Orchestrating incomplete TOSCA applications with Docker

Antonio Brogi, Davide Neri, Luca Rinaldi, Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract

Cloud applications typically integrate multiple components, each needing a virtualised runtime environment that provides the required software support (e.g., operating system, libraries). This paper shows how TOSCA and Docker can effectively support the orchestration of multi-component applications, even when their runtime specification is incomplete. More precisely, we first introduce a TOSCA-based representation of multi-component applications, and we illustrate how such representation can be exploited to specify only the application-specific components. We then present TosKeriser, a tool for automatically completing TOSCA application specifications, which can automatically discover the Docker-based runtime environments that provide the software support needed by the application components. We also show how we fruitfully exploited TosKeriser in two concrete case studies. Finally, we discuss how the specifications completed by TosKeriser can be automatically orchestrated by already existing TOSCA engines.

Keywords: cloud applications, TOSCA, Docker, container reuse

1. Introduction

Cloud computing permits running on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [2]. 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 [11].

At the same time, the need for orchestrating the management of multi-component applications across heterogeneous cloud platforms has emerged [4, 17]. The deployment, configuration, enactment 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 [13].

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 [19]. As we discuss in Sect. 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.

Published article: Brogi A, Neri D, Rinaldi L, Soldani J. "Orchestrating incomplete TOSCA applications with Docker", Science of Computer Programming, vol. 166, pp. 194-213, 2018.
DOI: 10.1016/j.scico.2018.07.005

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 [3]. 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 paper, we present a solution geared towards providing such an enhanced support. Our solution is based on TOSCA [22], the OASIS standard for orchestrating cloud applications, and on Docker, the de-facto standard for cloud container virtualisation [24]. The main contributions of this paper 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 [7], for instance). Such engines will automatically orchestrate the deployment and management of the corresponding applications on top of the given runtime environments.

This paper extends [5] by (a) extending the approach of [5] 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 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 paper is organised as follows. Sect. 2 illustrates an example further motivating the need for an enhanced support for orchestrating the management of cloud applications. Sect. 3 provides some background on TOSCA and Docker. Sect. 4 shows how to specify application-specific components only, with TOSCA. Sect. 5 then presents our tool to automatically determine appropriate Docker-based environments for hosting the components of an application. Sect. 6 illustrates the two case studies, while Sects. 7 and 8 discuss related work and draw some concluding remarks, respectively.

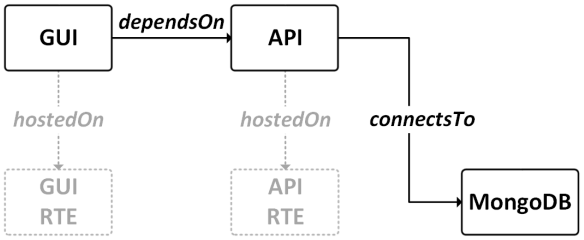

Figure 1: Running example: The application *Thinking*.

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. 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).

Docker containers work as virtualised environments for running application components [24]. 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 of all 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. 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

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

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. 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 paper, we show a TOSCA-based solution geared towards providing such a support.

3. Background

3.1. TOSCA

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [22]) 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 paper[3].

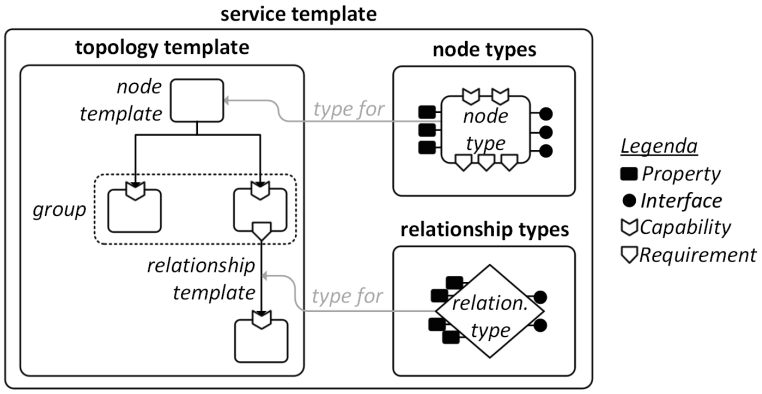

Figure 2: The TOSCA metamodel [22].

