TOSCA-based SWITCH Workbench for application composition and infrastructure planning of time-critical applications

Polona Štefanič*, Matej Cigale*, Francisco Quevedo Fernandez*, David Rogers*, Louise Knight*,
Andrew C. Jones* and Ian Taylor*
*School of Computer Science and Informatics,
Cardiff University,
Queen's Buildings, 5 The Parade, Roath, Cardiff CF24 3AA, UK
Email: [StefanicP | CigaleM | QuevedoFernandezF | RogersDM1 | KnightL2 | JonesAC | TaylorIJ1]@cardiff.ac.uk

***Abstract*—Real-time applications, such as disaster early warning systems, live event broadcasting, video conferencing and online gaming, present particular challenges for successful development and deployment: they only achieve their expected business value when they meet critical requirements, such as high performance and availability for outstanding Quality of Service and Quality of Experience. The development of time-critical applications needs to be supported by a customized software engineering environment that offers support for the entire application life-cycle. However, there is a lack of (component-based) software workbenches/tools suitable for time-critical applications, supporting customized software engineering through the entire life-cycle. In this paper we present the SIDE Workbench developed during the course of the SWITCH project. It uses TOSCA extensively for exchange of information within the SWITCH platform, and offers component-based application composition, software component modelling, infrastructure planning and provisioning through the entire life-cycle of time-critical applications. To show the application composition process we describe the development of a containerized interactive multi-tier application on the SIDE Workbench that is mapped into TOSCA, from which the corresponding Docker Compose file is then created.**

Keywords*—*Software engineering, time-critical applications, TOSCA, Distributed Cloud computing

I. Introduction

Time-critical applications such as disaster early warning systems (e.g. from BEIA[1]), on-demand business collaboration platforms (e.g. from Wellness Telecom[2]), live event broadcasting (e.g. from MOG Technologies[3]), online gaming, and so on, can only achieve their expected business value or social impact when they satisfy critical requirements, such as high performance, availability and scalability. They must achieve consistently high Quality of Service (QoS) through their entire runtime — not least, in order to achieve the required Quality of Experience (QoE). This group of time-critical applications is very difficult to develop and maintain. Usually a customized software engineering environment is needed which supports the developer in specifying the QoS and other requirements and provides facilities for deployment, monitoring, steering, etc. throughout the entire life-cycle.

In recent years research effort has been expended in the development of tools, application interfaces and specifications for component-based development of cloud applications, orchestration, virtualization and automatic deployment. However, here is still a lack of tools, application programming interfaces and specifications that can fully support the development of time-critical applications through their entire life-cycle in an intelligent and autonomous manner [1].

In this paper we present the SIDE (SWITCH Interactive Development Environment) Workbench, which has been developed during the SWITCH (Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications) project[4]. It supports the development of Microservice and Multi-tier applications throughout their entire life-cycle. It uses OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [2] a specification that provides the following functionalities [3]:

(i) automated application deployment and management,
(ii) portability of application descriptions,
(iii) interoperability and re-usability of components,
(iv) orchestration, controllability and programmability,
(v) description of components, their relationships and dependencies among them.

One of the main novelties of the SIDE Workbench is that it offers component-based creation of cloud applications by dragging and dropping the components from a component repository to the SIDE Workbench canvas and, for each component, specifying associated quality constraints and hardware requirements and other dependencies and components. The other main novelty presented here is that it allows mapping all this into TOSCA, for exchange of information with other SWITCH components.

This paper is structured as follows. Section II presents related work such as various orchestration-based specifications and languages, known OASIS TOSCA implementations and similar projects that enable cloud orchestration and provisioning. Section III briefly introduces all three subsystems of SWITCH with their corresponding architecture components. In section IV the component-based infrastructure with its

[1]http://www.beiaro.eu/services/
[2]http://www.wtelecom.es/?lang=en
[3]http://www.mog-technologies.com/
[4]http://www.switchproject.eu/

supporting nodes is presented. Section V presents an example of how an application can be created via the SIDE Workbench and mapped into TOSCA. Finally, with section VI we conclude the paper by giving some further remarks and discussing future potential enhancements of the SIDE Workbench.

