the web-portal shown by *Thinking*, none of the shared thoughts was displayed. Fig. 6.18(a) shows the reason for this, with the console of the browser notifying the failure of the GET request sent to the *API* for retrieving shared thoughts. Such an error is due to the fact that the *API* was successfully stopped, as indicated by the logs visualised by the *Logsniffer* (shown at the bottom of the same figure).

To proceed with our experiment, we restarted the *API*, i.e., we remotely invoked its management operation *start* through the RESTful API offered by TOSKOSE MANAGER.

```
$ curl -X POST -H "accept: application/json" \
       "http://192.168.99.115:12000/api/v1/node/maven/api/start"
```

Fig. 6.18(b) shows the outcomes of such an invocation, i.e., the web-portal returned to visualise the shared thoughts, and the logs visualised by *Logsniffer* showed that the *API* was successfully restarted and returned serving HTTP requests.

Even if simple, the above experiment further highlights how our approach enables a component-aware orchestration of the management of a multi-component application. We were indeed able to stop and restart a component (i.e., *API*), without requiring to stop the container running it or interfering with the other components running on the same container. *Logsniffer* continued to run during the whole experiment, allowing us to visualise the logs of the *API*. This also means that the container hosting *Logsniffer* and *API* continued to run, as expected (as its main process is the *Supervisor* instance implementing the *Unit* managing *API* and *Logsniffer*).

6.7.1.4 Deploying and Managing *Thinking* with Kubernetes

We run two different experiments for deploying the Docker Compose file obtained from the TOSKOSE PACKAGER (i.e., `docker-compose.yaml`) on Kubernetes, differing on the tool exploited

for doing so, i.e., Kompose (`https://kompose.io/`) and Compose Object (`https://github.com/docker/compose-on-kubernetes`). For the sake of conciseness, we hereafter only report on that based on Kompose.

After creating a Kubernetes cluster, we exploited Kompose to deploy the `docker-compose.yaml` file on the cluster. This was done by running

```
$ kompose up docker-compose.yml
```

which outcomes are shown in Fig. 6.19. We then exploited the Kubernetes client (i.e., `kubectl`) to specify that the TOSKOSE MANAGER acts as Ingress node for our deployment [88], hence allowing us to reach the RESTful API it offers.

```
$ kubectl patch svc toskose-manager -p '{"spec":{"type":"LoadBalancer"}}'
```

We issued analogous commands to allow remotely accessing *GUI* and *API*, and this completed the deployment and configuration of the containers in *Thinking*, fully carried out by exploiting existing capabilities featured by the Kubernetes environment.

We then repeated the activities for deploying the software components in *Thinking*, and for stopping and restarting its *API*. In other words, we repeated the same sequence of `curl` commands shown in Sect. 6.7.1.3, with the only difference given by the IP address where the TOSKOSE MANAGER was listening (`10.97.103.166` in this case, as shown in Fig. 6.19). All such commands successfully executed and resulted in the same outcomes as those presented in Sect. 6.7.1.3.

The above, together with the fact that the experiment run by exploiting Compose Object produced the same outcomes, show that we successfully ported the TOSKOSE-based orchestration approach on different Docker-based container orchestrators. It also shows that, while the management of containers changed accordingly to the employed container orchestrator, the actual management of the components running on such containers remained unchanged, as it was independent from the employed container orchestrator.

6.7.2 The *Sock Shop* Case Study

Sock Shop is an open-source web-based application simulating the user-facing part of an e-commerce website selling socks. It is developed and maintained by Weaveworks and Container Solutions, and its goal is to allow to test and showcase solutions and tools for orchestrating multi-component applications. *Sock Shop* is indeed a multi-component application, which components are the followings:

- A *Frontend* displays a web-based graphical user interface for e-shopping socks.
- Different pairs of services and databases allow to store and manage the catalogue of available socks (i.e., *Catalogue* and *CatalogueDB*), the users of the application (i.e., *Users*

and *UsersDB*), the users' shopping carts (i.e., *Carts* and *CartsDB*), and the users' orders (i.e., *Orders* and *OrdersDB*).

- Two services (i.e., *Payment* and *Shipping*) simulate the payment and shipping of orders.
- A message queue (i.e., *RabbitMQ*) allows to enqueue shipping requests, which are then consumed by a service (i.e., *Queue Master*) simulating the actual shipping of orders.

6.7.2.1 Modelling *Sock Shop* with TOSCA

We specified *Sock Shop* in TOSCA by exploiting the TOSCA-based representation given by TosKer [31] (Sect. 6.2.2). We modelled all databases and infrastructure components as nodes of type *tosker.nodes.Container*, which actual implementations were given by the Docker containers already configured by WeaveWorks and Container Solutions. We instead specified the services *Frontend*, *Catalogue*, *Users*, *Carts*, *Orders*, *Payment* and *Shipping* as nodes of type *tosker.nodes.Software*, each hosted on a node of type *tosker.nodes.Container* providing the runtime environment it needs (e.g., as *Frontend* requires NodeJS, it is hosted on a container implemented with the Docker image `node:6`). The resulting TOSCA-based representation of the application is shown in Fig. 6.20.

We also implemented the operations to `install`, `configure`, `start`, `stop` and `uninstall` the above mentioned services, each with a different shell script. In other words, for each service, the shell script `install.sh` installs the service in a dedicated folder of its hosting container, by cloning the GitHub repository containing its sources within such folder and by compiling (if needed) such sources. The script `configure.sh` configures the endpoints to be offered by the service. The scripts `start.sh` and `stop.sh` start and kill the process corresponding to the service, respectively. Finally, the script `uninstall.sh` deletes the folder containing the service installation.

The CSAR archive packaging the above described TOSCA application specification is publicly available on GitHub (`https://github.com/di-unipi-socc/toskose-packager/blob/master/tests/data/sockshop/sockshop.csar`).

6.7.2.2 Generating a Deployable Artifact for *Sock Shop*

We generated the Docker Compose file enabling a component-aware orchestration of *Sock Shop* on Docker-based container orchestrators by executing the TOSKOSE PACKAGER as follows:

```
$ toskose -p sockshop.csar toskose.yml
```

The archive `sockshop.csar` listed above was a local copy of the CSAR packaging *Sock Shop* available on GitHub (see Sect. 6.7.2.1), while `toskose.yml` is the Toskose configuration file shown in Fig. 6.21. The option `-p` is used to force pushing the automatically generated *toskosed* images to the Docker Hub.

A snippet of the Docker Compose file automatically generated by the TOSKOSE PACKAGER is shown in Fig. 6.22. The TOSKOSE PACKAGER introduced and suitably configured each container from the original *Sock Shop* application, and it also includes an additional container running the TOSKOSE MANAGER. All such containers are specified as `services` running on the same overlay network, i.e., `toskose-network`.

6.7.2.3 Deploying and Managing *Sock Shop* on Existing Container Orchestrators

We exploited the Docker Compose file obtained from the TOSKOSE PACKAGER to deploy *Sock Shop* with Docker Compose on the same cluster of virtual machines as that used for *Thinking* (Sect. 6.7.1.3), and for repeating the activities of stopping and restarting its components by keeping their hosting containers up and running. We also deployed and managed *Sock Shop* with Kubernetes on a cluster composed by four cloud-hosted virtual machines (i.e., four nodes hosted by Microsoft Azure with the Azure Kubernetes Service), in order to test whether our approach can also effectively in a real-world cloud scenario. For the sake of brevity, we hereafter only report on the Azure-based deployment and management of *Sock Shop*.

To enact a deployment of *Sock Shop* with the Azure Kubernetes Service, we were first required to setup a cluster on Azure where to run *Sock Shop*. We hence locally launched a containerised instance of the Azure CLI, which we first used to log into the platform:

```
$ docker run -it -v ${HOME}/.ssh:/root/.ssh mcr.microsoft.com/azure-cli
$ az login
```

We then exploited the Azure CLI to create our own resource group, which we used to create a cluster of four virtual machines. The latter was done by directly exploiting Azure Kubernetes Service (i.e., `aks`):

```
$ az group create --name MyResourceGroup --location eastus
$ az aks create --resource-group myResourceGroup --name AKSCluster \
                --dns-name-prefix AKSCluster-dns --node-count 4 \
                --node-vm-size Standard_DS1_v2 --enable-addons monitoring \
                --generate-ssh-keys --kubernetes-version 1.13.12
```

Once created, we were able to connect to the Kubernetes-managed cluster with the command:

```
$ az aks get-credentials --resource-group myResourceGroup --name AKSCluster
```

We then deployed the containers in *Sock Shop* by executing

```
$ kubectl create -f sockshop
```

where `sockshop` was the directory containing the Kubernetes specification automatically obtained from Kompose, to which we fed the Docker Compose file automatically generated by the TOSKOSE PACKAGER.

The above allowed us to deploy the containers of *Sock Shop* and have them up and running. The software components hosted on such containers were instead not deployed yet, as their actual management was to be orchestrated through TOSKOSE MANAGER. We hence completed the deployment of *Sock Shop* by (i) starting *Shipping* and *Carts* with

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/shipping/shipping-sw/start"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/carts/carts-sw/start"
```

(ii) installing and starting *Users*, *Payment* and *Catalogue* with

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/user/user-sw/create"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/user/user-sw/start"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/payment/payment-sw/create"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/payment/payment-sw/start"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/create"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/start"
```

(iii) installing and starting *Orders* with

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/orders/orders-sw/create"
$ curl -X POST -H "accept: application/json" \
 "http://52.142.38.162:10000/api/v1/node/orders/orders-sw/start"
```

and (iv) installing, configuring and starting *Frontend* with

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/front-end/front-end-sw/create"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/front-end/front-end-sw/configure"
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/front-end/front-end-sw/start"
```

As a result, we obtaind a running instance of *Sock Shop* allowing to browse among the catalogue of socks e-sold through its web-based portal (Fig. 6.23).

The above again showed us that we were able to piggyback on an existing Docker-based container orchestrator for deploying the containers forming an application, and to independently orchestrate the deployment of the components running on such containers by exploiting the TOSKOSE toolset. To further experiment the independent management of components and containers, we stopped the service managing the catalogue of available socks by issuing the following cURL command:

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/stop"
```

The above resulted in the web-portal no more being able to show the socks that can be e-shopped, because of the web-based *Frontend* not being able to connect to *Catalogue* (Fig. 6.24(a)). This happened even if the container hosting *Catalogue* was still working (Fig. 6.24(b)), hence showing that the lifecycle of *Catalogue* and of its hosting container were managed independently.

Finally, to return having the instance of *Sock Shop* properly working, we issued the cURL command