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

[3] A more detailed, self-contained introduction to TOSCA can be found in [3, 10].

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.

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 [27] 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. Specifying applications only, with TOSCA

Multi-component applications typically integrate various and heterogeneous software components [13]. We hereby propose a TOSCA-based representation for such components (Sect. 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. 4.2).

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. 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 [18]).

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).

[4] A more detailed introduction to Docker can be found in [18, 25].

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

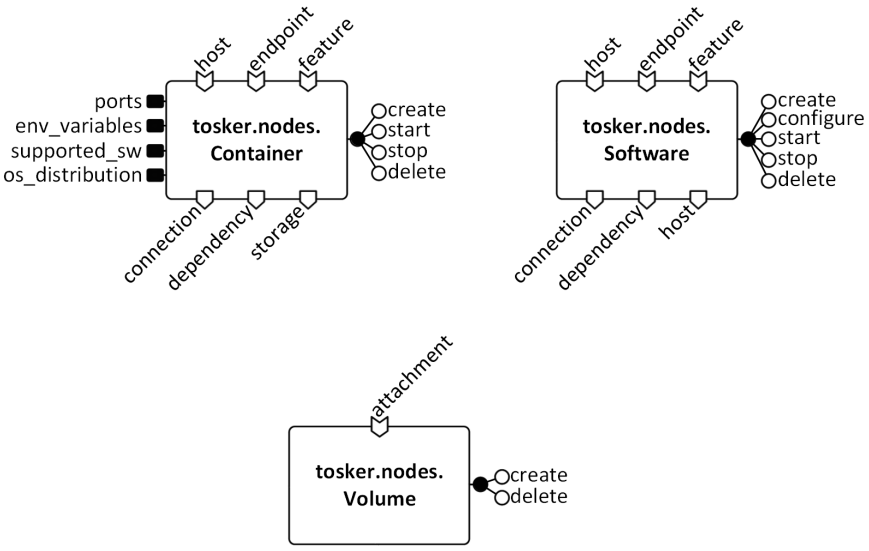

Figure 3: TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

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 [18]).

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 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 [22]. Namely, *tosca.relationships.AttachesTo* can be used to attach a Docker volume to a Docker container, *tosca.relationships.ConnectsTo* can indicate interconnections

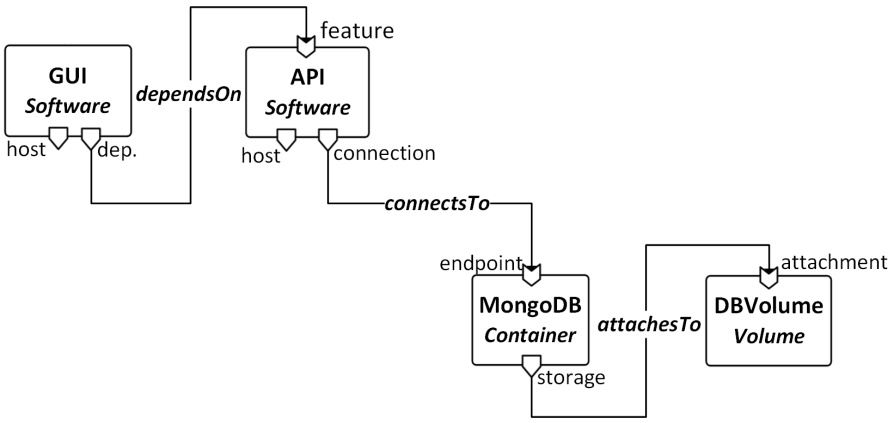

Figure 4: A specification of our running example in TOSCA (where nodes are typed with *tosker.nodes.Container*, *tosker.nodes.Volume*, or *tosker.nodes.Software*, while relationships are typed with TOSCA normative types [22]).

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].

4.2. 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 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 1. Consider again the application *Thinking* in our motivating scenario (Sect. 2). The components specific to *Thinking* (viz., *MongoDB*, *API*, and *GUI*) can be specified in TOSCA as illustrated in Fig. 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).