II. Related work

In this section we present an overview of (i) orchestration, automation and interoperability based specifications and languages, (ii) known TOSCA implementations in various cloud-based projects, platforms and libraries that implement or visualize TOSCA tools, and (iii) cloud-based projects and tools that support orchestration, but do not implement TOSCA.

A. Orchestration-based specifications, languages and libraries

Cloud Application Management for Platforms (CAMP) [4] is a language, framework and platform independent OASIS specification standard for the interoperability and management of applications across multiple cloud infrastructures, offered as a PaaS. CAMP consists of the following related resources: Platform Assemblies, Platform Components and Application Components. In contrast to these general specifications, AWS CloudFormation [5] is a platform dependent orchestration approach that relies on JSON templates. It describes AWS related resources (e.g. EC2, S3 bucket, Load balancer). The creation of AWS resources is described in JSON templates where the platform automatically orchestrates the relationships and dependencies among them. OpenStack [6] is an open source cloud platform that supports on-demand infrastructure and resource provisioning. It comes with an interactive web interface that offers management of the resources and monitoring. HEAT [7] is a project of the OpenStack Orchestration program that enables infrastructure provisioning and a cross-compatible AWS CloudFormation implementation for OpenStack with templates that are based on the YAML language format. Business Process Execution Language (BPEL) [8] is an OASIS executable language for the orchestration, management planning and specification of actions within business processes in web services that is based on XML. BPEL describes interactions with web services as executable and abstract business processes. However, BPEL is business-oriented and is not suitable for time-critical application. CHEF [9] is an open source configuration management tool that can be integrated with various cloud platforms, such as Internap, Amazon EC2, Google Cloud Platform, OpenStack and so on. It offers automatic provisioning and configuration of machines and resources. Various resources can be configured by writing Ruby-based scripts called recipes that describe resources, automation tasks, their relationships and dependencies, such as installation of packages, running instances and so on. These recipes are organized in cookbooks of Ruby scripts. TOSCA2Chef [10] is a service orchestration framework that comes with a set of components and services in order to support two basic operations: parsing TOSCA for the deployment of the defined topology and the execution of the application logic. Furthermore, we have analyzed languages that can be used for designing and modelling real-time applications, such as MARTE [11] and SysML [12]. MARTE and SysML are modelling languages used for defining real-time applications allowing the specification of QoS attributes by using object constraints language extensions to UML models. They both facilitate the formal verification of a system by transforming some of the models into Timed Petri Nets or Layered Queuing Networks over which various verification techniques can be run in order to verify the quality attributes.

B. Known TOSCA implementations

There are several industrial cloud-based tools and research projects that implement TOSCA, but they are not suitable for time-critical applications in their current form. Alien4Cloud [13] is an open source enterprise platform that enables selection of the best infrastructure on-demand in any phase of the application life-cycle by abstracting the application requirements according to the infrastructure and its resources catalog [13]. The key features of Alien4Cloud are design and portability of applications using TOSCA, DevOps, supporting collaboration among development and deployment teams, application life-cycle management, deployment and post-deployment, orchestration, deployable middleware recipes and blueprints catalog. Although Alien4Cloud is a cloud-based platform for the development and deployment of cloud applications, it is more focused on business enterprise solutions than on time-critical applications. Apache ARIA [14] TOSCA orchestration engine is an open source, embeddable, lightweight library and command line tool for the TOSCA-based modelling and orchestration of cloud-based applications. CELAR [15] is an Elasticity Provisioning Platform that provides methods and tools for the integration and orchestration of the various sub-modules of CELAR, such as the decision module, monitoring system, application description tool and so on. Cloudify [16] is an open source TOSCA-based model-driven framework and management platform for cloud orchestration enabling the modelling of applications and services and automation of their entire life-cycle. Similar to [13] its aim is getting the business and developers to work together, and the management and automation of the entire application life-cycle. DICER [17] is a tool that enables the model-driven deployment of Big Data applications. It automatically generates Infrastructure as Code (IAC) for Big Data applications from UML models into TOSCA blueprints. The OpenTOSCA Container [18] presents a runtime environment of the deployment and management of cloud applications. It enables the deployment and automated provisioning of applications that can be modeled by TOSCA and CSAR. It analyses a TOSCA model and invokes a Build Plan for instantiating new application instances. Ubicity uses TOSCA to model service topologies, including service composition, policies and custom work flows [19]. It offers a repository of normative and non-normative Types (e.g. node, group, artifact, interface types) [20] and a TOSCA validator [21]. Furthermore, there are some web-based tools that offer visual notation and graphical modelling of TOSCA topologies, plans and policies. The most common are Winery [22] and Vino4TOSCA [23]. However, these tools do not offer orchestration.