```
$ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/stop"
```

which (re-)started the *Catalogue* and brought the application back to offer a fully working web-based portal for e-shopping socks (such as that in Fig. 6.23).

6.8 Related work

Various solutions exist for orchestrating the management of multi-component applications, based on TOSCA or Docker. The closest to ours is TosKer[31], which —to the best of our knowledge— is currently the only solution enabling a component-aware orchestration of TOSCA-based application on top of Docker. It does so by implementing from scratch a new orchestration engine, allowing to coordinate the management of both the software components and the Docker containers forming an application. The TosKer engine is designed to run on a single host, which must be configured to provide root privileges to the engine itself (so as to allow it to spawn Docker containers and run application components on them). Our approach hence differs from that of TosKer, as we enable a component-aware management of TOSCA-based applications on top of existing container orchestrators, which natively support multi-host deployments. In addition, our approach does not need root priviliges to properly work, hence making it suited also for scenarios where such privileges cannot be granted (e.g., on Container-as-a-Service platforms).

Another closely related approach is that tackled by the EDMM modelling and transformation framework [165, 166]. Even if not TOSCA-based, the EDMM modelling and transformation framework allows to specify the software components and Docker containers forming a multi-component application, and the operation allowing to manage each of components and containers [166]. It

then support the automated generation of the artifacts for deploying the application on top of existing orchestrators, including Docker Compose and Kubernetes [165]. The latter essentially consists in creating deployment scripts coordinating the executable files implementing the management operations of the components of an application, in such a way that the dependencies occurring among such components are satisfied. It can hence be viewed as a solution for the component-aware deployment of multi-component applications on top of existing deployment platforms. However, once the deployment is enacted on a container orchestrator, the application is managed through such orchestrator, which considers containers as "black-boxes", i.e., not allowing to manage the components forming an application independently from the containers used to host them. Our approach hence differs from that EDMM-based, as we aim at supporting a component-aware management of multi-component applications during their whole lifecycle.

Considering containers as "black-boxes" is a baseline also shared by all other existing approaches trying to synergically combine the OASIS standard TOSCA and Docker for orchestrating multi-component applications. For instance, Kehrer and Blochinger [100] propose to use TOSCA for specifying the internals of a container, which are then manually built by developers to allow their orchestration (as "black-boxes") on top of Mesos. Our approach is instead intended to enable a component-aware management of multi-component applications on top of existing container orchestrators, by allowing to manage application components independently from their hosting containers.

Other approaches worth mentioning are OpenTOSCA [16], Alien4Cloud (`http://alien4cloud.github.io`), Cloudify (`https://cloudify.co`), and the Apache ARIA TOSCA incubator (`https://ariatosca.incubator.apache.org`). OpenTOSCA is an open-source engine for deploying and managing TOSCA applications, which components include containers. Alien4Cloud, Cloudify and ARIA TOSCA also allow to manage multi-component applications, which components include Docker containers. However, they all differ from our approach to managing application since they Docker containers as "black-boxes", i.e., not supporting the management of software components separately from that of the containers hosting them.

SeaClouds [23] and Apache Brooklyn (`https://brooklyn.apache.org`) also relate to our approach. SeaClouds [23] is a middleware solution for deploying and managing multi-component applications on heterogeneous IaaS/PaaS cloud infrastructure. SeaClouds fully supports TOSCA, but it lacks a support for Docker containers. The latter makes SeaClouds not suitable to manage multi-component applications including Docker containers.

Apache Brooklyn (`https://brooklyn.apache.org`) instead natively supports both TOSCA and Docker containers. Thanks to its extension called *Brooklyn-TOSCA* [42], Brooklyn enables the management of the software components and Docker containers forming an application. However, Brooklyn treats Docker containers as black-boxes, and this does not permit managing the components of an application independently of that of the containers used to host them.

It is finally worth relating our approach with currently existing solutions for orchestrating

multi-container Docker applications. Docker natively supports their orchestration by means of Docker Compose (`https://docs.docker.com/compose`), which allows to indicate the images of the Docker containers forming an application, the virtual network to setup to allow the to intercommunicate, and the volumes to mount to persist their data. Based on such information, Docker Compose can enact the deployment of the specified application. Docker Compose however treats containers as "black-boxes", meaning that there is no information on which components are running within a container, and since it does not allow to orchestrate the management of application components independently from their hosting containers. In addition, no information is provided on the actual interdependencies and interconnections occurring among the components and containers of an application. Our approach instead allows to explicitly model the software components forming an application, to orchestrate their management independently from their hosting containers, and to explicitly consider the different types of relationships occurring among the components and containers in an application. This not only makes the interactions occurring among the components of an application easier to understand, but also brings various advantages in terms of reuse and maintenance [26].

Other solutions worth mentioning are Docker swarm (`https://docs.docker.com/engine/swarm`), Kubernetes (`https://kubernetes.io`), and Mesos (`http://mesos.apache.org`). Docker swarm permits creating a cluster of replicas of a Docker container, and seamlessly managing it on a cluster of hosts. Kubernetes and Mesos instead permit automating the deployment, scaling, and management of containerized applications over clusters of hosts. Docker swarm, Kubernetes and Mesos differ from our orchestration system as they focus on how to schedule and manage containers (as "black-boxes") on clusters of hosts, while we aim at piggybacking on top of them to enable a component-aware orchestration of the management of multi-component applications.

Similar considerations apply to ContainerCloudSim [134], which provides support for modelling and simulating containerized computing environments. ContainerCloudSim is based on CloudSim [36], and it focuses on evaluating resource management techniques, such as container scheduling, placement and consolidation of containers in a data center, by abstracting from the application components actually running in such containers. Our solution instead focuses on allowing to independently manage the components forming an application while delegating container management to a container orchestrator.

6.9 Conclusions

We presented a solution enabling the component-aware management of multi-component application on top of existing Docker-based container orchestrators. More precisely, starting from an existing TOSCA-based representation for multi-component applications, we illustrated a novel approach allowing to manage the components forming an application independently from the Docker containers used to host them. We also introduced three open-source prototype tools

implementing our approach. These are TOSKOSE UNIT (i.e., a bundling of *Supervisor* allowing to remotely manage the components running in a container), TOSKOSE MANAGER (i.e., a containerised orchestrator allowing to coordinate the *Supervisor* instances running in the containers of the application), and TOSKOSE PACKAGER (i.e., a tool for automatically generating Docker-based deployable artifacts from the TOSCA specification of a multi-component application).

We also illustrated how our approach and prototype tools effectively enabled the component-aware management of two existing multi-component applications (i.e., *Thinking* and *Sock Shop*) on top of Docker Swarm and Kubernetes. After representing such applications in TOSCA, we exploited the TOSKOSE PACKAGER for generating deployable Docker Compose files. The latter were then effectively deployed with both Docker Swarm and Kubernetes on a cluster of virtual hosts. This allowed us to showcase that the containers forming the infrastructure of the considered applications were deployed and managed by relying on the capabilities of existing Docker-based container orchestrators, while the lifecycle of the software components hosted on such containers was independently orchestrated through the TOSKOSE MANAGER. The above held for both considered Docker container orchestrators (i.e., Docker Swarm and Kubernetes), and independently of whether the cluster of virtual hosts was running on premise or on a real cloud (i.e., MS Azure Cloud).

We believe that this paper can help researchers and practitioners wishing to independently orchestrate the components and containers forming an application. For instance, we discussed several issues while illustrating the development of our solution, e.g., signal management and zombie reaping, or potential conflicts when packaging multiple components in the same container. The discussion on issues and their possible solutions can be of help to researchers and practitioners needing to face similar problems while developing alternative solutions to ours, or simply because their applications need multiple components to reside in a single container.

In addition, the TOSKOSE open-source toolset can already be exploited (as-is) by researchers and practitioners to enable a component-aware orchestration of their applications on existing container orchestrators. The current prototype of TOSKOSE can also be exploited as the basis for the development of other research solutions or tools, or for validating existing approaches. For instance, we exploited TOSKOSE to further validate the outcomes of our former research, i.e., we exploited it to run TOSCA application specification automatically completed by TOSKERISER [26]. The latter is a tool for completing TOSCA application specifications, which automatically discovers and includes the Docker containers offering all what needed to run the components of an application, based on the information automatically retrieved by DOCKERFINDER [28]. Application specification generated by TOSKERISER were translated to Docker Compose files, which were then effectively orchestrated on existing Docker-based container orchestrators in a component-aware manner.

TOSKERISER, DOCKERFINDER and TOSKOSE actually form an open-source toolchain, which helps researchers and practitioners in automating the orchestration of multi-component ap-

plications with TOSCA and Docker (Fig. 6.25). They can indeed focus on only describing the components forming an application in TOSCA, and their runtime requirements. The TOSCA-based application representation is then automatically completed by TOSKERISER with the containers allowing to run its components, and then transformed by the TOSKOSE PACKAGER in a deployable solution (i.e., a Docker Compose file). The latter includes the TOSKOSE MANAGER and the *toskosed* images enabling a component-aware management of the application on top of existing Docker-based container orchestrators.

At the same time, the TOSKOSE open-source toolset requires to be further engineered to improve its capabilities. It currently features basic capabilities for scaling and self-healing components, fully relying on the mechanisms natively featured by the Docker-based container orchestrator employed for deploying an application. The minimal entity that can be scaled or self-healed is currently a container, and we are currently working on including support for component-aware scaling and self-healing.

We also plan to design and develop a support for automatically determining the workflow of management operations to invoke to allow an application to reach a desired configuration. Currently, the sequence of operations for reaching a given application configuration is to be manually issued by the application administrator. We plan to integrate existing analysis and planning techniques (e.g., based on Aeolus [53] or on management protocols [22]), in such a way that the administrator just instructs the TOSKOSE MANAGER with the desired application configuration, and the TOSKOSE MANAGER automatically issues the management operations allowing to reach and maintain such a configuration, even in presence of unexpected failures.

```
---
version: '3.7'
services:
  maven:
    image: giulen/thinking-maven-toskosed:0.1.3
    init: true
    networks:
      toskose-network: { aliases: [ maven ] }
    environment:
    - SUPERVISORD_ALIAS=maven
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=user_21ty5
    - SUPERVISORD_PASSWORD=1t5mYp4ss
    - SUPERVISORD_LOG_LEVEL=INFO
    - INPUT_REPO=https://github.com/matteobogo/thoughts-api
    - INPUT_BRANCH=master
    - INPUT_DBURL=mongodb
    - INPUT_DBPORT=27017
    - INPUT_DBNAME=thoughtsSharing
    - INPUT_COLLECTIONNAME=thoughts
    - INPUT_DATA=/toskose/apps/api/artifacts/default_data.csv
    - INPUT_PORT=8080
    ports: [ "8000:8080/tcp" ]
  node:
    image: giulen/thinking-node-toskosed:2.1.5
    init: true
    networks:
      toskose-network: { aliases: [ node ] }
    environment:
    - SUPERVISORD_ALIAS=node
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=user_a4bc2
    - SUPERVISORD_PASSWORD=p4ssw0rd
    - SUPERVISORD_LOG_LEVEL=DEBUG
    - INPUT_REPO=https://github.com/matteobogo/thoughts-gui
    - INPUT_BRANCH=master
    - INPUT_APIURL=localhost
    - INPUT_APIPORT=8000
    - INPUT_APIRESOURCE=thoughts
    ports: [ "8080:3000/tcp" ]
  mongodb:
    image: mongo:3.4
    init: true
    networks:
      toskose-network: { volumes: [ "dbvolume:/data/db" ] }
  toskose-manager:
    image: giulen/thinking-manager-toskosed:latest
    init: true
    deploy: *id001
    networks:
      toskose-network: { aliases: [ toskose-manager ] }
    environment:
    - TOSKOSE_MANAGER_PORT=12000
    - TOSKOSE_APP_MODE=production
    - SECRET_KEY=my_secret
    ports: [ "12000:12000/tcp" ]
networks:
  toskose-network: { driver: "overlay", attachable: true }
volumes:
  dbvolume:
```

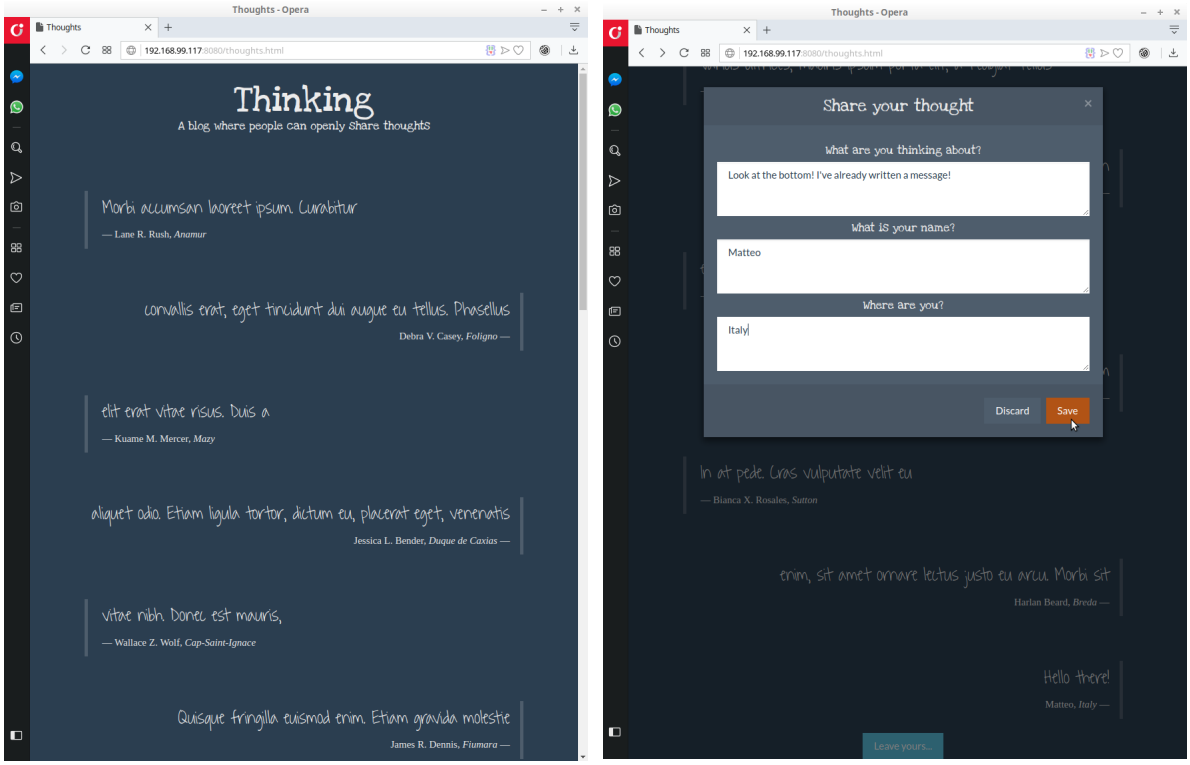
FIGURE 6.14. Docker Compose file for deploying *Thinking*, automatically generated by the TOSKOSE PACKAGER.

