From (incomplete) TOSCA specs to running apps, with Docker

Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract. 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 paper, we show how TOSCA and Docker can be effectively exploited to orchestrate applications, even if their (runtime) specification is incomplete. More precisely, we 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 then discuss how the obtained specifications can be automatically orchestrated by existing TOSCA engines.

1 Introduction

Cloud computing permits running on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [1]. 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 [7].

At the same time, the need for orchestrating the management of multi-component applications across heterogeneous cloud platforms has emerged [13]. The deployment, configuration, enactment and termination of the components forming an application must be suitably orchestrated. This must be done by taking into account 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 [9].

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 [15]. 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 [2]. 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 automatically modify the virtual environment running an application component whenever they wish.

In this paper, we present a solution geared towards providing such an enhanced support. Our solution is based on TOSCA [17], the OASIS standard for orchestrating cloud applications, and on Docker, the de-facto standard for cloud container virtualisation [18]:

– We first propose a TOSCA-based representation for multi-component applications, which can be used to specify the components forming an application, the dependencies among them, and the software support that each component requires to effectively run.
– We then present 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 [4], for instance). Such engines will automatically orchestrate the deployment and management of the corresponding applications on top of the specified runtime environments.

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. Sects. 6 and 7 discuss related work and draw some concluding remarks, respectively.

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:

[1] The source code of *Thinking* is publicly available on GitHub at `https://github.com/di-unipi-socc/thinking`.

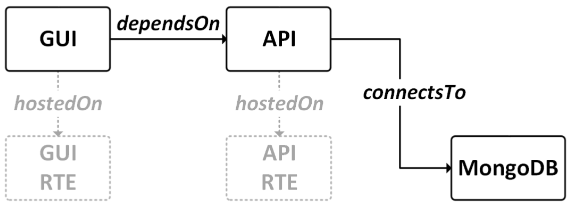

Fig. 1. Running example: The application *Thinking*.

(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 [18]. However, we have currently 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 have then to manually package the *API* and *GUI* components within such Docker containers, and to explicitly describe the orchestration of all the Docker containers in our application. In other words sense, we have to identify, develop, configure and orchestrate all components in Fig. 1, including those not specific to the *Thinking* application (viz., the lighter nodes *API RTE* and *GUI RTE*).

Our effort would be much lower if we were provided with a support requiring us to describe our application only, and automating the 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 application *Thinking*. 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* [17]) 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

[2] `https://hub.docker.com/_/mongo/`.

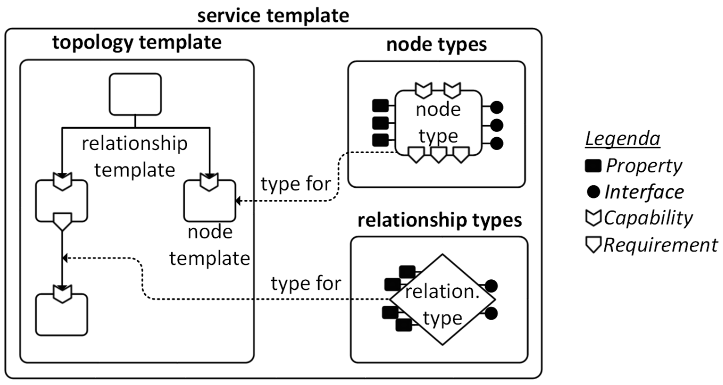

Fig. 2. The TOSCA metamodel [17].

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].

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.

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 permitting 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 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.

[3] A more detailed, self-contained introduction to TOSCA can be found in [2,6].

TOSCA applications are 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 [21] exploits the kernel of the host's operating system 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. 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 it removes volumes that are no longer referenced by any container.

Docker also allows containers to intercommunicate. It indeed permits creating virtual networks, which span from bridge networks (for single hosts), to complex overlay networks (for clusters of hosts)[4].

4 Specifying apps only, with TOSCA

Multi-component applications typically integrate various and heterogenous components [9]. We hereby propose a TOSCA-based representation for such components (Sect. 4.1). We also illustrate how such representation can be used to specify only the components 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 multi-component apps

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).

[4] A more detailed introduction to Docker can be found in [14,19].

[5] The actual TOSCA definition of all node types discussed in this section is publicly available on GitHub at `https://github.com/di-unipi-socc/tosker-types`.