C. Tools that do not implement TOSCA

In the scope of industrial tools there are two software engineering tools that are used for the creation of native cloud applications and services, such as Juju [24] and Fabric8 [25].

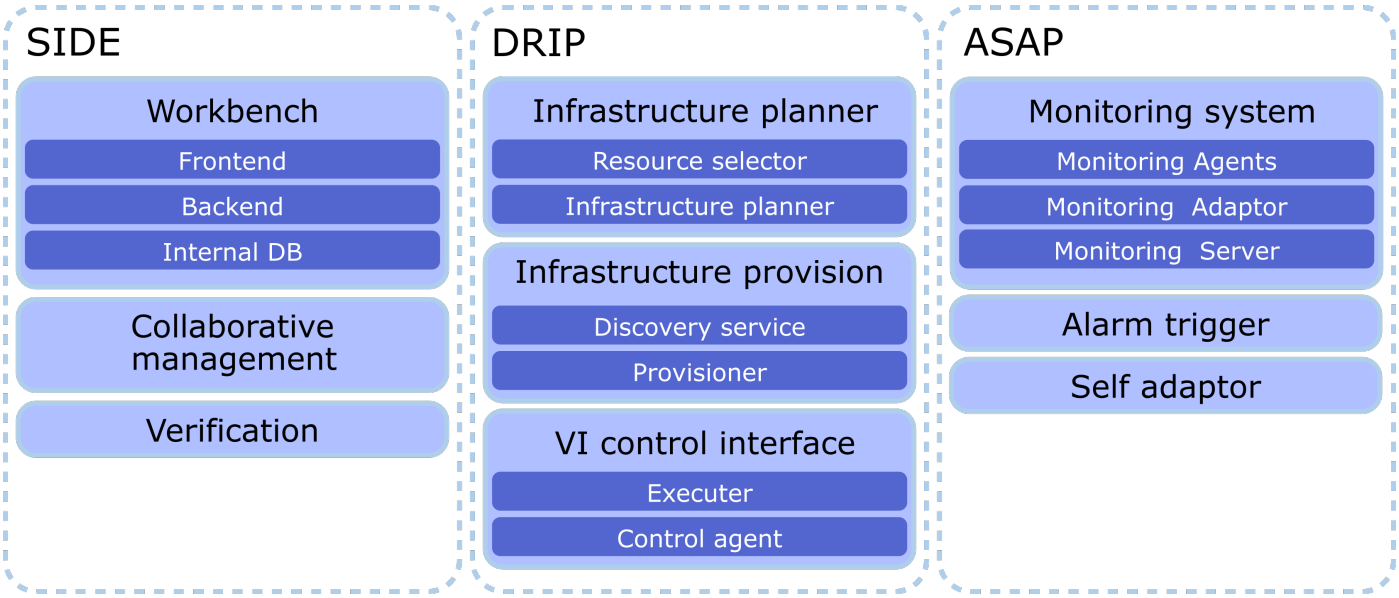

Fig. 1: Main components of the SWITCH architecture.

Fabric8 presents an open source platform that uses Docker containers and Kubernetes as visualization and orchestration technologies and enables the creation, deployment and continuous improvement of Microservices [25]. On the other hand Juju is a universal component-based graphical modelling tool for native application deployments. It offers sets of predefined software assets ("charms") and relationships and configurations among them that come with a knowledge of how to properly deploy and configure them in the cloud [24].

D. Summary

Most of the tools we have described offer orchestration, provisioning and automation of cloud-based applications, and some of them implement TOSCA as well. However, none of the above described projects, tools, libraries or platforms offers a workbench that would support component-based development of application logic, programmability and controllability throughout the entire life-cycle of cloud applications or services, modelling software components and enabling adaptation. These are all necessary features to support the development of time-critical applications.