```
→ tmp docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
virtual-node-1   *        virtualbox   Running   tcp://192.168.99.114:2376           v18.09.9
virtual-node-2   -        virtualbox   Running   tcp://192.168.99.115:2376           v18.09.9
virtual-node-3   -        virtualbox   Running   tcp://192.168.99.116:2376           v18.09.9
virtual-node-4   -        virtualbox   Running   tcp://192.168.99.117:2376           v18.09.9
→ tmp docker node ls
ID                            HOSTNAME            STATUS        AVAILABILITY        MANAGER STATUS      ENGINE VERSION
70v67iic6qt0ox23uc43tm41q *   virtual-node-1      Ready         Active              Leader              18.09.9
01doyi50ofqc1sewk7klzw84u     virtual-node-2      Ready         Active                                  18.09.9
wv81zrzj3tpq0ua8m5perut7g     virtual-node-3      Ready         Active                                  18.09.9
lovqkjg8369ojm29k5euv5tn5     virtual-node-4      Ready         Active                                  18.09.9
```

FIGURE 6.15. Multi-host cluster provisioned for allowing the deployment of *Thinking* with Docker Swarm.

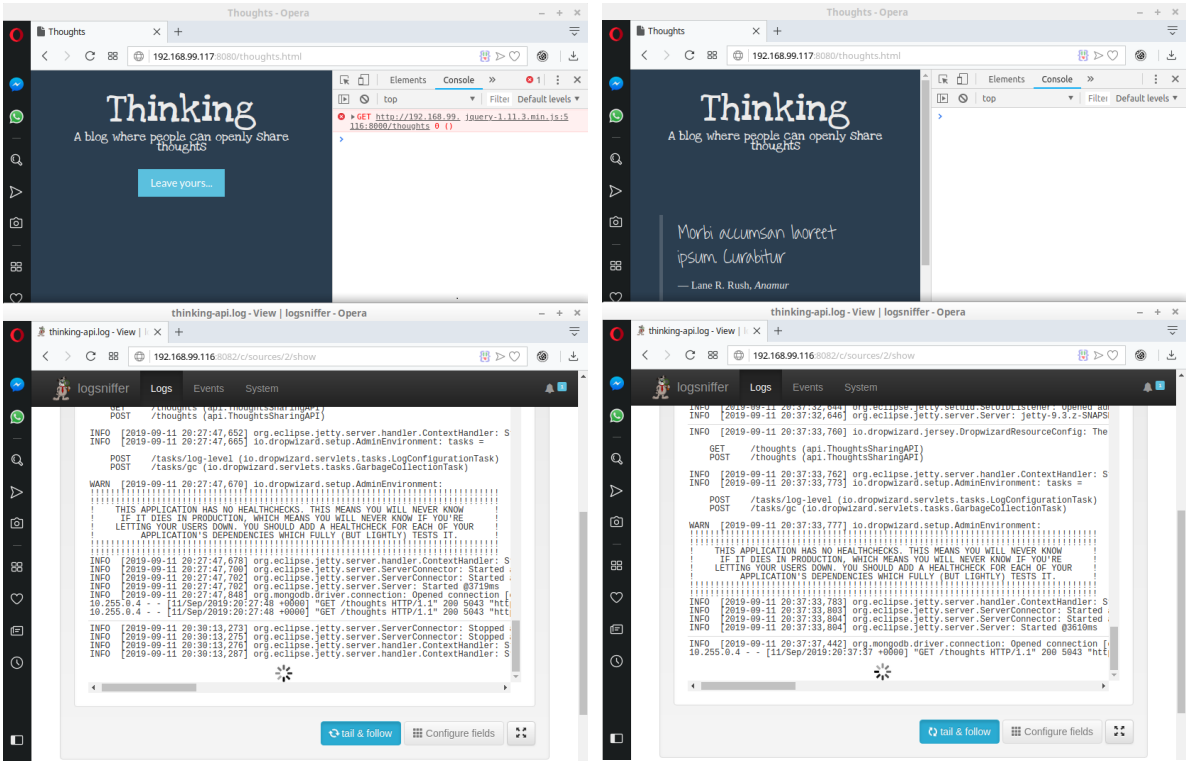
```
→ tmp docker stack deploy --compose-file docker-compose.yml --orchestrator swarm thinking-stack
Creating network thinking-stack_toskose-network
Creating service thinking-stack_node
Creating service thinking-stack_mongodb
Creating service thinking-stack_toskose-manager
Creating service thinking-stack_maven
→ tmp docker stack ps thinking-stack
ID                  NAME                                IMAGE                                          NODE
w9ocmq1zlx0s        thinking-stack_maven.1              giulen/thinking-v2-maven-toskosed:0.1.3        virtual-node-3
m32m9zojmcug        thinking-stack_toskose-manager.1    giulen/thinking-v2-manager-toskosed:latest     virtual-node-2
1nvy2a5vvia7        thinking-stack_mongodb.1            mongo:3.4                                      virtual-node-1
p24rndong7n6        thinking-stack_node.1               giulen/thinking-v2-node-toskosed:2.1.5         virtual-node-4
→ tmp docker service ls
ID                  NAME                              MODE         REPLICAS  IMAGE
j2sbjxn8em7m        thinking-stack_maven              replicated   1/1       giulen/thinking-v2-maven-toskosed:0.1.3
p6fzex24de05        thinking-stack_mongodb            replicated   1/1       mongo:3.4
u3l5vzy8r2mf        thinking-stack_node               replicated   1/1       giulen/thinking-v2-node-toskosed:2.1.5
xh5401y3az24        thinking-stack_toskose-manager    replicated   1/1       giulen/thinking-v2-manager-toskosed:latest
```

FIGURE 6.16. Execution and outcomes of the command for deploying *Thinking* on our Swarm cluster.



(a) (b)

FIGURE 6.17. Snapshots of the running instance of *Thinking* obtained after completing its deployment. The snapshots show the web-based interfaces for (a) reading shared thoughts and for (b) sharing new thoughts.



(a) (b)

FIGURE 6.18. Snapshots of the running instance of *Thinking* and of the logs sniffed by the *Logsniffer*, (a) after stopping the *GUI* and (b) after restarting it.

```
→ tmp kompose up docker-compose.yml
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims for your Dockerized application. If you need
different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands instead.

INFO Deploying application in "default" namespace
INFO Successfully created Service: maven
INFO Successfully created Service: node
INFO Successfully created Service: toskose-manager
INFO Successfully created Pod: maven
INFO Successfully created Pod: mongodb
INFO Successfully created PersistentVolumeClaim: dbvolume of size 100Mi. If your cluster has dynamic storage provisioning, you don't
have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created Pod: node
INFO Successfully created Pod: toskose-manager

Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,pods,pvc' for details.
→ tmp kubectl get deployment,svc,pods,pvc

NAME                       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
service/kubernetes         ClusterIP   10.96.0.1       <none>        443/TCP             15h
service/maven              ClusterIP   10.104.191.58   <none>        8000/TCP,8082/TCP   15s
service/node               ClusterIP   10.106.20.66    <none>        8080/TCP            15s
service/toskose-manager    ClusterIP   10.97.103.166   <none>        12000/TCP           15s

NAME                  READY   STATUS    RESTARTS   AGE
pod/maven             1/1     Running   0          15s
pod/mongodb           1/1     Running   0          15s
pod/node              1/1     Running   0          15s
pod/toskose-manager   1/1     Running   0          15s

NAME                                STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/dbvolume      Bound    pvc-314187ba-5eb4-463d-9684-aed5d1eaa739   100Mi      RWO            standard       15s
```

FIGURE 6.19. Execution and outcomes of the command for deploying *Thinking* on Kubernetes (with Kompose).

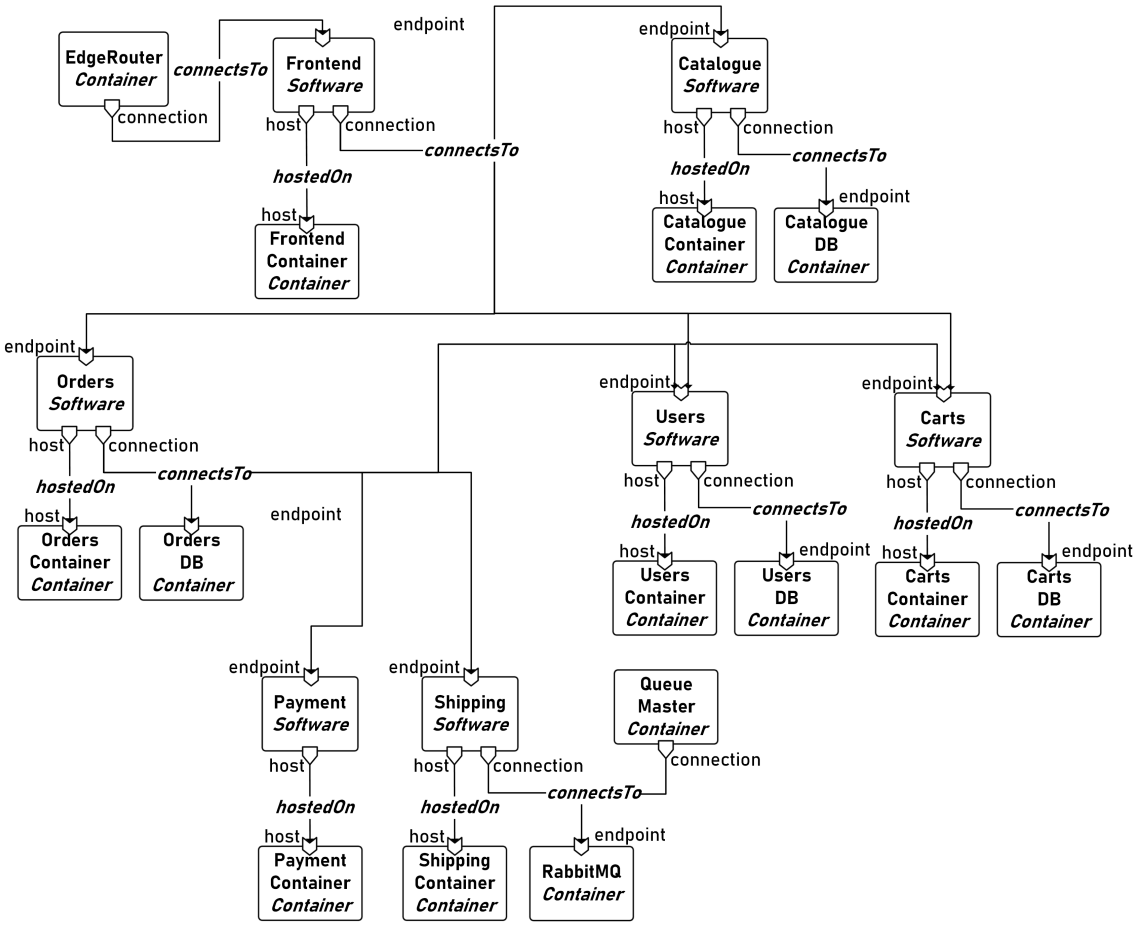


FIGURE 6.20. TOSCA-based representation of *Sock Shop*, obtained by exploting the TOSCA types defined by TosKer [31].

```
nodes:
  front-end_container:
    alias: front-end
    port: 9001
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/sockshop-front-end_container-toskosed, tag: latest }
  catalogue_container:
    alias: catalogue
    port: 9002
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/sockshop-catalogue_container-toskosed, tag: latest }
  user_container:
    alias: user
    port: 9003
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/sockshop-user_container-toskosed, tag: latest }
  carts_container:
    alias: carts
    port: 9004
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/carts_container-toskosed, tag: latest }
  orders_container:
    alias: orders
    port: 9005
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/orders_container-toskosed, tag: latest }
  payment_container:
    alias: payment
    port: 9006
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/payment_container-toskosed, tag: latest }
  shipping_container:
    alias: shipping
    port: 9007
    user: admin
    password: admin
    log_level: INFO
    docker: { name: giulen/shipping_container-toskosed, tag: latest }
manager:
  alias: toskose-manager
  port: 10000
  user: admin
  password: admin
  mode: production
  secret_key: secret
  docker: { name: giulen/sockshop-manager, tag: latest }
```

FIGURE 6.21. Toskose configuration file used for generating a deployable artifact for *Sock Shop*.