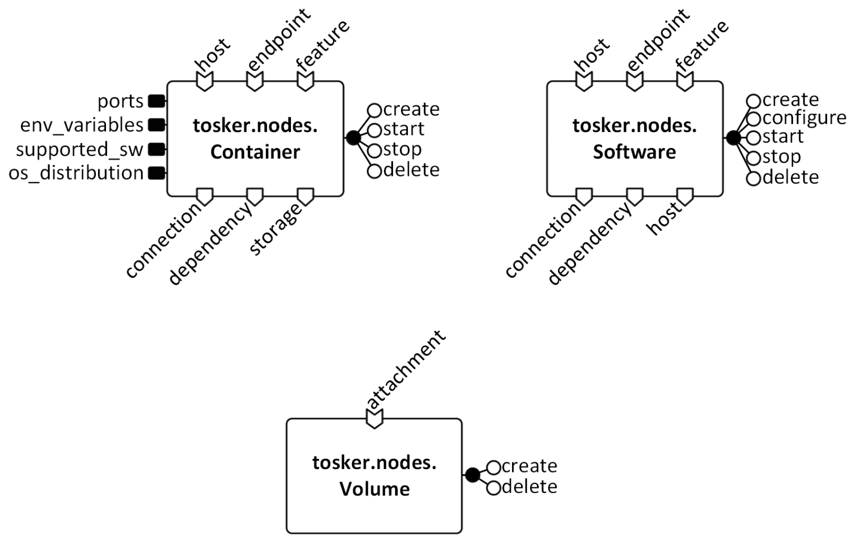

Fig. 3. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

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 [14]).

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.

tosker.nodes.Volume permits specifying 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 basic operations offered by the Docker platform [14]).

tosker.nodes.Software permits indicating 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[6]. *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 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*).

The interconnections and interdependencies among the nodes forming a multi-component application can then be indicated by exploiting the TOSCA normative relationship types [17]. 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.

4.2 Specifying app-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. 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[7].
– *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 *host* requirement is mandatory for nodes of type *tosker.nodes.Software*, as we assume that each application component must be installed in another component or in a Docker container.

[7] The documentation of `mongo` explicitly states that a `mongo` container must be attached to a Docker volume to persistently store data.

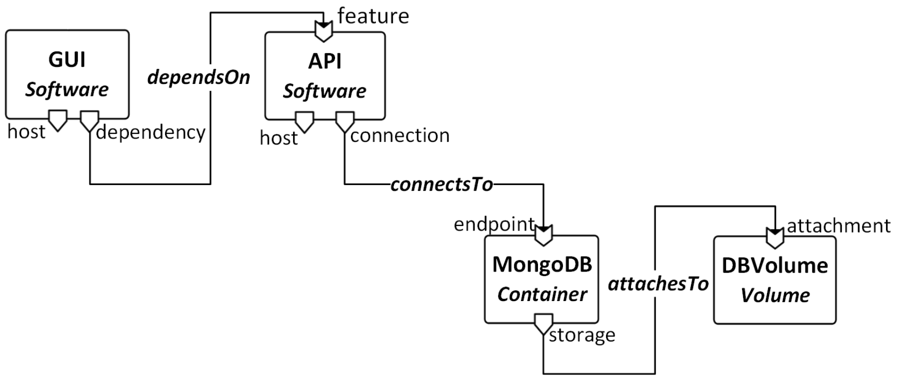

Fig. 4. A specification of our running example in TOSCA (where nodes are typed with *tosker.nodes.Container*, *tosker.nodes.Volume*, or *tosker.nodes.Software*, while relationships are typed with TOSCA normative types [17]).

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 [17], 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 permit 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 node of type *tosker.nodes.Container*). We can also indicate constraints on the software support to be provided by such containers, on the operating system distribution they must run, and on how to configure them (e.g., which port mappings they must expose, or which environment variables they should define)