III. The SWITCH concept

The overall objective of SWITCH is to develop an interactive Workbench and a set of middleware services that can support the entire life-cycle of time-critical cloud applications by enabling software engineers to specify critical requirements, and plan, provision and adapt the virtual cloud infrastructure for a specific application in order to ensure QoS. Figure 1 illustrates the main SWITCH functionalities.

A. SWITCH Interactive Development Environment subsystem

The SWITCH Interactive Development Environment (SIDE) subsystem provides interfaces for all stakeholders (e.g. software developers, application users) by providing a set of graphical interfaces and APIs that tie all SWITCH services to a Web-based workbench [26]. The aim of SIDE is to enable easier creation and definition of Microservice-based time-critical applications. In more detail, the main elements of SIDE are as follows:

1) Workbench: built using EmberJS MVC Framework as a front-end technology, Django Framework as a back-end technology and MySQL as its internal database. Using responsive HTML, EmberJS and JointJS the front-end implementation of the Workbench offers (i) composition of the application logic by dragging and dropping the components into the main canvas, defining QoS attributes and hardware requirements for specific components and linking the components in the application by connecting them together on the canvas, and (ii) defining the abstract infrastructure of the environment. All these features are mapped into TOSCA; one can also directly manipulate and edit the generated TOSCA. The back-end provides the APIs so that the application graph is stored and changed into TOSCA as well as working with the other systems using their APIs.

2) Collaborative management: offers application community support via user management facilities (e.g. user authentication and authorization and the personalization of user profiles – applications created on the SIDE Workbench belong to, and can be used only by, specific users or a group of users).

3) Formal reasoner/verifier: responsible for mapping application graphs created on the canvas into TOSCA, verification of TOSCA using semi-formal models and reasoners and exposing all functionalities through the REST APIs and sent to Dynamic Real-time Infrastructure planner.

B. Dynamic Real-time Infrastructure Planner subsystem

The Dynamic Real-time Infrastructure Planner (DRIP) subsystem is responsible for planning and provisioning of the infrastructure, and deployment and execution of applications.

TOSCA documents are sent to the DRIP manager, which offers various APIs for preparing and planning the application deployment [27]. The main modules of DRIP are as follows:

1) Infrastructure planner: The planner takes as its input the TOSCA generated by SIDE. It uses the information provided to estimate the required machine types and creates the plan that contains the information on what kind of VMs are needed and what is to be deployed on them. It does this with the information that is provided by the user.

2) Infrastructure provisioner: The provisioning system is described by H. Zhou et al. [28]. This provisioning can work with many different cloud providers such as EGI, AWS and others. The mechanism provides fast and transparent provisioning and enables the creation of different machine types based on the uploaded user credentials.

3) Automatic deployment: The final part of the system deals with the deployment of the application. The deployment is done in several stages, as first the machines need to be prepared, then they have to be connected into a cluster and finally the application is deployed with the required metadata. Part of deployment is also collecting the final information of the application, such as the IPs of the deployed containers, their IDs and other information.

C. Autonomous System Adaptation Platform subsystem

The Autonomous System Adaptation Platform is responsible for monitoring application status and the infrastructure within which the application is running, and dynamically adapting applications during runtime in order to ensure Quality of Service. ASAP monitors the application and based on the monitored metrics, and input from the user, triggers adaptation of the application.

1) Monitoring system: The monitoring system is a modified version of the JCatascopia monitoring system [29]. The monitoring system contains several conceptual components. Monitoring agents are part of an application. The Agents send the data using the StatsD protocol to the Monitoring Adapter. The adapter can service multiple agents. It takes the information received from the agents, registers a new Agent with the Monitoring Server and starts sending data to it. The Monitoring Server includes a Time-Series Database to store the data from the monitoring agents. It also includes a logic layer that provides several APIs, and a presentation layer. This architecture was used to ensure that the system is flexible enough so that the application can be lightweight while still providing the desired information.