```
---
version: '3.7'
services:
  orders-db:
    image: mongo:latest
    init: true
    networks:
      toskose-network:
  front-end_container:
    image: giulen/sockshop-front-end_container-toskosed:latest
    init: true
    networks:
      toskose-network: { aliases: [ front-end ] }
    environment:
    - SUPERVISORD_ALIAS=front-end
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=admin
    - SUPERVISORD_PASSWORD=admin
    - SUPERVISORD_LOG_LEVEL=INFO
    - INPUT_REPO=https://github.com/matteobogo/front-end.git
    - INPUT_CATALOGUE=catalogue
    - INPUT_CARTS=carts
    - INPUT_USER=user
    - INPUT_ORDERS=orders
  catalogue_container:
    image: giulen/sockshop-catalogue_container-toskosed:latest
    init: true
    networks:
      toskose-network: { aliases: [ catalogue ] }
    environment:
    - SUPERVISORD_ALIAS=catalogue
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=admin
    - SUPERVISORD_PASSWORD=admin
    - SUPERVISORD_LOG_LEVEL=INFO
    - INPUT_PORT=80
  ...
  toskose-manager:
    image: giulen/sockshop-manager:latest
    networks:
      toskose-network: { aliases: [ toskose-manager ] }
    init: true
    environment:
    - TOSKOSE_MANAGER_PORT=10000
    - TOSKOSE_APP_MODE=production
    - SECRET_KEY=secret
    ports: [ "10000:10000/tcp" ]
  ...
networks:
  toskose-network: { driver: "overlay", attachable: true }
```

FIGURE 6.22. A snippet of the Docker Compose file for deploying *Sock Shop*. The full version of the file is available on GitHub (`https://github.com/di-unipi-socc/toskose-packager/blob/master/tests/data/sockshop/docker-compose.yml`).

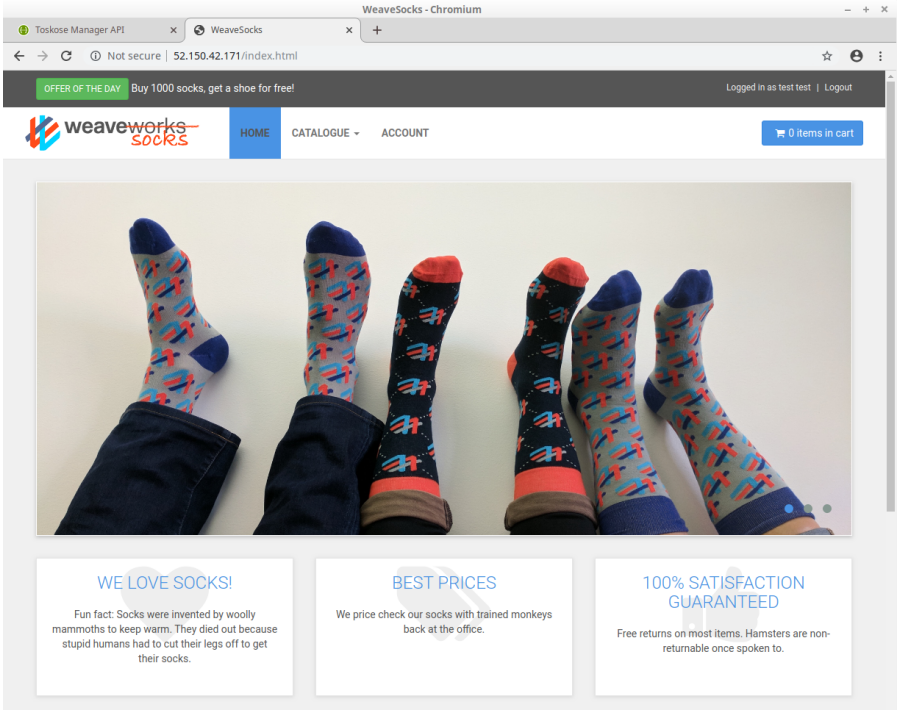
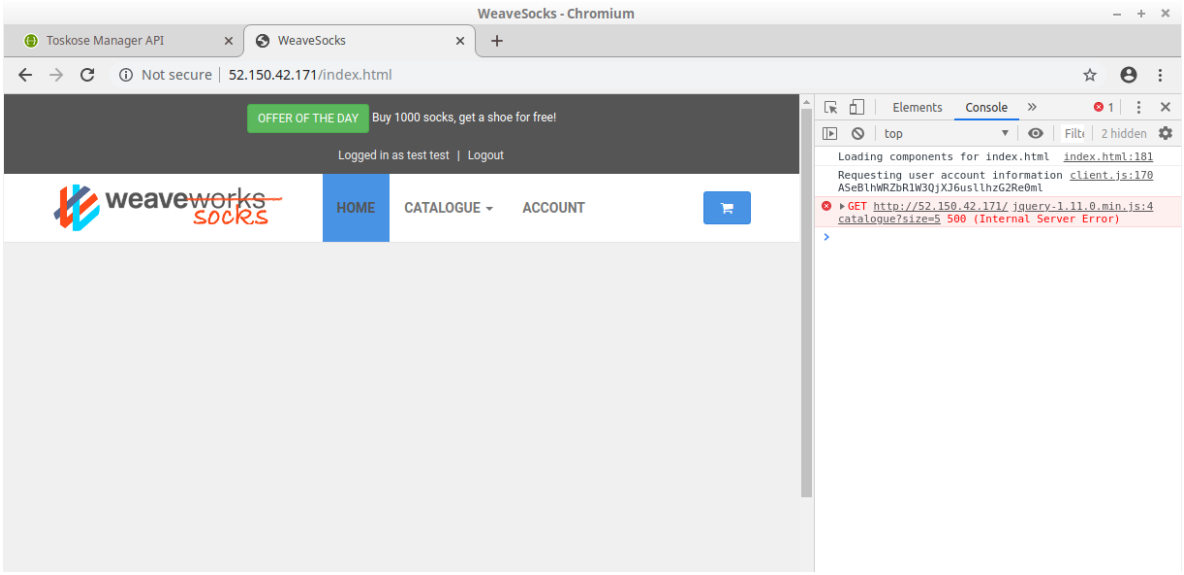


FIGURE 6.23. Snapshot of the running instance of *Sock Shop* obtained after completing its deployment.



(a)