Example (cont.). 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

```
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)

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

git (any version). It also indicates a port mapping to be configured in the hosting container and that such container must be based on a Ubuntu distribution[8].

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. □

5 Completing TOSCA specs, with Docker

We hereby present TosKeriser, an open-source prototype tool[9] 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 Sect. 4.2).

TosKeriser is a command-line tool, which works as illustrated in Fig. 6:

❶ TosKeriser inputs a (CSAR or YAML) file containing a TOSCA application specification. It then parses the application topology, and it identifies the set of software components whose requirement *host* has to be fulfilled (according to the constraints indicated in the clause `node_filter` of such requirement).

[8] 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 Ubuntu-based distributions).

[9] The Python sources of TosKeriser are publicly available on GitHub at `https://github.com/di-unipi-socc/toskeriser` (under MIT license).

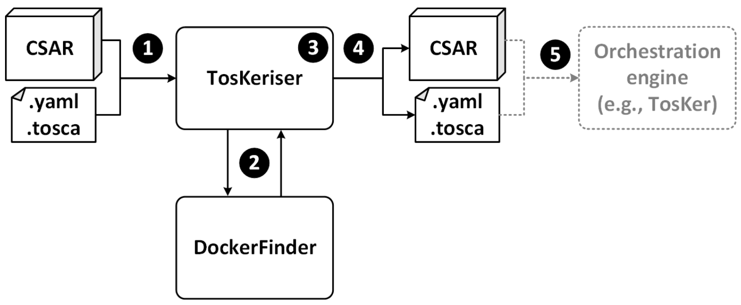

Fig. 6. How TosKeriser works.

❷ For each of such components, it invokes DockerFinder[10] to identify a Docker container providing the needed support (viz., satisfying the constraints concerning the `supported_sw` and the `os_distribution`).
❸ The discovered containers are then included in the application topology. More precisely, TosKeriser satisfies the pending requirements *host* by connecting them to new nodes of type *tosker.nodes.Container*. Each of the newly introduced nodes is configured to satisfy the constraints indicated by the software components it hosts (e.g., if a software component is requiring some port mappings, then the newly introduced container that hosts it will have the property *port* set accordingly).
❹ TosKeriser outputs the (CSAR or YAML) file containing the automatically completed TOSCA application specification.
❺ The obtained file can then be passed to an orchestration engine supporting TOSCA and Docker (e.g., TosKer [4]), which will automatically deploy and manage the actual instances of the specified application.

Example. Consider again the application *Thinking* in our motivating scenario, whose corresponding TOSCA representation is displayed in Fig. 4. The CSAR file (`thinking.csar`) containing the TOSCA application specification of *Thinking* is publicly available on GitHub[11]. Such file can be automatically completed by executing the following command:

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

The above will generate a new file (`thinking.completed.csar`), whose topology is completed by including two new Docker containers, namely *APIContainer* and *GUIContainer* (Fig. 7, lighter nodes). Such nodes provide the software support and the port mappings needed by *API* and *GUI*, respectively. We can then run

[10] DockerFinder [3] is a tool allowing to search for Docker containers based on multiple attributes, including the software distributions they support and the operating system distribution they are based on.

[11] `https://github.com/di-unipi-socc/TosKeriser/blob/master/toskeriser/tests/examples/thinking-app/thinking.csar`.

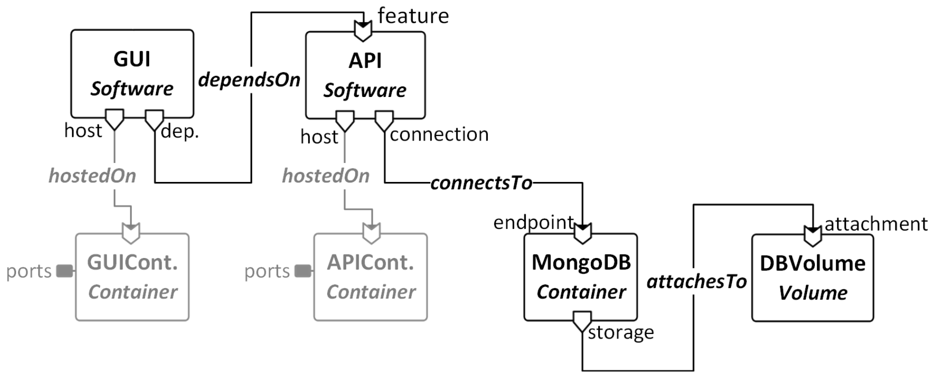

Fig. 7. Application topology obtained by completing the partial topology of the application *Thinking* (Fig. 4). Lighter nodes and relationships are automatically included by TosKeriser.

such file TosKer [4] (or with another orchestration engine supporting TOSCA and Docker), which will automatically deploy and manage the actual instances of the specified application (see Appendix A).

Please note that we run TosKeriser with the option `--policy size`. The latter instructs TosKeriser to concretely implement *APIContainer* and *GUIContainer* with the images of Docker containers having the smallest size (among all images of containers providing the needed software support). Suppose now that we wish to change the containers used to host *GUI* and *API*, e.g., because now we want to select the containers are most used by Docker users. We can run again TosKeriser on the obtained specification, by setting the option `-f` to force TosKeriser to change the actual implementation of the Docker containers it previously created (viz., *APIContainer* and *GUIContainer*):

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

□

6 Related work

We presented a solution for automatically completing TOSCA specifications, much in the spirit of [12]. The latter indeed inputs TOSCA specifications containing only the components specific to an application, and it automatically determines their runtime environments. However, [12] only checks type-compatibility between nodes and runtime environments, while we also allow developers to impose additional constraints on the nodes that can be used to host a component (e.g., by allowing to indicate a component requires a certain software support on a certain operating system distribution).

[5] and [20] can also be used to automatically determine the runtime environment needed by the components of an applications. They indeed allow to

abstractly specify desired nodes, and they determine actual implementations for such nodes by matching and adapting existing TOSCA application specifications. [5] and [20] 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 to 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, [8] and [11] 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 applications' source code, and it is hence applicable also to third-party components whose source code is not available nor open.