2) Alarm trigger and Self-adapter: The Alarm trigger analyses the data collected by the monitoring system and triggers adaptation based on the definitions provided by the developer. The Alarm trigger sends this information to SIDE, which informs the user and the Self-adapter. The Self-adapter uses the adaptation provided by DRIP — ie. scale VM, scale Service etc. — and thus maintains the required system system performance when conditions change.

IV. Component-based infrastructure

Software developers can create containerized components from scratch in the composition view on the SIDE Workbench (see Figure 2) by packing services and other functionalities into container images, such as Docker containers. Alternatively the software developer can gather containerized components from public repositories, such as Docker Hub[5] or App Hub[6] and import and store them in the SIDE components repository. This information is stored in the Artifacts node of TOSCA.

A. Supporting nodes

Once a component is defined it needs to be described in SIDE so that the other SWITCH systems can use the descriptions. The nodes are attached to the created component and describe the requirements and the Non Functional parts of the application. We can see examples of each of these functionalities in Figure 2.

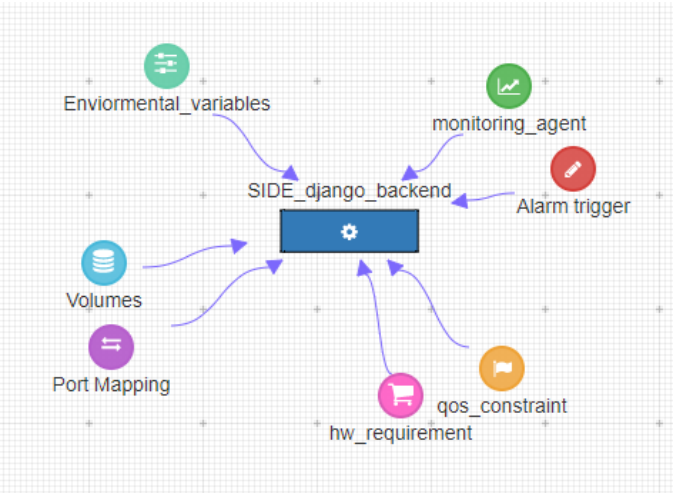

Fig. 2: Creation of the component.

Starting from QoS constraint and going in clockwise order, the functionalities that can be specified are:

- **Hardware requirements:** A node of this type enables the user to specify additional constraints on the planner for the VMs it needs. An example of these can be seen in Figure 3.

```
host:
  cpu_frequency: 2 GHz
  disk_size: 10 Gb
  mem_size: 2 Gb
  num_cpus: 1
os:
  architecture: x86_64
  distribution: ubuntu
  os_version: 16.04
  type: linux
```

Fig. 3: Defining the hardware requirements.

- **Quality of Service constraint:** Determines the metric and its value that is needed to maintain the desired

[5]https://hub.docker.com/
[6]https://apphub.io/

Quality of Experience of the system. This information is used by the planner to determine what kind of resources to provision, and by the modelling algorithm to create the Quality Metadata Markers (QMM) [30] that denote what affects the performance of the component (Figure 4).

```yaml
QoS:
  response_time: 50 ms
```

Fig. 4: Defining the Quality of Service metrics.

- **Monitoring agent:** This node denotes that the component includes a monitoring agent. This enables the SIDE Workbench to display monitoring information when the component is running. This information is also crucial during the validation stage, as SIDE uses this information to determine whether the Monitoring Adapter and Monitoring Server are necessary for the system to work. The SIDE Workbench will add these latter two components automatically as necessary if the user did not add them before.
- **Alarm trigger:** The user can specify the events that trigger the Alarms. The YAML file that is added also includes the information on what action to take (Figure 5).

```yaml
metric1:
    type: vm_level
    metric_group_name: CPUProbe
    subid: "1ccba0cc92174ce788695cfc0a027b57"
    properties:
        metric_name: cpuTotal
        data_type: double
        action: average
        units: percent
        period: 20
    range:
        minimum: 0.0
        maximum: 100.0
    alarm:
        warning:
            warning_value: 80.0
            warning_operator: ">="
        critical:
            critical_value: 100.0
            critical_operator: ">="
```

Fig. 5: Defining the Alarm trigger.