```
→  kubernetes git:(master) ✗ kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
carts             1/1     1            1           4h42m
carts-db          1/1     1            1           4h42m
catalogue         1/1     1            1           4h42m
catalogue-db      1/1     1            1           4h42m
dev-pod           1/1     1            1           119m
edge-router       1/1     1            1           4h42m
front-end         1/1     1            1           4h42m
orders            1/1     1            1           4h42m
orders-db         1/1     1            1           4h42m
payment           1/1     1            1           4h42m
rabbitmq          1/1     1            1           4h42m
shipping          1/1     1            1           4h42m
toskose-manager   1/1     1            1           4h42m
user              1/1     1            1           4h42m
user-db           1/1     1            1           4h42m
→  kubernetes git:(master) ✗
```

(b)

FIGURE 6.24. Snapshot of (a) the instance of *Sock Shop* after stopping its *Catalogue*, and of (b) the containers actually running while such instance was not properly working.

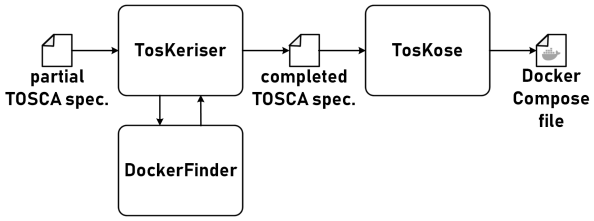


FIGURE 6.25. Open-source toolchain for generating deployable solutions from partial TOSCA application specifications, i.e., specifications only indicating the components forming an application and the requirements they need to run.

Part IV

Closing

CHAPTER 7

CONCLUSIONS

In this chapter we summarise the research contributions contained in this thesis (see Sect. 7.1), and we discuss the assessment of the contributions (see Sect. 7.2). We also give some perspectives for future work (see Sect. 7.3).

7.1 Summary of contributions

The aim of this thesis is to enhance the support for designing and deploying microservice-based applications by advancing the state of the art for (o_1) analysing and refactoring microservices, (o_2) automatically packaging microservices, and (o_3) enacting the deployment of microservices. We hereafter summarise the research contributions in this thesis, by separately discussing each of the three research objectives o_1, o_2 and o_3.

Analysing and refactoring microservices

We illustrated the outcomes of a multivocal review focused on identifying architectural smells denoting possible violations of key design principles of microservices (viz., independent deployability, horizontal scalability, fault isolation and decentralisation), as well as the refactorings allowing to resolve such smells (Chapter 2). We indeed presented a taxonomy organising seven architectural smells and 16 refactorings, by associating each smell with the possibly violated design principle, and each refactoring with the smell it resolves. We then discussed the perceived impact of such smells and refactorings, as well as why an architectural smell violates a design principle and why an architectural refactoring resolves its corresponding smell —according to the authors of the selected literature.

Starting from the outcomes of the above mentioned review, we presented a methodology to systematically identify the architectural smells that possibly violate key design principles of microservices (Chapter 3). We first presented μTOSCA, a model (based on the OASIS standard TOSCA [124]) allowing to specify the architecture of a microservice-based application. Based on such representation, we formally defined the conditions to identify the occurrence of the considered architectural smells in a microservice-based application, and we illustrated how to refactor its architecture to resolve identified smells. We also presented μFRESHENER, a prototype showcasing our methodology.

Automatically packaging microservices

We presented a TOSCA-based representation for multi-component applications, which allows developers to describe *only* the components forming an application, the dependencies among such components, and the software support needed by each component. We have also presented TOSKERISER, a tool which permits to automatically package the TOSCA specification, by automatically discovering and configuring the Docker containers needed to host its microservices (Chapter 4). We have also shown how TOSKERISER permits changing –when/if needed– the deployment requirements (e.g., operating system, versions of the libraries) of a microservice and it automatically searches and updates the corresponding container.

In order to automatically discover the appropriate container for each microservice, TOSKERISER exploits DOCKERFINDER, i.e., a Docker Image analyser built with DOCKERANALYSER. DOCKERANALYSER (Chapter 5) is a microservice-based tool that permits building customised analysers of Docker images. We showed how users can build their own image analysers by instantiating DOCKERANALYSER with a custom analysis function. We have shown two concrete instance of analysers, namely (i) the DOCKERFINDER analyser exploited by TOSKERISER and (ii) DOCKERGRAPH, which analyses the sources of Docker images stored in their corresponding GitHub repository to determine dependencies between existing images.

Enacting the deployment of microservices

We proposed a novel architectural approach for deploying microservice-based applications on top of existing container orchestrators, by also allowing to manage each microservice independently from the container used to run it (Chapter 6). Such an approach allows to (i) manage a component independently from the container running it, (ii) coordinating the management of multi-component applications, and (iii) generating application deployments that can be enacted by existing container orchestrators. We presented the implementation of TOSKOSE, a tool that takes as input a TOSCA-based representation of an application and packages the components of the application in their hosting containers together with Supervisor (by exploiting TOSKOSE UNIT), as well as including an instance of the TOSKOSE MANAGER to enable the component-aware management of the application. We have also discussed how the deployment artifact

produced by TOSKOSE can be deployed on top of existing container orchestrators that use Docker as container-based virtualisation (i.e., Docker Swarm [62], Kubernetes[157]).

7.2 Assessment of contributions

As we discussed in Sect. 1.1, this thesis aims at advancing the state-of-the art for (o_1) analysing and refactoring microservices, (o_2) automatically packaging microservices and (o_3) enacting the deployment of microservices. We hereby assess the research contributions in this thesis first with respect to o_1, and then with respect to o_2 and o_3.

Analysing and refactoring microservices

Our multivocal review of design principles, architectural smells and refactoring of microservices can provide benefits to researchers interested in microservices. Indeed, a systematic presentation of the state of the art and practice on architectural smells and refactorings for microservices provides a body of knowledge to develop new theories and solutions, to analyse and experiment research implications, and to establish future research directions.

Our review can also help practitioners as, together with the reviews by Carrasco et al. [41] and by Taibi et al. [154], our results can help them in getting acquainted with the well-known architectural smells for microservices, and in choosing the refactorings allowing to resolve such smells. This can have a pragmatic value for practitioners, who can use our study as a starting point for microservices experimentation or as a guideline for day-by-day work with microservices.

The methodology presented in Chapter 3 can also support researchers and practitioners in designing microservice-based applications, as it can automatically detect the architectural smells affecting an application and it can help tp decide which refactorings to apply to resolve identified smells.

The feasibility of approaches based on our methodology has been illustrated with the μFRESHENER prototype presented in Sect. 3.4. In order to assess its potential, we have also run a case study and a controlled experiment, which are presented in the Appendix A.

Automatically packaging microservices

The methodology proposed by TOSKERISER can help developers and operators to containerise microservices. Indeed, it permits to describe only the components forming an application, the dependencies occurring among them, and the software support that each component requires, and TOSKERISER automatically completes such specifications by discovering and including the proper container. It does so by exploiting the discovery support provided by DOCKERFINDER.

In order to assess our methodology, in Chapter 4 we compared TOSKERISER with respect to the classical approach for packaging applications into Docker containers, based on three KPIs

(i.e., lines of code to be added/changed/deleted, files to be added/changed/deleted, and programming languages employed). We indeed evaluated the effort (i) for performing the first deployment of a newly developed application and for (ii) maintaining an existing, third-party application, with and without our solution. The results showed that the initial effort required by our solution is slightly higher (in terms of lines of code and number of files) than that currently required by Docker, but we also demonstrated that it actually pays off while maintaining an application (e.g., when the requirements of components change). Our approach indeed outperformed the Docker-based deployment in term of application maintainability, since it requires little intervention on the specification with respect to the classical Docker-based deployment.

The analysis and discovery support provided by DOCKERANALYSER (Chapter 5) can help researchers and practitioners interested in building custom Docker analysers. In order to showcase the possibility of building such custom Docker analysers, we showed how we exploited DOCKERANALYSER to run two concrete case studies consisting in configuring and running two different analysers (i.e., DOCKERFINDER and DOCKERGRAPH). Other than that, we showed that the choice of implementing DOCKERANALYSER with a microservice-based architecture eases building customisable and scalable analysers, also thanks to the replaceability and scalability natively supported by its microservice-based architecture. In particular, we showed how replaceability allows obtaining a different analyser from an existing one by only updating the configuration of the microservice running the analysis function on Docker images. We also showed how scalability allows improving the performance of obtained analysers, by illustrating how scaling the microservices of DOCKERFINDER effectively reduced the time needed to analyse Docker images.

It is finally worth noting that, beyond being exploited by TOSKERISER for automatically discovering containers, the DOCKERFINDER analyser obtained from DOCKERANALYSER has also been exploited to achieve other research results. For instance, by applying explainable data mining approaches to the data gathered by DOCKERFINDER, we illustrated how the internals of a Docker image impact on its popularity [78], and we exploited this relation to develop a support for designing Docker images with a higher chance of widespreading [79].

Enacting the deployment of microservices

The architectural approach and tools presented in Chapter 6 allows orchestrating the management of services independently from that of the containers used to host them. We have shown how TOSKOSE can be used to manage any type of service-based application (and not only microservices) by allowing to independently manage each service without stopping the container.

To illustrate and assess what above, we exploited our solution to run two concrete case studies, based on two different, already existing applications. We processed both applications with our tool, in order to generate concrete artifacts for deploying and managing them with Docker, i.e., their specification in Docker Compose. We then deployed both applications by running

the corresponding deployment artifacts with two reference container orchestrators (i.e., Docker Swarm and Kubernetes), both in a single-host and in a multi-host setting. Finally, we tested our orchestration system by managing the lifecycle of the microservices forming the applications independently from that of their hosting containers. We indeed showed that (in all run application deployments) our solution effectively enabled the management of a single microservice running in a container, by allowing to stop and restart the considered microservice without touching its hosting containers.

7.3 Possible directions for future work

We hereafter elicit some possible directions for future work, geared towards further supporting software developers and administrators in analysing and deploying their microservice-based applications.

Enhancing the support for analysing and refactoring microservices. We plan to extend the set of architectural smells that can be identified and resolved with our methodology (and with μFRESHENER). For instance, we plan to extend our methodology by allowing to identify the architectural smells proposed by Carrasco et al. [41] and by Taibi et al. [154], by also including architectural refactorings allowing to resolve such smells.

Another interesting direction of future work is to extend our methodology in such a way it suggests the refactoring to apply to resolve an architectural smell, by taking into account the context and surroundings where such smell is occurring. For example, the occurrence of a WOBBLY SERVICE INTERACTION smell may be resolved differently depending on whether it is affecting a synchronous or asynchronous interaction, with circuit breakers being more suited solution in the former case, and timeouts being more suited in the latter case [44].

We are also already working on considering teams of developers while analysing and refactoring microservice-based applications. We already allow to specify the assignment of microservices to the team of developers working on them, and we are investigating how team assignment can impact on the choice of which smells to resolve, and how. Consider, for instance, the case of adding a *Message Broker* for resolving an ENDPOINT-BASED SERVICE INTERACTION between two services. Whether and how to apply such refactoring depends on whether the involved services are managed by the same team or by different teams. In the first case, the team is owning both services, and it can proceed by applying the refactoring on its responsibility and control. The same does not apply if the services are assigned to different teams, as such teams need to interact among them and with the tribe leader. To support such a kind of reasoning, we have extended μTOSCA and μFRESHENER to explicitly represent team assignment (i.e., which components are assigned to which team). We now plan to investigate how to perform team-aware analyses (e.g., how team assignment impacts smells and refactorings, smells due to team assignment) and to extend μFRESHENER correspondingly.