[6]The TOSCA specification [22] explains how to validly instantiate normative relationship types.

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

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. 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.). □

TOSCA natively supports the possibility of expressing constraints on the nodes that can satisfy requirements left pending [22], 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 2. Consider again the multi-component application *Thinking*, modelled in TOSCA as in Fig. 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. 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. 5.(b). The latter prescribes that *GUI* must run on a Docker 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]. □

4.3. 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 [22]. 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. 4.1, 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 3. Consider again the application *Thinking* in our motivating scenario (Sect. 2). Example 1 showed how to specify the components forming such application in TOSCA, while Example 2 illustrated how to indicate constraints on the Docker containers that can effectively run its software components *API* and *GUI*.

[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).

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

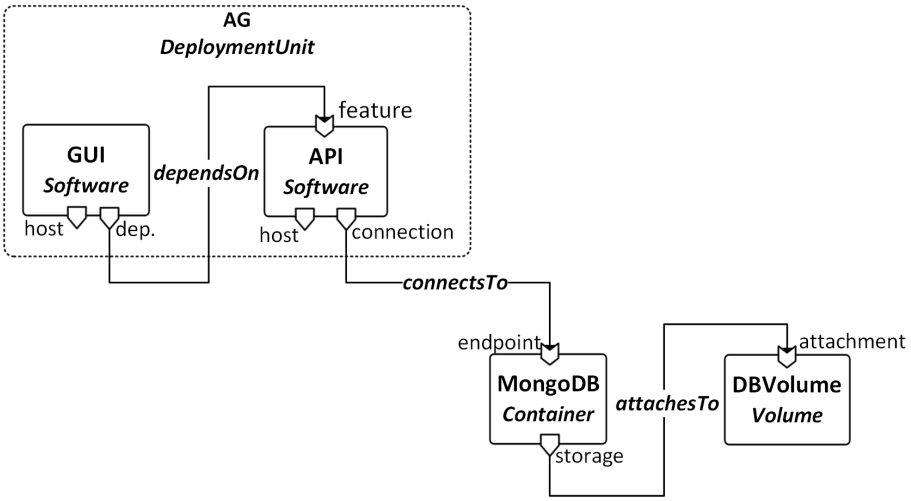

Figure 6: A specification of our running example in TOSCA, including a group (of type *tosker.groups.DeploymentUnit*) specifying that *API* and *GUI* must be hosted by the same Docker container.

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. 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. 5). □

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. 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 containers providing the support needed by such components. TosKeriser then completes the application topology by properly including the discovered

[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 [6] 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.

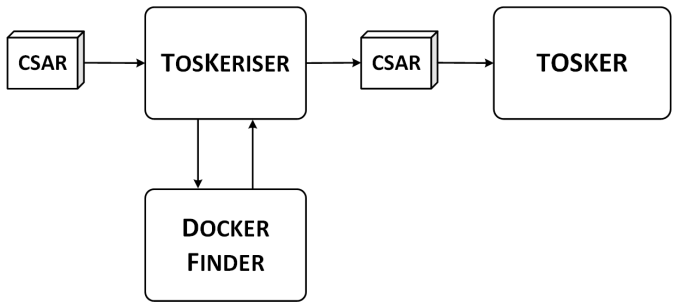

Figure 7: Open-source toolchain for orchestrating multi-component applications with TOSCA and Docker.