- **Port mapping:** If the user wants to expose certain ports to the outside world (s)he can specify these ports in this node. Internal ports, what they map to, what protocol the port supports and the mode of the port can all be defined, see Figure 6.
- **Environmental variables:** This node contains the information that the component needs to start its work. Information can include the IPs of other components, URLs of supporting services, width of the video and so on (Figure 7).

```yaml
ports_mapping:
  port_mapping_0:
    container_port: 8080
    host_port: 8080
```

Fig. 6: Defining the port mapping.

```yaml
ARI_URL: http://notify_asterisk:8088
ARI_USERNAME: ari_user
ARI_PASSWORD: ari_secret
REDIS_HOST: notify_redis
REDIS_PORT: 6379
DEFAULT_SIP: sipserver
MONITORING_ADAPTER: monitoring_adapter
```

Fig. 7: Defining the environmental variables.

- **Volumes:** This node includes the information about volumes, a Docker specific system that enables a container to mount parts of the disk, so that the information is not lost if the container is restarted (Figure 8).

```yaml
data-volume: "/var/lib/db"

```

Fig. 8: Defining Volumes.

B. Modelling the component

In order to make the usage of the component simpler for the developer a model can be created, that describes the relationship and dependencies between certain parameters of the infrastructure, such as CPU frequency, number of CPUs, RTT to a required service and so on and the QoS. This is important information as it gives the developer a glimpse into what it is required to make his or her component meet the desired NFR. This is elaborated on in the SWITCH IDE position paper [30].

The QMMs created by this system are qualitative i.e. they do not present the exact function of how the system will perform but only the general influence of a metric to the final application's QoS. It is enough information to give the user guidance in what part of the infrastructure (s)he should change in order to achieve the desired performance and improve the QoS and QoE of the application. The rationale for choosing these types of models is that they require less data to be created. This means that less testing of the application component is required. They are also more resistant to noise in the data that is especially strong in the network part of the system where the connections over the public Internet can change dramatically in a small amount of time.