We are also working on extending our methodology and μFRESHENER to also consider containers and container orchestration while analysing and refactoring microservice-based applications. Depending on the chosen container orchestrator (e.g., Docker Compose, Kubernetes) and of related support (e.g., Istio [94]), some architectural smells may be possibly resolved by the orchestrator itself. For instance, by simply setting ingress nodes in Kubernetes one can add API gateways to an application, hence potentially resolving NO API GATEWAY architectural smells. We already included a support for automatically completing the μTOSCA specification of the architecture of microservice-based applications based on their Kubernetes-based deployment in μFRESHENER. We now plan to work on further investigating how container orchestration impacts on smells and refactorings, as well as on identifying, classifying and resolving architectural smells related to the container-based orchestration of a given microservice-based application.

In addition, the mapping between software components and containers deals with an important clustering problem, occurring whenever N components need to be deployed into M containers (with $N > M$). We hence plan to extend our approach in order to support the choice of optimal deployment configurations. A optimal deployment configuration is a mapping of components into containers that optimize the used resources of the overall application based on some heuristics. For example, by using the result obtained in Section 4.6.3, an heuristic can be develop in order to suggest the optimal groups of multiple components to be deployed into single containers in order to reduce the overall traffic generated by the application.

It is also worth observing that our methodology uses TOSCA as modelling language but it can be also applied using other modelling languages. Indeed, out methodology defines the architecture of microservice-based applications at a high level (e.g., a directed graph with nodes, relationships, and groups) and it provides formal conditions for identifying smells and resolving them thorough refactorings. Given the formal definition, the methodology can be defined using other modelling languages by following the defined definition.

Fostering the usability of our approach. As presented in Chapter 3, our approach allows to analyse and refactor microservice-based applications for resolving potential architectural smells. This comes at the price of modelling the architecture of the application in μTOSCA, either manually or by exploiting the editing support provided by μFRESHENER. Such an approach may, however, result cumbersome, also because production-ready microservice-based applications can involve hundreds of interacting components [147].

To further support developers of microservice-based applications, we plan to foster the usability and scalability of our approach, by developing novel solutions allowing to automatically determine the architecture of an existing application. We are already developing one such solution, whose aim is to automatically extract the μTOSCA specification a running microservice-based application. In line with the tools presented in [136] , starting from the deployment artifacts of an application (i.e., the Kubernetes and Istio deployment files), the solution we are currently developing will generate the corresponding μTOSCA specification by monitoring its actual

runtime behavior (i.e., the interactions occurring among its components).

We are also working on featuring a team-wise usage of μFRESHENER, so that a team of developers can focus only on analysing and refactoring the microservices under its responsibility. μFRESHENER has indeed already been extended to explicitly represent team assignment (i.e., which components are assigned to which team) and it is allowing to restrict the focus of the editing, analysis and refactoring to only the microservices that are assigned to a team. We are now working on extending the team-wise usage of μFRESHENER, not only for featuring team-aware analyses and refactorings (as discussed above), but also to support the interactions among teams and tribe leaders in a multi-tenant fashion. Having such team-wise support, our methodology could better support applications consisting of very high numbers of microservices. Indeed it would permit to view, analyse and refactoring only a subset of the microservices (i.e., those related to only a single team of developers) without needing to manage all the microservices in the architecture.

Finally, we plan to permit to use our methodology for analysing and packaging microservices even if the application is described using proprietary management languages. In the line of [164], we want to develop solutions for automatically translating applications described with proprietary languages into our TOSCA-based modelling. Indeed, a future work is to extend the approach presented in Section 4 in order to complete also proprietary languages and not only TOSCA-based specification. In addition, we want to extend the TOSKERISER tool in order to build directly Docker images based on the information of the TOSCA specification.

Cost and QoS. As per their current formulation, the analysis, refactoring and deployment approaches proposed in this thesis do not take into account costs and QoS. Interesting questions to answer would anyway be whether and how an architectural smell can impact on the cost of running and application, or on its ensured QoS. Similarly, it would be interesting to understand whether choosing one container or another for packaging a microservice can impact on the overall cost and QoS of an application, and the same for the employed container orchestration solution. To answer these questions, we plan to extend our solutions to take into account cost and QoS as well, in particular to drive the refactoring and packaging of a microservice-based application based on constraints on acceptable costs (both in terms of implementation costs and runtime execution costs) and on desired QoS.

Engeneering our toolchain. As mentioned in Chapter 1, the contributions of this thesis can form a toolchain (Fig 6.25) aimed at supporting developers and operators of microservices, from the design to deployment. The first step is to engineer our toolchain, to improve its usability and reliability. We indeed plan to provide an integrated dashboard for jointly managing and configuring the tools forming our toolchain. Currently, users are required to use such tools separately, by taking care of manually configuring and executing each of them. The availability of a dashboard would allow developers and operators to use our tools as an integrated solution, by also automating the process of configuration and management.

Another improvement in this direction comes from allowing users to seamlessly alternate between the TOSCA-based modellings employed for analysing and deploying microservice-based applications. Our toolchain currently employs the μTOSCA modelling for describing the architecture of a microservice-based application, and the TosKer modelling for describing the mapping of microservices into containers. As μTOSCA and TosKer are both TOSCA-based, and since they represent similar entities in different ways, the process of switching from μTOSCA to TosKer (and vice versa) can be easily automated. We plan to develop an intermediate translator, allowing the μTOSCA output of μFRESHENER to be automatically translated to the TosKer input of TosKeriser, to enable their seamless integration.

BIBLIOGRAPHY

[1] *Business process model and notation (bpmn), version 2.0*. OMG Standard, 2011.

[2] *Api design guide*. Google Cloud Docs, `https://cloud.google.com/apis/design/`, 2019.

[3] *Run multiple services in a container*. Docker Documentation, `https://docs.docker.com/config/containers/multi-service_container`.

[4] *Docker CLI reference*. Docker Documentation, `https://docs.docker.com/engine/reference/commandline/docker`.

[5] *Deploy a stack to a swarm*. Docker Documentation, `https://docs.docker.com/engine/swarm/stack-deploy/`.

[6] E. ÁBRAHÁM, F. CORZILIUS, E. B. JOHNSEN, G. KREMER, AND J. MAURO, *Zephyrus2: On the fly deployment optimization using smt and cp technologies*, in Dependable Software Engineering: Theories, Tools, and Applications: Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings, M. Fränzle, D. Kapur, and N. Zhan, eds., vol. 9984 of LNCS, Springer International Publishing, 2016, pp. 229–245.

[7] V. ALAGARASAN, *Seven microservices anti-patterns*. InfoQ, `https://www.infoq.com/articles/seven-uservices-antipatterns`, 2015.

[8] N. ALSHUQAYRAN, N. ALI, AND R. EVANS, *A systematic mapping study in microservice architecture*, in 2016 IEEE 9th Int. Conf. on Service-Oriented Computing and Applications (SOCA), 2016, pp. 44–51.

[9] D. ARCELLI, V. CORTELLESSA, AND D. D. POMPEO, *Automating performance antipattern detection and software refactoring in UML models*, in 2019 Int. Conf. on Software Analysis, Evolution and Reengineering, IEEE, 2019, pp. 639–643.

[10] M. ARMBRUST, A. FOX, R. GRIFFITH, A. D. JOSEPH, R. KATZ, A. KONWINSKI, G. LEE, D. PATTERSON, A. RABKIN, I. STOICA, AND M. ZAHARIA, *A view of cloud computing*, Communications of the ACM, 53 (2010), pp. 50–58.

[11] U. AZADI, F. A. FONTANA, AND D. TAIBI, *Architectural smells detected by tools: a catalogue proposal*, in International Conference on Technical Debt (TechDebt 2019), 2019.

[12] A. BALALAIE, A. HEYDARNOORI, AND P. JAMSHIDI, *Microservices architecture enables devops: Migration to a cloud-native architecture*, IEEE Software, 33 (2016), pp. 42–52.

[13] A. BALALAIE, A. HEYDARNOORI, P. JAMSHIDI, D. A. TAMBURRI, AND T. LYNN, *Microservices migration patterns*, Software: Practice and Experience, 48 (2018), pp. 2019–2042.

[14] L. BASS, I. WEBER, AND L. ZHU, *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional, 2015.

[15] R. BHOJWANI, *Design patterns for microservices*. DZone, `https://dzone.com/articles/design-patterns-for-microservices`, 2018.

[16] T. BINZ, U. BREITENBÜCHER, F. HAUPT, O. KOPP, F. LEYMANN, A. NOWAK, AND S. WAGNER, *OpenTOSCA – a runtime for TOSCA-based cloud applications*, in Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings, S. Basu, C. Pautasso, L. Zhang, and X. Fu, eds., Berlin, Heidelberg, 2013, Springer, pp. 692–695.

[17] T. BINZ, U. BREITENBÜCHER, O. KOPP, AND F. LEYMANN, *TOSCA: Portable automated deployment and management of cloud applications*, in Advanced Web Services, A. Bouguettaya, Q. Sheng, and F. Daniel, eds., New York, NY, 2014, Springer, pp. 527–549.

[18] T. BINZ, C. FEHLING, F. LEYMANN, A. NOWAK, AND D. SCHUMM, *Formalizing the cloud through enterprise topology graphs*, in 2012 IEEE Fifth International Conference on Cloud Computing, June 2012, pp. 742–749.

[19] J. BOGNER, T. BOCECK, M. POPP, D. TSCHECHLOV, S. WAGNER, AND A. ZIMMERMANN, *Towards a collaborative repository for the documentation of service-based antipatterns and bad smells*, in 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019, pp. 95–101.

[20] M. BOGO, J. SOLDANI, D. NERI, AND A. BROGI, *Component-aware orchestration of cloud-based enterprise applications, from tosca to docker and kubernetes*, 2020. `https://arxiv.org/abs/2002.01699`.

[21] J. BONÉR, *Reactive Microservice Architecture: Design Principles for Distributed Systems*, O'Reilly, 2016.

[22] A. BROGI, A. CANCIANI, AND J. SOLDANI, *Fault-aware application management protocols*, in Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings, M. Aiello, B. E. Johnsen, S. Dustdar, and I. Georgievski, eds., Springer, 2016, pp. 219–234.

[23] A. BROGI, J. CARRASCO, J. CUBO, F. D'ANDRIA, A. IBRAHIM, E. PIMENTEL, AND J. SOLDANI, *EU Project SeaClouds - adaptive management of service-based applications across multiple clouds*, in Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014), 2014, pp. 758–763.

[24] A. BROGI, G. MENCAGLI, D. NERI, J. SOLDANI, AND M. TORQUATI, *Container-based support for autonomic data stream processing through the fog*, in Euro-Par 2017: Parallel Processing Workshops, D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, eds., vol. 10659 of LNCS, Springer International Publishing, 2018, pp. 17–28.

[25] A. BROGI, D. NERI, L. RINALDI, AND J. SOLDANI, *From (incomplete) tosca specifications to running applications, with docker*, in Software Engineering and Formal Methods, A. Cerone and M. Roveri, eds., vol. 10729 of LNCS, Springer International Publishing, 2018, pp. 491–506.

[26] A. BROGI, D. NERI, L. RINALDI, AND J. SOLDANI, *Orchestrating incomplete TOSCA applications with Docker*, Science of Computer Programming, 166 (2018), pp. 194–213.

[27] A. BROGI, D. NERI, AND J. SOLDANI, *DockerFinder: Multi-attribute search of Docker images*, in 2017 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2017, pp. 273–278.

[28] ——, *A microservice-based architecture for (customisable) analyses of docker images*, Software: Practice and Experience, 48 (2018), pp. 1461–1474.

[29] ——, *Freshening the air in microservices: Resolving architectural smells via refactoring*, in Service-Oriented Computing - ICSOC 2019 Workshops, LNCS, Springer, 2019.

[30] A. BROGI, L. RINALDI, AND J. SOLDANI, *Tosker: A synergy between tosca and docker for orchestrating multicomponent applications*, Software: Practice and Experience, 48 (2018), pp. 2061–2079.

[31] ——, *TosKer: Orchestrating applications with TOSCA and Docker*, in Advances in Service-Oriented and Cloud Computing, Z. Á. Mann and V. Stolz, eds., vol. 824 of CCIS, Springer International Publishing, 2018, pp. 130–144.

[32] A. BROGI AND J. SOLDANI, *Matching cloud services with TOSCA*, in Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers, C. Canal and M. Villari, eds., Springer, 2013, pp. 218–232.

[33] ——, *Finding available services in TOSCA-compliant clouds*, Science of Computer Programming, 115 (2016), pp. 177 – 198.

[34] A. BROGI, J. SOLDANI, AND P. WANG, *TOSCA in a nutshell: Promises and perspectives*, in Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings, M. Villari, W. Zimmermann, and K.-K. Lau, eds., Springer Berlin Heidelberg, 2014, pp. 171–186.

[35] R. BUYYA, C. S. YEO, S. VENUGOPAL, J. BROBERG, AND I. BRANDIC, *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*, Future Generation Computer Systems, 25 (2009), pp. 599–616.

[36] R. CALHEIROS, R. RANJAN, A. BELOGLAZOV, C. A. F. DE ROSE, AND R. BUYYA, *Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms*, Software Practice and Experience, 41 (2011), pp. 23–50.