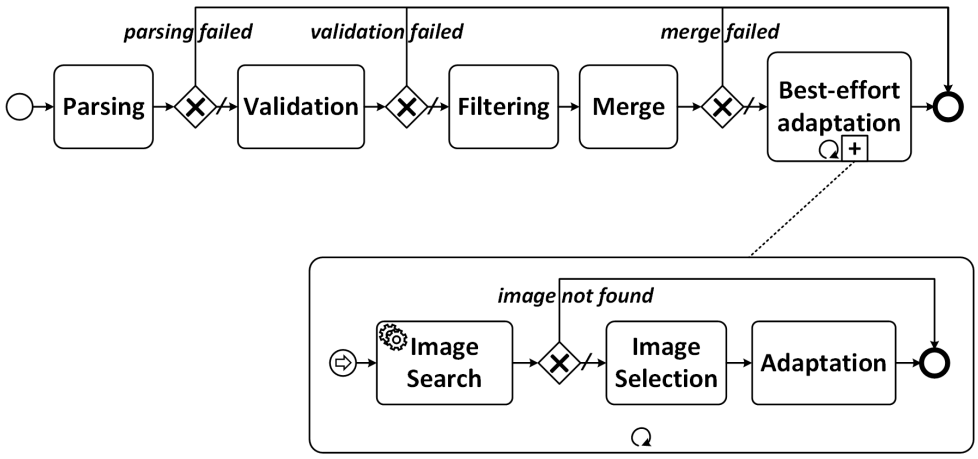

Figure 8: BPMN modelling of the process that TosKeriser performs to automatically complete TOSCA application specifications.

containers, and it outputs the resulting CSAR file. Such file can then be passed to TosKer [7] (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. 5.1), and we then show how to use TosKeriser in practice (Sect. 5.2).

5.1. How TosKeriser *completes applications, concretely*

TosKeriser completes TOSCA application specifications according to the workflow illustrated in Fig. 8.

Parsing. TosKeriser initially parses and validates the TOSCA application specification contained in the CSAR given as input. More precisely, the step *Parsing* first exploits the OpenStack TOSCA parser library [23] 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. 4.3.

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*.

Merge. For each pair $\langle nodes, conds \rangle$, 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 $\langle nodes, conds \rangle$, 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

[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.

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 $\langle nodes, mergedConds \rangle$, 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 $\langle nodes, mergedConds \rangle$ (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 $\langle nodes, mergedConds \rangle$, 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 $\langle nodes, mergedConds \rangle$ the following three sub-steps (Fig. 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 subprocess 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. 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 [7]), 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.

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 specification. In other words, it instructs TosKeriser to ignore the condition f_3 during the step *Filter* (see Sect. 5.1).

Example 4. Consider again the application *Thinking* in our motivating scenario, whose corresponding TOSCA representation is displayed in Fig. 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. 9, lighter node). Such node provides the software support and the port mappings needed

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

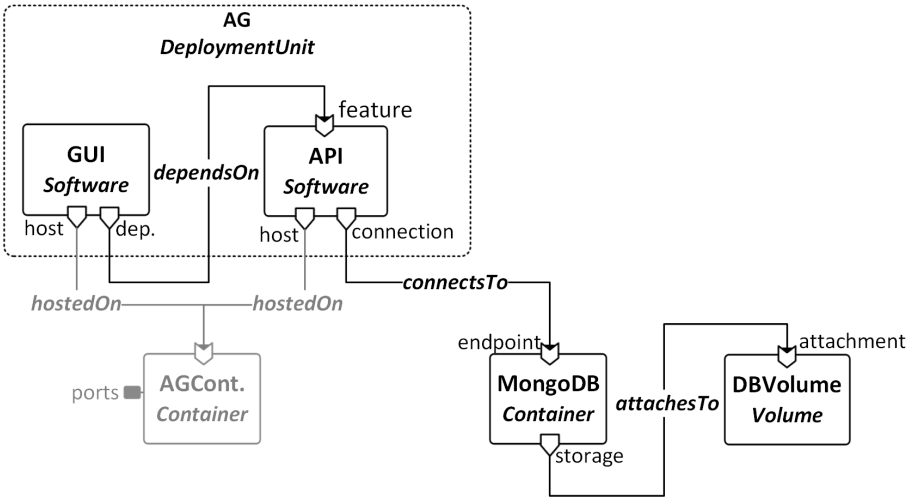

Figure 9: Application topology obtained by completing the partial topology of the application *Thinking* (Fig. 4). Lighter nodes and relationships are those automatically included by TosKeriser.

by both *API* and *GUI*. We can then run such file with TosKer [7] (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 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. □

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. 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

[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`.