[10] proposes a framework allowing developers to write the source code of cloud applications as if they were "on-premise" applications. [10] 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. [10] 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 [4]).

In general, most existing approaches to the reuse of cloud services 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 which proposes an approach for adapting cloud applications to work with heterogeneous cloud platforms, by relying on TOSCA [17] and Docker to achieve cloud interoperability, and by supporting an easy (re)use of third-party components.

On the one hand, TOSCA is proved to allow automating the orchestration of a multi-component application, thanks to the fact that deployment and management plans can be directly inferred from its topology [2,16]. On the other hand, Docker standardises the virtual runtime environment of application components to a Linux-based environment [18], hence allowing to implement their deployment and management operations as artefacts supported by such environment.

7 Conclusions

Cloud applications typically consist of multiple heterogeneous components, whose deployment, configuration, enactment and termination must be suitably orchestrated [9]. 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 [4], which can process specifications produced by TosKeriser, to automatically orchestrate the deployment and management of the corresponding applications.

TosKeriser is integrated with DockerFinder [3], and it produces specifications that can be effectively processed by TosKer [4]. TosKeriser, DockerFinder and TosKer are all open-source tools, and their ensemble provides a first support for automating the orchestration of multi-component applications with TOSCA and Docker. We plan to further extend this ensemble, 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 perspective, an interesting direction for future work is to investigate whether existing approaches for reusing fragments of TOSCA applications (e.g., ToscaMart [20]) 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.

TosKeriser currently relies only on DockerFinder [3] to search for existing images of Docker containers. If there is no image providing the software support and the operating system distribution needed by an application component, TosKeriser cannot complete the corresponding TOSCA specification of the application containing such component. This could be avoided by supporting the creation of ad-hoc images (configured from scratch, if necessary). The development of a tool allowing to build ad-hoc images, as well as its integration with TosKeriser, is in the scope of our immediate future work.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Communications of the ACM 53(4), 50–58 (2010)
2. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer, New York, NY (2014)

3. Brogi, A., Neri, D., Soldani, J.: DockerFinder: Multi-attribute search of Docker images. In: 2017 IEEE International Conference on Cloud Engineering (IC2E). pp. 273–278. IEEE (2017)
4. Brogi, A., Rinaldi, L., Soldani, J.: TosKer: Orchestrating applications with TOSCA and Docker., 2017. *[Submitted for publication]*
5. Brogi, A., Soldani, J.: Finding available services in tosca-compliant clouds. Science of Computer Programming 115, 177 – 198 (2016)
6. Brogi, A., Soldani, J., Wang, P.: TOSCA in a nutshell: Promises and perspectives. In: Villari, M., Zimmermann, W., Lau, K.K. (eds.) Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings. pp. 171–186. Springer Berlin Heidelberg (2014)
7. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: 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)
8. Di Martino, B., Petcu, D., Cossu, R., Goncalves, P., Máhr, T., Loichate, M.: Building a mosaic of clouds. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) 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. pp. 571–578. Springer Berlin Heidelberg (2011)
9. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)
10. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: A service-oriented framework for developing cross cloud migratable software. J. Syst. Softw. 86(9), 2294–2308 (2013)
11. Hamdaqa, M., Livogiannis, T., Tahvildari, L.: A reference model for developing cloud applications. In: Leymann, F., Ivanov, I., van Sinderen, M., Shishkov, B. (eds.) CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science
12. Hirmer, P., Breitenbücher, U., Binz, T., Leymann, F.: Automatic topology completion of tosca-based cloud applications. In: 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI). vol. 232, pp. 247–258. Lecture Notes in Informatics (LNI) (2014)
13. Leymann, F.: Cloud computing. it — Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik 53(4), 163–164 (2011)
14. Matthias, K., Kane, S.P.: Docker: Up and Running. O'Reilly Media (2015)
15. Newman, S.: Building microservices. O'Reilly Media, Inc. (2015)
16. 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)
17. 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)
18. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing `https://doi.org/10.1109/TCC.2017.2702586`, *[In press]*

19. Smith, R.: Docker Orchestration. Packt Publishing (2017)
20. Soldani, J., Binz, T., Breitenbcher, U., Leymann, F., Brogi, A.: ToscaMart: A method for adapting and reusing cloud applications. Journal of Systems and Software 113, 395–406 (2016)
21. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. SIGOPS Oper. Syst. Rev. 41(3), 275–287 (2007)

A Appendix

This appendix is included only for the convenience of the reviewers. It will not be included in the final version of the paper.

This appendix provides some additional details on how to concretely use TosKeriser (and TosKer) to automatically complete and orchestrate applications with TOSCA and Docker.

A.1 Usage guide of TosKeriser

TosKer is a Python command line tool available on the PyPI index and can be installed on Linux using *pip* package manager with the following command:

```
$ sudo pip install toskeriser
```

It is then possible to use TosKeriser by executing the command `toskerise`, whose usage is below

```
toskerise FILE [COMPONENT..] [OPTIONS]
toskerise --help|-h
toskerise --version|-v

FILE
  TOSCA YAML file or a CSAR to be completed

COMPONENT
  a list of component to be completed (by default all
  components are considered)

OPTIONS
  --debug                             active debug mode
  -q|--quiet                          active quiet mode
  -i|--interactive                    active interactive mode
  -f|--force                          force updating all
                                      containers
  --constraints=value                 additional constraints
                                      for searching images
  --policy=top_rated|size|most_used   policy for sorting images
```

The command inputs a TOSCA file (YAML or CSAR), along with an optional list of components to be analysed. If no component is specified, then all components are analysed. There is also an option `--contraints`, which permits to specify additional constraints on how to search for images (e.g., searching images whose size is lower of `200MB`). The option `--policy` can instead be used to indicate which images to privilege, viz., `size` for smallest images, `most_used` for most pulled images, and `top_rated` for images best rated by Docker users.

The flags `-i` and `--interactive` allow users to choose the image to use from a list of appropriate images. The flags `-f` and `--interactive` force TosKeriser to search for new Docker containers for hosting the application components (even if the latter already have their requirements *host* satisfied). Finally, the option `--debug` shows debugging logs during execution, while flags `-q` or `--quiet` prevent writing on the standard output.

Please note that TosKeriser comes with a set of example showing how to use `toskerise`. Such example are avaible in the folder `/usr/share/examples`.

A.2 From `thinking.csar` to a running instance of *Thinking*

This section is intended to complement the examples available in this paper. ‘ It indeed illustrates a concrete run of an instance of the application *Thinking*, which is obtained by passing a specification outputted by TosKeriser to
After completing `thinking.csar` by running

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

we can instruct TosKer[12] to automatically deploy and run an instance *Thinking* by executing the following command:

```
$ tosker thinking.completed.csar create start --gui_port 80
```

After the execution of this command, TosKer automatically created, run and interconnected all components and containers in the topology of the application *Thinking*. It also exposed port `80`, and it setted port mappings so that the *GUI* was accessible through such port (as shown in Fig. 8)

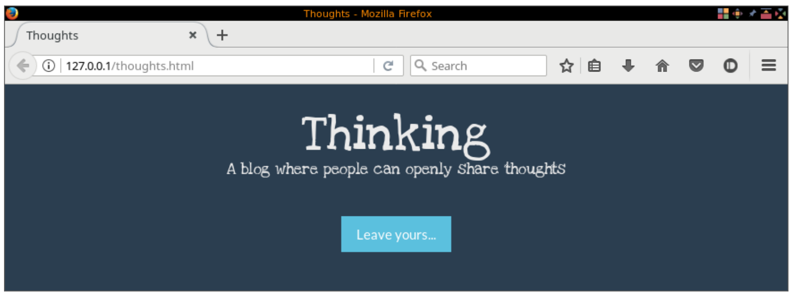
Thoughts - Mozilla Firefox
Thoughts
127.0.0.1/thoughts.html
Search
Thinking
A blog where people can openly share thoughts
Leave yours...

Fig. 8. A snapshot of the obtained instance of (the *GUI* of) *Thinking*.

After playing with the instance of *Thinking*, we instructed TosKer to automatically remove it by executing:

```
$ tosker thinking.completed.csar stop delete
```

[12] TosKer is publicly available on GitHub at `https://github.com/di-unipi-socc/TosKer`, along with a README explaining how to install and use it.