[37] M. CAMILLI, C. BELLETTINI, L. CAPRA, AND M. MONGA, *A formal framework for specifying and verifying microservices based process flows*, in Software Engineering and Formal Methods, A. Cerone and M. Roveri, eds., LNCS, Cham, 2018, Springer International Publishing, pp. 187–202.

[38] M. CARDARELLI, L. IOVINO, P. DI FRANCESCO, A. DI SALLE, I. MALAVOLTA, AND P. LAGO, *An extensible data-driven approach for evaluating the quality of microservice architectures*, in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, New York, NY, USA, 2019, ACM, pp. 1225–1234.

[39] C. CARNEIRO AND T. SCHMELMER, *Microservices From Day One: Build Robust and Scalable Software from the Start*, Apress, Berkeley, CA, 1st ed., 2016.

[40] J. CARNELL, *Spring Microservices in Action*, Manning Publications Co., 1st ed., 2017.

[41] A. CARRASCO, B. V. BLADEL, AND S. DEMEYER, *Migrating towards microservices: Migration and architecture smells*, in Proceedings of the 2Nd International Workshop on Refactoring, IWoR 2018, New York, NY, USA, 2018, ACM, pp. 1–6.

[42] J. CARRASCO, F. DURÁN, AND E. PIMENTEL, *Trans-cloud: Camp/tosca-based bidimensional cross-cloud*, Computer Standards & Interfaces, 58 (2018), pp. 167–179.

[43] M. CAVALLARI AND F. TORNIERI, *Information systems architecture and organization in the era of microservices*, in Network, Smart and Open, R. Lamboglia, A. Cardoni, R. P. Dameri, and D. Mancini, eds., vol. 24 of Lecture Notes in Information Systems and Organisation, Cham, 2018, Springer International Publishing, pp. 165–177.

[44] CESARE PAUTASSO , *Personal communication*, 2020.

[45] A. CHAUHAN, M. ALI BABAR, AND B. BENATALLAH, *Architecting cloud-enabled systems: A systematic survey of challenges and solutions*, Software Practice and Experience, 47 (2016).

[46] P. CHEKIN, *Multi-container pods and container communication in kubernetes*. The Mirantis Blog, 2017.

[47] K. COCHRANE, J. S. CHELLADHURAI, AND N. K. KHARE, *Docker Cookbook: Over 100 Practical and Insightful Recipes to Build Distributed Applications with Docker*, Packt Publishing, 2nd ed., 2018.

[48] A. COCKROFT, *Spigo*. `https://github.com/adrianco/spigo`.

[49] COREOS, *Clair*. `https://github.com/coreos/clair`.

[50] ——, *Rkt*. `https://coreos.com/rkt/`.

[51] R. DALL, *Performance patterns in microservices-based integrations*. DZone, `https://dzone.com/articles/performance-patterns-in-microservices-based-integr-1`, 2016.

[52] DATADOG, *Eight surprising facts about docker adoption*. `https://www.datadoghq.com/docker-adoption/`, 2017.

[53] R. DI COSMO, J. MAURO, S. ZACCHIROLI, AND G. ZAVATTARO, *Aeolus: A component model for the cloud*, Information and Computation, 239 (2014), pp. 100 – 121.

[54] P. DI FRANCESCO, P. LAGO, AND I. MALAVOLTA, *Migrating towards microservice architectures: An industrial survey*, in 2018 IEEE Int. Conf. on Software Architecture (ICSA), 2018, pp. 29–38.

[55] P. DI FRANCESCO, P. LAGO, AND I. MALAVOLTA, *Architecting with microservices: A systematic mapping study*, Journal of Systems and Software, 150 (2019), pp. 77 – 97.

[56] P. DI FRANCESCO, I. MALAVOLTA, AND P. LAGO, *Research on architecting microservices: Trends, focus, and potential for industrial adoption*, in 2017 IEEE Int. Conf. on Software Architecture (ICSA), 2017, pp. 21–30.

[57] B. DI MARTINO, D. PETCU, R. COSSU, P. GONCALVES, T. MÁHR, AND M. LOICHATE, *Building a mosaic of clouds*, in Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31–September 3, 2010, Revised Selected Papers, M. R. Guarracino, F. Vivien, J. L. Träff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Di Martino, and M. Alexander, eds., Springer Berlin Heidelberg, 2011.

[58] DOCKER INC., *Docker*. `https://www.docker.com/`.

[59] ——, *Docker Hub*. `https://hub.docker.com/`.

[60] ——, *Docker Hub hits 5 billion pulls*. `https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls`.

[61] ——, *Docker Store*. https://store.docker.com/.

[62] ——, *Docker Swarm*. https://docs.docker.com/swarm/.

[63] ——, *Overview of Docker Compose*. https://docs.docker.com/compose/.

[64] N. DRAGONI, S. GIALLORENZO, A. L. LAFUENTE, M. MAZZARA, F. MONTESI, R. MUSTAFIN, AND L. SAFINA, *Microservices: Yesterday, today, and tomorrow*, in Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[65] T. ERL, R. PUTTINI, AND Z. MAHMOOD, *Cloud Computing: Concepts, Technology & Architecture*, Prentice Hall Press, Upper Saddle River, NJ, USA, 1st ed., 2013.

[66] C. FEHLING, F. LEYMANN, R. RETTER, W. SCHUPECK, AND P. ARBITTER, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*, Springer, 2014.

[67] F. A. FONTANA, I. PIGAZZINI, R. ROVEDA, D. TAMBURRI, M. ZANONI, AND E. DI NITTO, *Arcan: A tool for architectural smells detection*, in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, 2017, pp. 282–285.

[68] W. FRAKES AND C. TERRY, *Software reuse: Metrics and models*, ACM Computing Surveys, 28 (1996), pp. 415–435.

[69] A. FURDA, C. FIDGE, O. ZIMMERMANN, W. KELLY, AND A. BARROS, *Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency*, IEEE Software, 35 (2018), pp. 63–72.

[70] J. GARCIA, D. POPESCU, G. EDWARDS, AND N. MEDVIDOVIC, *Identifying architectural bad smells*, in 13th Eur. Conf. on Software Maintenance and Reengineering, IEEE, 2009, pp. 255–258.

[71] V. GAROUSI, M. FELDERER, AND M. V. MÄNTYLÄ, *The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature*, in Proc. of the 20th Int. Conf. on Evaluation and Assessment in Software Engineering, EASE '16, ACM, 2016, pp. 26:1–26:6.

[72] N. GEHANI, *Want to develop great microservices? Reorganize your team*. TechBeacon, https://techbeacon.com/app-dev-testing/want-develop-great-microservices-reorganize-your-team, 2018.

[73] J. GHOFRANI AND D. LÜBKE, *Challenges of microservices architecture: A survey on the state of the practice*, in Proc. of the 10th Workshop on Services and their Composition (ZEUS 2018), CEUR-WS.org, 2018, pp. 1–8.

[74] B. GOLDEN, *5 fundamentals to a succesful microservice design*. TechBeacon, https://techbeacon.com/app-dev-testing/5-fundamentals-successful-microservice-design, 2017.

[75] ——, *Creating a microservice: Design first, code later*. TechBeacon, https://techbeacon.com/app-dev-testing/creating-microservice-design-first-code-later, 2018.

[76] G. GRANCHELLI, M. CARDARELLI, P. DI FRANCESCO, I. MALAVOLTA, L. IOVINO, AND A. DI SALLE, *MicroART: A software architecture recovery tool for maintaining microservice-based systems*, in 2017 IEEE Int. Conf. on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, 2017, pp. 298–302.

[77] C. GUIDI, I. LANESE, M. MAZZARA, AND F. MONTESI, *Microservices: A language-based approach*, in Present and Ulterior Software Engineering, Springer, 2017, pp. 217–225.

[78] R. GUIDOTTI, J. SOLDANI, D. NERI, AND A. BROGI, *Explaining successful docker images using pattern mining analysis*, in Software Technologies: Applications and Foundations, M. Mazzara, I. Ober, and G. Salaün, eds., Cham, 2018, Springer International Publishing, pp. 98–113.

[79] R. GUIDOTTI, J. SOLDANI, D. NERI, A. BROGI, AND D. PEDRESCHI, *Helping your Docker images to spread based on explainable models*, in Machine Learning and Knowledge Discovery in Databases, U. Brefeld, E. Curry, E. Daly, B. MacNamee, A. Marascu, F. Pinelli, M. Berlingerio, and N. Hurley, eds., Cham, 2019, Springer International Publishing, pp. 205–221.

[80] J. GUILLÉN, J. MIRANDA, J. M. MURILLO, AND C. CANAL, *A service-oriented framework for developing cross cloud migratable software*, Journal of Systems and Software, 86 (2013), pp. 2294–2308.

[81] M. HAMDAQA, T. LIVOGIANNIS, AND L. TAHVILDARI, *A reference model for developing cloud applications*, in CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, F. Leymann, I. Ivanov, M. van Sinderen, and B. Shishkov, eds., SciTePress, 2011.

[82] S. HASELBÖCK, R. WEINREICH, AND G. BUCHGEHER, *Decision models for microservices: Design areas, stakeholders, use cases, and requirements*, in Software Architecture, A. Lopes and R. de Lemos, eds., Cham, 2017, Springer International Publishing, pp. 155–170.

[83] S. HASSAN, N. ALI, AND R. BAHSOON, *Microservice ambients: An architectural meta-modelling approach for microservice granularity*, in 2017 Int. Conf. on Software Architecture, IEEE, 2017, pp. 1–10.

[84] S. HASSAN AND R. BAHSOON, *Microservices and their design trade-offs: A self-adaptive roadmap*, in 2016 Int. Conf. on Services Computing, IEEE, 2016, pp. 813–818.

[85] HEADWAY SOFTWARE TECHNOLOGIES, *Structure 101*. https://structure101.com.

[86] S. HENDRICKSON, S. STURDEVANT, T. HARTER, V. VENKATARAMANI, A. C. ARPACI-DUSSEAU, AND R. H. ARPACI-DUSSEAU, *Serverless computation with openlambda*, Elastic, 60 (2016), p. 80.

[87] HEROKU, *Dynos*. https://www.heroku.com/dynos.

[88] K. HIGHTOWER, B. BURNS, AND J. BEDA, *Kubernetes: Up & Running — Dive into the Future of Infrastructure*, O'Reilly Media, Inc., 1st ed., 2017.

[89] P. HIRMER, U. BREITENBÜCHER, T. BINZ, AND F. LEYMANN, *Automatic topology completion of TOSCA-based cloud applications*, in 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI), vol. 232, Lecture Notes in Informatics (LNI), 2014, pp. 247–258.

[90] G. HOHPE AND B. WOOLF, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Longman Publishing Co., Inc., 2003.

[91] K. INDRASIRI, *Microservices in practice: From architecture to deployment*. DZone, https://dzone.com/articles/microservices-in-practice-1, 2016.

[92] K. INDRASIRI AND P. SIRIWARDENA, *Microservices for the Enterprise: Designing, Developing, and Deploying*, Apress, Berkeley, CA, 1st ed., 2018.

[93] IRON.IO, *IronFunctions*. https://github.com/iron-io/functions.

[94] ISTIO AUTHORS, *Istio*. Istio, https://istio.io.

[95] P. JAMSHIDI, C. PAHL, N. MENDONÇA, J. LEWIS, AND S. TILKOV, *Microservices: The journey so far and challenges ahead*, IEEE Software, 35 (2018), pp. 24–35.

[96] P. JAMSHIDI, C. PAHL, AND N. C. MENDONÇA, *Pattern-based multi-cloud architecture migration*, Software: Practice and Experience, 47 (2017), pp. 1159–1184.

[97] JFROG LTD., *Docker: Secure Clustered HA Docker Registries With A Universal Artifact Repository*. https://www.jfrog.com/support-service/whitepapers/docker/.

[98] A. JOY, *Performance comparison between linux containers and virtual machines*, in 2015 International Conference on Advances in Computer Engineering and Applications, March 2015, pp. 342–346.

[99] M. KALSKE, N. MÄKITALO, AND T. MIKKONEN, *Challenges when moving from monolith to microservice architecture*, in Current Trends in Web Engineering, Springer, 2018, pp. 32–47.

[100] S. KEHRER AND W. BLOCHINGER, *TOSCA-based container orchestration on mesos*, SICS Software-Intensive Cyber-Physical Systems, (2018).

[101] H. KNOCHE AND W. HASSELBRING, *Using microservices for legacy software modernization*, IEEE Software, 35 (2018), pp. 44–49.

[102] Z. KOZHIRBAYEV AND R. O. SINNOTT, *A performance comparison of container-based technologies for the cloud*, Future Generation Computer Systems, 68 (2017), pp. 175–182.

[103] L. KRAUSE, *Microservices: Patterns and Applications*, Microservicesbook.io, 1st ed., 2015.

[104] P. KRUCHTEN, *The 4+1 view model of architecture*, IEEE Software, 12 (1995), pp. 42–50.

[105] KUBELESS, *Kubeless: A Bitnami project*. `https://github.com/kubeless/kubeless`.

[106] P. A. LAPLANTE, *What every engineer should know about software engineering*, CRC Press, 2007.

[107] LEARNBOOST, *Mongoose*. `http://mongoosejs.com/`.

[108] J. LEWIS AND M. FOWLER, *Microservices: A definition of this new architectural term*. ThoughtWorks, `https://www.martinfowler.com/articles/microservices.html`, 2014.

[109] F. LEYMANN, *Cloud computing*, it — Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik, 53 (2011), pp. 163–164.

[110] Z. LI, M. KIHL, Q. LU, AND J. A. ANDERSSON, *Performance overhead comparison between hypervisor and container based virtualization*, in 31st IEEE International Conference on Advanced Information Networking and Applications, AINA 2017, Taipei, Taiwan, March 27-29, 2017, 2017, pp. 955–962.

[111] J. LONG, *The power, patterns, and pains of microservices*. DZone, `https://dzone.com/articles/the-power-patterns-and-pains-of-microservices`, 2015.

[112] S. MA, C. FAN, Y. CHUANG, W. LEE, S. LEE, AND N. HSUEH, *Using service dependency graph to analyze and test microservices*, in 42nd Annual Computer Software and Applications Conf., vol. 02, IEEE, 2018, pp. 81–86.

[113] P. MAHLEN, *Modelling microservices at Spotify*. jFokus Developer Conf., 2016.

[114] K. MATTHIAS AND S. P. KANE, *Docker: Up and Running*, O'Reilly Media, 2015.

[115] C. MELÉNDEZ, *7 container design patterns you need to know*. TechBeacon, `https://techbeacon.com/enterprise-it/7-container-design-patterns-you-need-know`, 2018.

[116] MICROSCALING SYSTEM., *Image layers*. https://imagelayers.io/.

[117] ——, *Microbadger*. https://microbadger.com/.

[118] M. MITCHELL, J. OLDHAM, AND A. SAMUEL, *Advanced Linux Programming*, Sams Publishing, June 2001.

[119] I. NADAREISHVILI, R. MITRA, M. MCLARTY, AND M. AMUNDSEN, *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly Media, Inc., 1st ed., 2016.

[120] D. NERI, J. SOLDANI, O. ZIMMERMANN, AND A. BROGI, *Design principles, architectural smells and refactorings for microservices: a multivocal review*, SICS Software-Intensive Cyber-Physical Systems, (2019).

[121] S. NEWMAN, *Building Microservices*, O'Reilly Media, Inc., 1st ed., 2015.

[122] NODE.JS FOUNDATION, *Express*. http://expressjs.com/.

[123] M. NYGARD, *Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2nd ed., 2018.

[124] OASIS, *Topology and orchestration specification for cloud applications (tosca)*, 2013. version: 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf.

[125] ——, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer*. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf, 2013.

[126] OPENSTACK DEVELOPMENT COMMUNITY, *Parser for TOSCA simple profile in YAML*. https://github.com/openstack/tosca-parser.

[127] C. PAHL, A. BROGI, J. SOLDANI, AND P. JAMSHIDI, *Cloud container technologies: A state-of-the-art review*, IEEE Transactions on Cloud Computing, 7 (2019), pp. 677–692.

[128] C. PAHL AND P. JAMSHIDI, *Microservices: A systematic mapping study*, in CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science, SciTePress, 2016, pp. 137–146.

[129] C. PAHL AND B. LEE, *Containers and clusters for edge cloud architectures – a technology review*, in 2015 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 379–386.

[130] C. PAUTASSO, O. ZIMMERMANN, M. AMUNDSEN, J. LEWIS, AND N. JOSUTTIS, *Microservices in practice, part 1: Reality check and service design*, IEEE Software, 34 (2017), pp. 91–98.

[131] ——, *Microservices in practice, part 2: Service integration and sustainability*, IEEE Software, 34 (2017), pp. 97–104.

[132] K. PETERSEN, R. FELDT, S. MUJTABA, AND M. MATTSSON, *Systematic mapping studies in software engineering*, in Proc. of the 12th Int. Conf. on Evaluation and Assessment in Software Engineering, EASE'08, BCS Learning & Development Ltd., 2008, pp. 68–77.