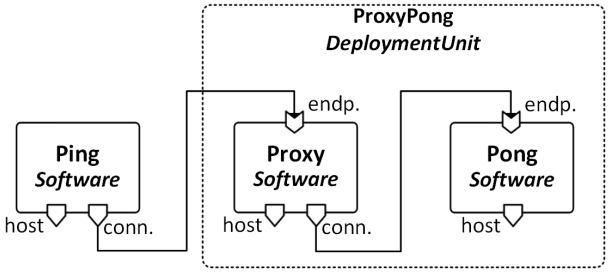

Figure 10: A specification of the topology of the *PingPong* application in TOSCA (where all relationships are of type *tosca.relationships.ConnectsTo*).

(Sect. 6.2). We finally present an example illustrating the usefulness of using groups in such case studies (Sect. 6.3).

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 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. 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

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

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 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.

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 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-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.

6.2. Maintenance of a third-party, existing application

Sock Shop [28] 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.

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

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 2: Technical requirements of the main services in *Sock Shop*.

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 2.

The *Sock Shop* application is then completed by three other components, namely *Edge Router*, *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. 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 2). We also developed 25 shell scripts for

[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`.

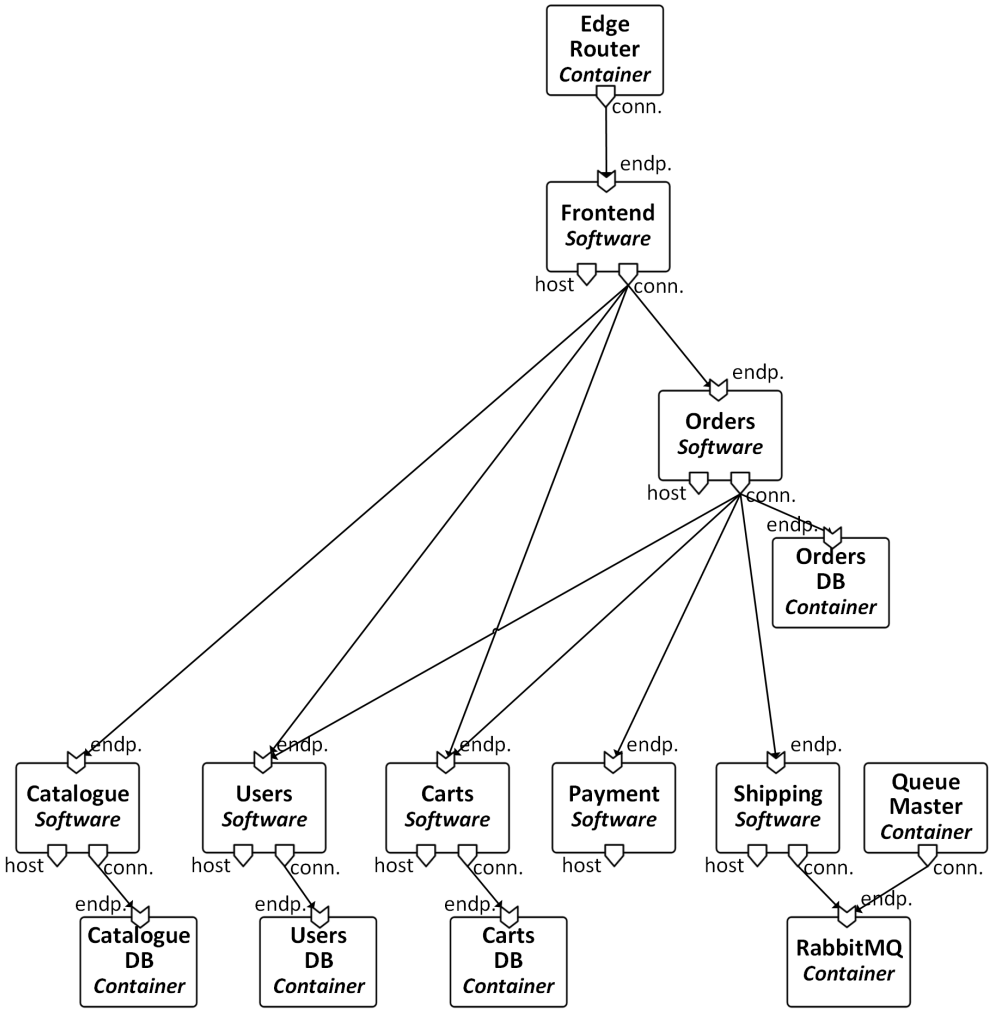

Figure 11: A specification of the topology of *Sock Shop* in TOSCA (where all relationships are of type *tosca.relationships.ConnectsTo*).

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 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 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

[20]The management operations of a component have to be implemented by an associated artifact only when the component actually needs such operations [22]. For instance, as *Users* does not require to be configured, we do not need to develop a script for implementing its management operation *configure*.

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

Table 3: Values of the considered KPIs for the the initial deployment of *Sock Shop* with TosKeriser, and for its already existing Docker-based deployment.

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: 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.

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 provided by the container hosting it, which is a frequent issue while maintaining in multi-component applications [20]. 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.

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 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.

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*. 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 5). This is even more evident if we compare the lines and files changed with those of the initial specification (Table 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.

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 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 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

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 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.

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	341
	a:7,c:7,d:0	*a:215,c:4,d:112*
Files	1	9
	a:0,c:1,d:0	*a:2,c:2,d:5*
Languages	1	3
	TOSCA	*Dockerfile,Docker Compose,bash*

Table 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.

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 [29], 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 50 times a test executing 100 "ping" requests, and

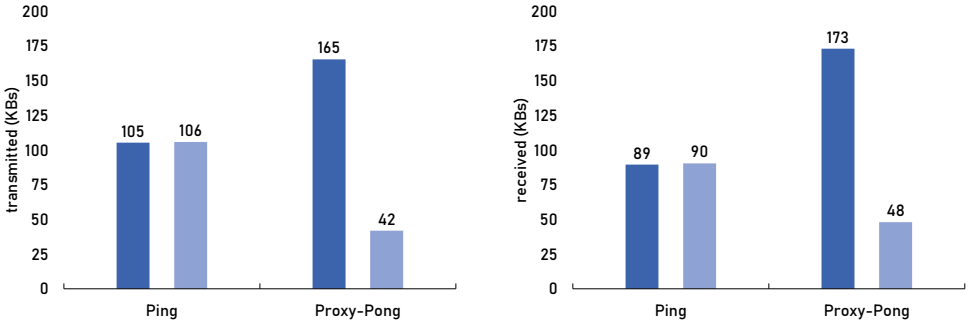

Figure 12: Average network traffic (a) transmitted and (b) received by the components of *PingPong* for processing 100 "ping" requests. Darker histograms plot the values for a deployment of *PingPong* with each component in a separate container, while lighter histograms plot the values for a deployment of *PingPong* with the components *Proxy* and *Pong* hosted in the same container.

we measured the network traffic generated by the containers running the components of *PingPong* for each iteration.

Fig. 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. 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 8 shows the average network traffic transmitted and received by the containers 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. 13).

[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 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*.

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.

7. Related work

We presented a solution for automatically completing TOSCA specifications, which is much in the spirit of [16]. The goal of [16] 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 [16] 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 [16] 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 [8], [9] and [26], 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 [8], [9] and [26] 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

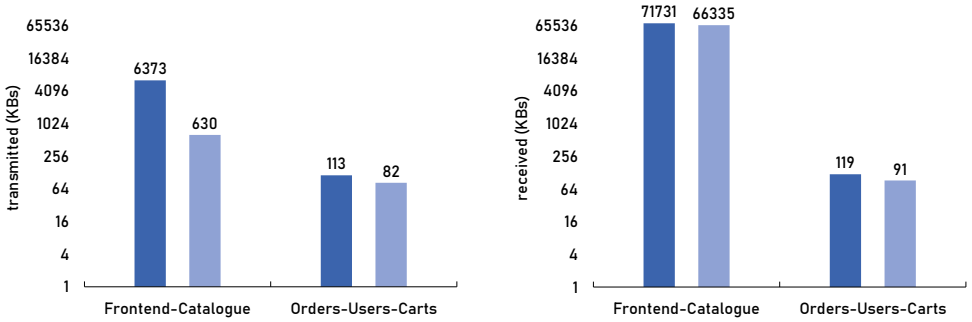

Figure 13: Average network traffic (a) transmitted and (b) received by the containers running the grouped components of *Sock Shop* executing its end-to-end test. Darker histograms plot the values for a deployment of *Sock Shop* with each component in a separate container, while lighter histograms plot 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.