[133] PIKA, *Introduction to pika*. `https://pika.readthedocs.io`.

[134] S. F. PIRAGHAJ, A. V. DASTJERDI, R. N. CALHEIROS, AND R. BUYYA, *ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers*, Software Practice and Experience, 47 (2017), pp. 505–521.

[135] PYTHON SOFTWARE FOUNDATION, *Requests*. `https://github.com/psf/requests`.

[136] M. RAHMAN, S. PANICHELLA, AND D. TAIBI, *A curated dataset of microservices-based systems*, Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution, (2019).

[137] M. RICHARDS, *Microservices Antipatterns and Pitfalls*, O'Reilly Media, Inc., 1st ed., 2016.

[138] C. RICHARDSON, *Microservices: Decomposing applications for deployability and scalability*. InfoQ, `https://www.infoq.com/articles/microservices-intro`, 2014.

[139] ——, *Microservices patterns*, Manning publications, 1st ed., 2018.

[140] M. ROBERTS., *Serverless Architectures*. `https://martinfowler.com/articles/serverless.html`.

[141] B. RUECKER, *3 common pitfalls of microservices integration and how to avoid them*. InfoWorld, `https://www.infoworld.com/article/3254777/3-common-pitfalls-of-microservices-integrationand-how-to-avoid-them.html`, 2018.

[142] T. SALEH, *Microservices antipatterns*. InfoQ, `https://www.infoq.com/presentations/cloud-anti-patterns`, 2016.

[143] A. SANCHEZ, L. S. BARBOSA, AND A. MADEIRA, *Modelling and verifying smell-free architectures with the archery language*, in Software Engineering and Formal Methods, Springer, 2015, pp. 147–163.

[144] D. SAVCHENKO, G. RADCHENKO, AND O. TAIPALE, *Microservices validation: Mjolnirr platform case study*, in 2015 38th Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015, pp. 235–240.

[145] R. SMITH, *Docker Orchestration*, Packt Publishing, 2017.

[146] J. SOLDANI, T. BINZ, U. BREITENBÜCHER, F. LEYMANN, AND A. BROGI, *ToscaMart: A method for adapting and reusing cloud applications*, Journal of Systems and Software, 113 (2016), pp. 395–406.

[147] J. SOLDANI, D. A. TAMBURRI, AND W. J. VAN DEN HEUVEL, *The pains and gains of microservices: A systematic grey literature review*, Journal of Systems and Software, 146 (2018), pp. 215 – 232.

[148] S. SOLTESZ, H. PÖTZL, M. E. FIUCZYNSKI, A. BAVIER, AND L. PETERSON, *Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors*, SIGOPS Oper. Syst. Rev., 41 (2007), pp. 275–287.

[149] J. SPILLNER, *Snafu: Function-as-a-Service (FaaS) runtime design and implementation*, CoRR, abs/1703.07562 (2017).

[150] M. STOCKER, O. ZIMMERMANN, D. LÜBKE, U. ZDUN, AND C. PAUTASSO, *Interface quality patterns – communicating and improving the quality of microservices APIs*, in 23rd European Conference on Pattern Languages of Programs 2018, July 2018.

[151] D. TAIBI AND V. LENARDUZZI, *On the definition of microservice bad smells*, IEEE Software, 35 (2018), pp. 56–62.

[152] D. TAIBI, V. LENARDUZZI, AND C. PAHL, *Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation*, IEEE Cloud Computing, 4 (2017), pp. 22–32.

[153] ——, *Architectural patterns for microservices: A systematic mapping study*, in Proc. of the 8th Int. Conf. on Cloud Computing and Services Science - Volume 1: CLOSER,, SciTePress, 2018, pp. 221–232.

[154] ——, *Microservices anti-patterns: A taxonomy*, in Microservices - Science and Engineering, A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, eds., Springer, 2019. *[In press]*.

[155] B. TERZIĆ, V. DIMITRIESKI, S. KORDIfÁ, G. MILOSAVLJEVIfÁ, AND I. LUKOVIfÁ, *Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures*, Enterprise Information Systems, 12 (2018), pp. 1034–1057.

[156] J. TESSIER, *DependencyFinder*. `https://github.com/jeantessier/dependency-finder`.

[157] THE LINUX FOUNDATION., *Kubernetes*. `https://kubernetes.io/`.

[158] C. TOZZI, *Enterprise Docker*, O'Reilly Media, Inc., 1 ed., 2017.

[159] S. VIDAL, H. VAZQUEZ, J. A. DIAZ-PACE, C. MARCOS, A. GARCIA, AND W. OIZUMI, *JSpIRIT: A flexible tool for the analysis of code smells*, in 34th Int. Conf. of the Chilean Computer Science Society, IEEE, 2015, pp. 1–6.

[160] WEAVEWORKS INC. AND CONTAINER SOLUTIONS INC., *Sock Shop*. `https://microservices-demo.github.io/`.

[161] J. WETTINGER, V. ANDRIKOPOULOS, AND F. LEYMANN, *Automated capturing and systematic usage of devops knowledge for cloud applications*, in Proceedings of the 2015 IEEE International Conference on Cloud Engineering, IC2E '15, Washington, DC, USA, 2015, IEEE Computer Society, pp. 60–65.

[162] C. WOHLIN, P. RUNESON, M. HÖST, M. C. OHLSSON, B. REGNELL, AND A. WESSLÉN, *Experimentation in software engineering: an introduction*, Kluwer Academic Publishers, 2000.

[163] E. WOLFF, *Microservices: Flexible Software Architecture*, Addison-Wesley Professional, 1st ed., 2016.

[164] M. WURSTER, U. BREITENBÜCHER, A. BROGI, G. BROGI, L. HARZENETTER, F. LEYMANN, J. SOLDANI, AND V. YUSSUPOV, *The EDMM modeling and transformation system*, in 17th International Conference on Service-Oriented Computing (ICSOC 2019), 2019. *[In press]*.

[165] M. WURSTER, U. BREITENBÜCHER, A. BROGI, G. FALAZI, L. HARZENETTER, F. LEYMANN, J. SOLDANI, AND V. YUSSUPOV, *The edmm modeling and transformation system*, (2019).

[166] M. WURSTER, U. BREITENBÜCHER, M. FALKENTAL, C. KRIEGER, F. LEYMANN, K. SAATKAMP, AND J. SOLDANI, *The essential deployment metamodel: a systematic review of deployment automation technologies*, SICS Software-Inensive Cyber-Physical Systems, (2019).

[167] H. ZENG, B. WANG, W. DENG, AND W. ZHANG, *Measurement and evaluation for docker container networking*, in 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), IEEE, 2017, pp. 105–108.

[168] O. ZIMMERMANN, *Architectural refactoring for the cloud: a decision-centric view on cloud migration*, Computing, 99 (2017), pp. 129–145.

[169] ——, *Microservices tenets*, Computere Science: Research and Development, 32 (2017), pp. 301–310.

[170] O. ZIMMERMANN, M. STOCKER, D. LÜBKE, AND U. ZDUN, *Interface representation patterns - crafting and consuming message-based remote apis*, in 22nd European Conference on Pattern Languages of Programs (EuroPLoP 2017), July 2017, pp. 1–36.

All links were last followed on the 30th of October 2019.

APPENDIX A

EVALUATING OUR ANALYSIS AND REFACTORING METHODOLOGY

In Chapter 3 we presented a methodology for automatically detecting the architectural smells affecting a microservice-based applicatoions and for applying architectural refactorings in order to resolve identified smells. We also illustrated the feasibility of our approach by presenting its prototype implementation, i.e., μFRESHENER, in Sect. 3.4. To further assess its potential, we have also run a case study and a controlled experiment, which we present hereafter.

A.1 Case study

We applied our methodology to a case study based on a real-world application developed and maintained by an italian IT company we are cooperating with. The considered application is a platform involving 21 components, whose purpose is to allow to monitor, manage and configure a smart factory. As a result, we were able to identify five architectural smells affecting the considered application, which have been resolved by the company in accordance to what we presented in Sect. 3.3.

Fig. A.1 illustrates the 12 services, 7 data stores and 2 message brokers composing the considered platform, anonymised for privacy reasons. The figure also illustrates the interactions between them, with service-to-service interactions being such that the endpoint of the target of the interaction is dynamically resolved by exploiting a service discovery, and that timeouts are exploited to enhance the fault resilience of the source of the interaction.

Even if the considered topology is small, the amount of components and interactions makes it not easy to manually identify all occurrences of architectural smells. We hence modelled the application topology with μFRESHENER, and this allowed to identify the five instances of architectural smells affecting the considered topology, i.e., four instances of the *no API gateway*

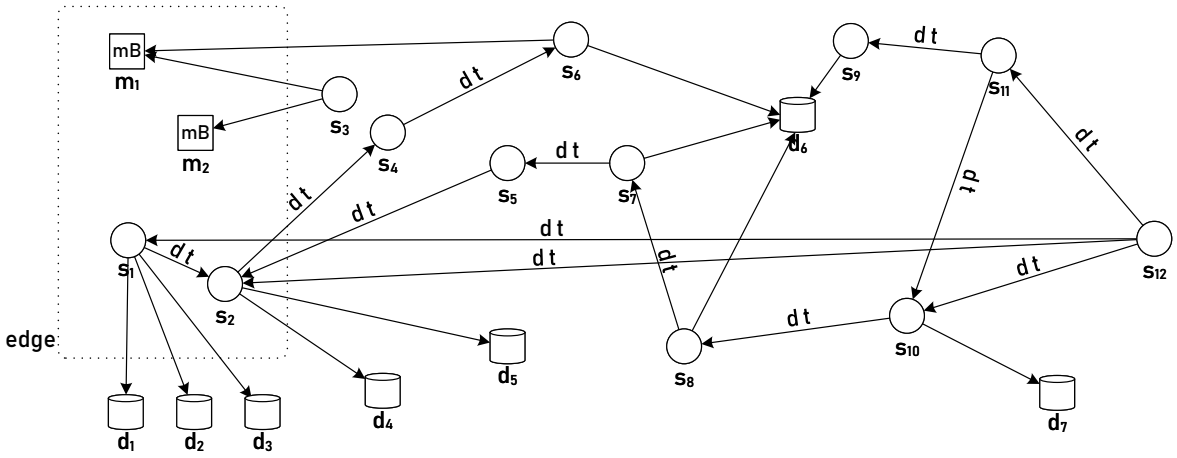


Figure A.1: Anonymised architecture of the considered application. Labels on relationships indicate which properties are true (with the other being false).

smell (regarding m_1, m_2, s_1 and s_2), and one instance of the *shared persistence* smell (due to services s_6, s_7, s_8 and s_9 all accessing the same data store d_6).[1]

The identified architectural smells were to be refactored, and solutions were found by exploiting μFRESHENER. The company indeed exploited μFRESHENER to explore the possible architectural refactorings, and it decided which refactoring to apply based on its business requirements and on the actual cost for implementing an architectural refactoring (i.e., for refactoring the application sources by following the guidelines given by an architectural refactoring). The resulting, refactored architecture is in Fig. A.2.

A message router g_1 was first introduced for resolving the no API gateway smell indicated by m_1. Then, given that the external clients placing messages in m_1 and m_2 were the same (i.e., smart production machines sending monitored data to the platform), the gateway g_1 was exploited to resolve also the no API gateway smell indicated m_2. Similarly, since s_1 and s_2 were services accessed by the same clients, a message router g_2 was introduced for managing the access to s_1 and s_2 from outside of the platform.

The shared persistence smell due to services s_6, s_7, s_8 and s_9 all accessing the same data store d_6 was instead resolved by introducing a novel service s_{13}, acting as data manager for d_6. Services s_6, s_7, s_8 and s_9 were then directly interacting with s_{13} to access the data in d_6, and this resulted in adding novel architectural smells (as each newly introduced service-to-service interaction was endpoint-based and wobbly). To resolve such smells, and similarly to the other service-to-service interactions in the considered topology, the newly introduced interactions were refactored in such a way that the endpoint of the target of the interaction was dynamically resolved and that proper timeouts were set.

[1]The TOSCA specifications of both the initial and the refactored topologies are publicly available at `https://github.com/di-unipi-socc/microFreshener-core/tree/master/data/examples/case-study`.

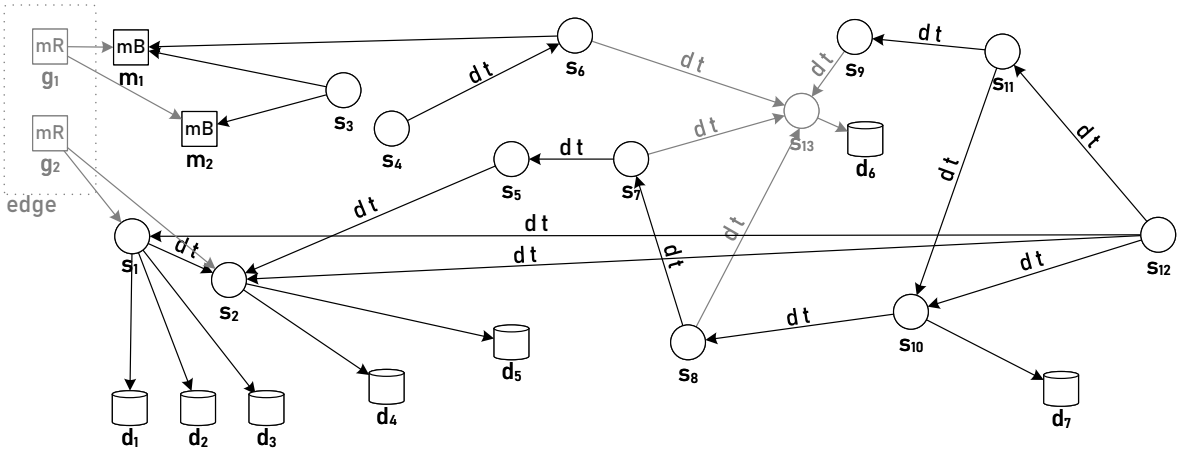


Figure A.2: Refactoring of the architecture in Fig. A.1. Labels on relationships indicate which properties are true (with the other being false). Updates due to refactorings are in grey.

A.2 Controlled experiment

We have also run a small controlled experiment, whose aim was to provide a first quantitative evaluation of how much our approach can be of help in identifying and resolving the architectural smells affecting a microservice-based application.

The experiment was requiring participants to perform two main tasks, namely:

(t_1) discovering all the architectural smells *initially* affecting an existing, third-party microservice-based application, and

(t_2) identifying a sequence of architectural refactorings resolving all the architectural smells affecting an application (i.e., making it "smell-free").

Both t_1 and t_2 were intended to be run with and without the support provided by μFRESHENER, on two existing, third-party applications, i.e., *FTGO*[2] and *Sock Shop*[3].

We submitted the tasks t_1 and t_2 to 24 volounteers students of the MSc in Computer Science (and all holding a BSc in Computer Science) and knowing the basics of microservices. The volounteers were partitioned into two groups, i.e., A and B, and each group was working in two rounds, with each round lasting in 45 minutes.

Round 1 → Group A was working on *Sock Shop* and group B was working on *FTGO*. Both groups were working *without* μFRESHENER.

Round 2 → Group A was working on *FTGO* and group B was working on *Sock Shop*. Both groups were allowed to exploit the support offered by μFRESHENER.

[2] `https://github.com/microservices-patterns/ftgo-application`.
[3] `https://github.com/microservices-demo/`

In this way, group A was acting as control group for group B, and vice versa.[4]

The goal of the experiment was to compare the success rate for solving tasks t_1 and t_2 first without using μFRESHENER and then by exploiting the support it provides. The obtained success rates are reported in Fig. A.1, which shows that the group exploiting the support provided by μFRESHENER was outperforming with respect to that not using it. Indeed, in both cases, the group

	identified smells		**resolving all smells**		**remaining smells**	
	average perc.	*stand. dev.*	*number of particip.*	*perc. of particip.*	*average count*	*stand. dev.*
without μFRESHENER	94.23%	15.77%	6	50.00%	3.7	4.2
with μFRESHENER	100.00%	0	12	100.00%	0.0	0.0

(a)

	identified smells		**resolving all smells**		**remaining smells**	
	average perc.	*stand. dev.*	*number of particip.*	*perc. of particip.*	*average count*	*stand. dev.*
without μFRESHENER	48.96%	32.29%	1	8.33%	9.8	4.7
with μFRESHENER	100.00%	0	10	83.33%	0.2	0.4

(b)

TABLE A.1. Results of the controlled experiments on (a) FTGO and (b) Sock Shop.

exploiting μFRESHENER was capable of identifying all architectural smells initially affecting the considered application, while the same does not hold for the group not using μFRESHENER. In the case of *Sock Shop*, not even a half of the smells initially affecting the application were recognised by the group manually analysing it.

μFRESHENER was also definitely of help for resolving the identified smells and obtaining "smell-free" applications. Almost all participants exploiting μFRESHENER were able to identify and resolve the architectural smells affecting the considered application, with only two participants not being able to resolve 1 of the smells affecting *Sock Shop*. Instead, only half of the participants working on *FTGO* without μFRESHENER were capable of resolving the architectural smells affecting it, and only 1 out of 12 participants was capable of doing the same for *Sock Shop*. In those cases, the average number of remaining smells were indeed 3.7 for *FTGO* (with a standard deviation of 4.2) and 9.8 for *Sock Shop* (with a standard deviation of 4.2).

The small experiment (24 participants at work on 2 applications, each involving 13 or 14 components) that we conducted clearly indicates that the group using μFRESHENER completed the tasks with a higher success rate, even if the time for analysing and resolving architectural smells was limited to 45-minutes. This was more evident in the case of *Sock Shop* which includes a higher amount of interactions among microservices.

[4]A copy of the form submitted to the volounteers participating in the study is publicly available at `https://github.com/di-unipi-socc/microFreshener/blob/master/docs/the-microFreshener-experiment.pdf`.