TOSCA application specifications. [8], [9] and [26] 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, [12] and [15] 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.

[14] proposes a framework allowing developers to write the source code of cloud applications as if they were "on-premise" applications. [14] 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. [14] 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 [7]).

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 [22] 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 [3, 21]. 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 [24]).

8. Conclusions

Cloud applications typically consist of multiple heterogeneous components, whose deployment, configuration, enactment and termination must be suitably orchestrated [13]. 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 paper, 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 [7], which can process specifications produced by TosKeriser, to automatically orchestrate the deployment and management of the corresponding applications.

TosKeriser is integrated with DockerFinder [6], and it produces specifications that can be effectively processed by TosKer [7]. 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 [26]) 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 [1], 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.

References

[1] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the fly deployment optimization using smt and cp technologies. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications: Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, pages 229–245. Springer International Publishing, 2016.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Springer, New York, NY, 2014.

[4] Antonio Brogi, José Carrasco, Javier Cubo, Francesco D'Andria, Ahmad Ibrahim, Ernesto Pimentel, and Jacopo 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 - Volume 1: MultiCloud, (CLOSER 2014)*, pages 758–763. SciTePress, 2014.

[5] Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. From (incomplete) TOSCA specifications to running applications, with Docker. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods*, volume 10729 of *Lecture Notes in Computer Science*, pages 491–506. Springer International Publishing, 2018.

[6] Antonio Brogi, Davide Neri, and Jacopo Soldani. DockerFinder: Multi-attribute search of Docker images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 273–278. IEEE, 2017.

[7] Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. TosKer: A synergy between TOSCA and Docker for orchestrating multi-component applications. *Software: Practice and Experience*, 2018. *[In press]*.

[8] Antonio Brogi and Jacopo Soldani. Matching cloud services with TOSCA. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers*, pages 218–232. Springer, 2013.

[9] Antonio Brogi and Jacopo Soldani. Finding available services in TOSCA-compliant clouds. *Science of Computer Programming*, 115:177 – 198, 2016.

[10] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a nutshell: Promises and perspectives. In Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors, *Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings*, pages 171–186. Springer Berlin Heidelberg, 2014.

[11] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[12] Beniamino Di Martino, Dana Petcu, Roberto Cossu, Pedro Goncalves, Tamás Máhr, and Miguel Loichate. Building a mosaic of clouds. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *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*. Springer Berlin Heidelberg, 2011.

[13] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[14] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A service-oriented framework for developing cross cloud migratable software. *J. Syst. Softw.*, 86(9):2294–2308, 2013.

[15] Mohammad Hamdaqa, Tassos Livogiannis, and Ladan Tahvildari. A reference model for developing cloud applications. In Frank Leymann, Ivan Ivanov, Marten van Sinderen, and Boris Shishkov, editors, *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science*.

[16] Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, and Frank Leymann. Automatic topology completion of TOSCA-based cloud applications. In *44. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume 232, pages 247–258. Lecture Notes in Informatics (LNI), 2014.

[17] Frank Leymann. Cloud computing. *it — Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(4):163–164, 2011.

[18] Karl Matthias and Sean P. Kane. *Docker: Up and Running*. O'Reilly Media, 2015.

[19] Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2015.

[20] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[21] OASIS. 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.

[22] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Simple Profile in YAML, Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf`, 2016.

[23] OpenStack development community. Parser for TOSCA simple profile in YAML. `https://github.com/openstack/tosca-parser`.

[24] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*. `https://doi.org/10.1109/TCC.2017.2702586`, *[In press]*.

[25] Randall Smith. *Docker Orchestration*. Packt Publishing, 2017.

[26] Jacopo Soldani, Tobias Binz, Uwe Breitenbücher, Frank Leymann, and Antonio Brogi. ToscaMart: A method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406, 2016.

[27] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[28] Weaveworks Inc. and Container Solutions Inc. Sock Shop. `https://microservices-demo.github.io/`. last accessed December 1st, 2017.

[29] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 105–108. IEEE, 2017.