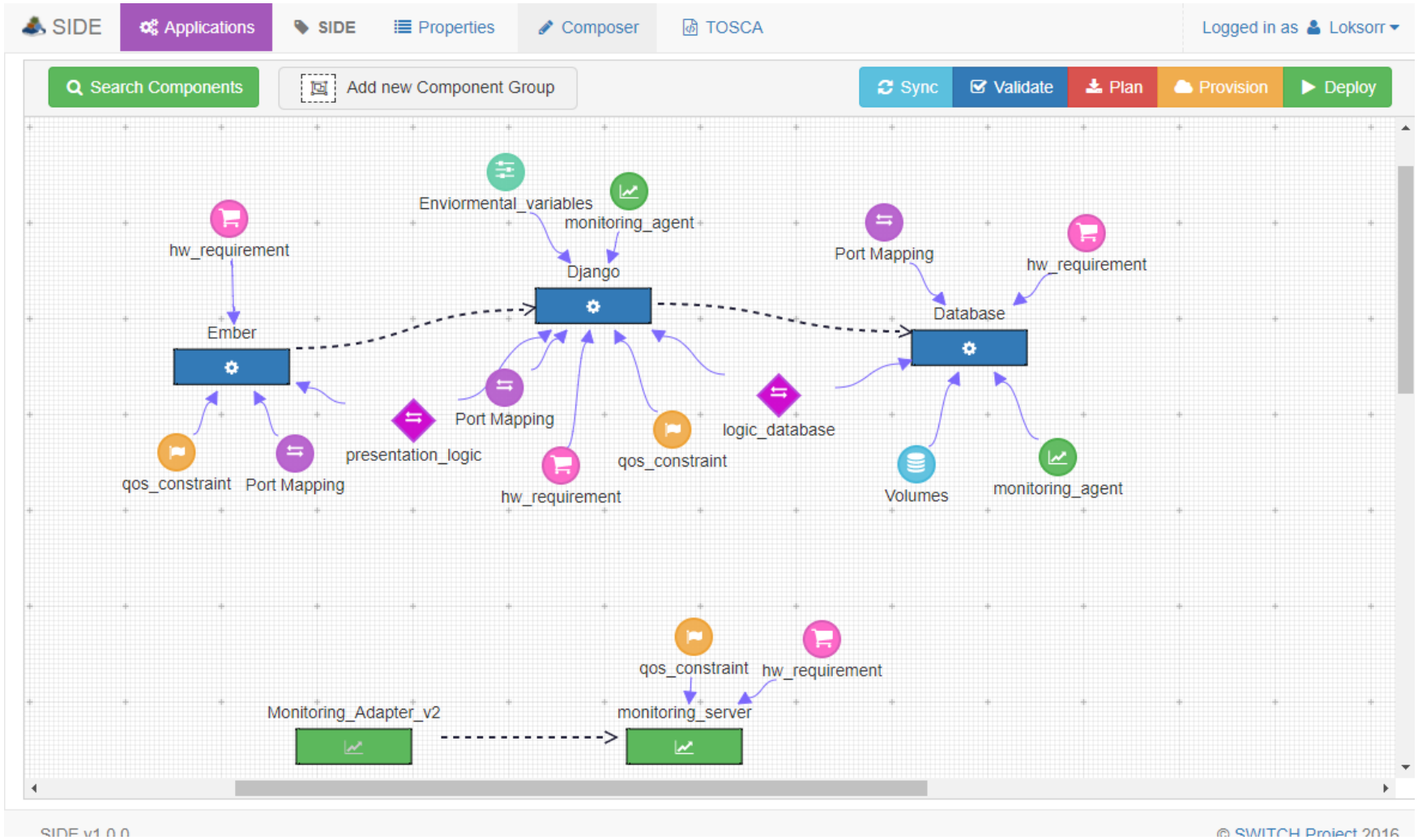

Fig. 9: An example of a Three-tier application created in SIDE.

V. SIDE Workbench

A. Connecting the pieces

Once the components are described they can be used to create an application. To illustrate this, we created a simple three-tier application, composed of the front-end, back-end and a database. To create this application the user simply drags the three components to the canvas, where they are created with the nodes attached. The user must then fill out some of the information of the system that could not be determined beforehand, for instance the required size of the disk for the Database, as this is application specific. The user can in addition connect some of the specific application components, such as networks, that enable the applications to reuse the same port mapping if applicable (Figure 10). The example contains definitions needed to use multicast, but it is usually enough to just provide the name of the network. The depends-on relationship is not strictly necessary for this example, as the components are robust enough to work even if the others are not reachable, but it is good practice to include this relationship, as this is not always the case.

```
driver: weaveworks/net-plugin:latest_release
driver_opts:
    works.weave.multicast: "1"
```

Fig. 10: An example of more complex network definition.

The information in the nodes can be changed for each deployment so that the performance of the application can be tweaked. If, for example, the user finds that the front-end is not sufficiently responsive, the number of CPUs can be increased, thus increasing the performance.

If the user clicks on the validate button the system indicates whether there are any values that were unintentionally left blank, thus forcing the user to fill them in for each component. This is denoted by the ”SET_ITS_VALUE” string. The system also checks if there is monitoring present in the system. If there is any monitoring node on the canvas, the SIDE Workbench will add the Monitoring Server and Monitoring Adapter to the system. If the user does not want to use monitoring, these nodes can be deleted from the canvas. Lastly the validator checks that each component on the canvas has an unique name. This is important, as the component names are mapped to service names in the Docker Compose file and are expected to be unique for each service in an application.

B. Example of creating a containerized Three-tier application

In order to show the creation of an application on the SIDE Workbench we have created a containerized three-tier application (see in Figure 9), each tier being realised as a distinct component. The three containers are:

- Basic application example of EmberJS MVC Framework as a Presentation tier
- Django Framework as a back-end Logic Tier
- MySQL Database as a Database tier

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: "Simpe three-tier Application"
topology_template:
  node_templates:

    Ember:
          artifacts:
        side_ember_image:
          type: "tosca.artifacts.Deployment.Image.Container.Docker"
          repository: SWITCH_docker_hub
          file: "SIDE/ember"
      requirements:
        dependency:
          - Django
        networks:
          - presentation_logic
      type: "Switch.nodes.Application.Container.Docker.Ember_frontend"
      properties:
        ports_mapping:
          ports_mapping:
            port_mapping_0:
              host_port: 7002
              container_port: 8080
        scaling_mode: single
        Constraints:
          QoS:
            response_time: 100 ms
    Django:
       artifacts:
        side_django_backend_image:
          type: "tosca.artifacts.Deployment.Image.Container.Docker"
          repository: SWITCH_docker_hub
          file: "side/django_backend"
      requirements:
        dependency:
          - Monitoring_Adapter_v2
          - Database
        networks:
          - monitoring_v2
          - presentation_logic
          - logic_database
      type: "Switch.nodes.Application.Container.Docker.Django_backend"
      properties:
        Environment_variables:
          variable: value
        ports_mapping:
          ports_mapping:
            port_mapping_0:
              host_port: 8080
              container_port: 8080
        scaling_mode: single
        Constraints:
          QoS:
            response_time: 20 ms
    Monitoring_Adapter_v2: [Removed for brevity.]
    monitoring_server: [Removed for brevity.]
    Database: [Removed for brevity.]
node_types: [Rest of TOSCA by definition. Removed for brevity.]
```

Fig. 11: An example of an application defined in TOSCA.

From the information provided by the user, the entire application graph is mapped into TOSCA and finally, a Docker Compose YAML file (see Figure 11 and Figure 12) is created.

C. Creation of TOSCA in the SIDE Workbench

Once the components are on the canvas, a corresponding TOSCA representation is generated by the system. The user can at any time check the TOSCA values to see what the actual composition of the application is. The components are defined as node templates. The properties denoted by the nodes are mirrored in the TOSCA definition.

In general each component has a fairly simple representation in TOSCA. An example can be seen in Figure 11.

The "SET_ITS_VALUE" text means that the user must fill out this information. The definitions of the types are removed for brevity. Other parts of the system, such as networks and volumes can be defined in a similar manner.

TOSCA is the exchange format within all SWITCH subsystems. It is the information that is sent from SIDE to DRIP and from DRIP to ASAP so that other services can perform their tasks. In order to simplify some of the parsing necessary SIDE provides additional endpoints that enable the component to read this information in a simplified manner.

D. Deploying using DRIP

Once the user is satisfied with the composition of the application (s)he can start the second phase, facilitated by DRIP. There are several functionalities that DRIP can provide:

1) Infrastructure Planning: The first of these steps is the infrastructure planning. This step looks at the components in the TOSCA and, based on the requirements and their internal testing, determines what kind of VMs are needed for the system.

2) Infrastructure Provisioning: The second part of this is provisioning, which takes the output of the planner and uses it to provision the appropriate VMs in the cloud. The planner needs the cloud credentials that are given by the user via SIDE to do its work. These credentials are stored inside DRIP.

```
Django:
  image: side/django_backend
  environment:
    variable: value
  ports:
    -"8080:8080"
  depends_on:
    - Monitoring_Adapter_v2
    - Database
  networks:
    - monitoring_v2
    - presentation_logic
    - logic_database
```

Fig. 12: An example of a component in Docker Compose.

3) Deploying the cluster: Once all the VMs have been provisioned the first part of deployment takes place. Essentially this sets up a SWARM[7] cluster on the created VMs. SWARM is a group of VMs that can run Docker containers and it allows transparent deployment of containers that might include replicas of services. SWARM provides certain recovery options, so that the system can recover if necessary. SWARM allows some interesting simplifications, such as the ability to access the ports opened by an application via the master node, thus simplifying the usage of the system.

4) Docker Compose: In order to deploy to SWARM the system needs a Docker Compose file. This is parsed from the TOSCA file created by SIDE. The Docker Compose is similar to the TOSCA but somewhat simpler, as it only deals with containers. Most of the data is a one-to-one transformation of the TOSCA YAML as seen in Figure 12.

After this the application deployment is assumed to be finished and the application is running and accessible on a specific IP address.

[7]https://docs.docker.com/engine/swarm/

VI. Conclusion

In this paper we have presented a complete overview of the SIDE Workbench in its current iteration. It allows the user to create an application, define the infrastructure the application will use, and start the application. The core of the workbench is the canvas that enables component creation and composition and deployment of the application. SIDE is designed in a manner that allows it to be very flexible. If the need for more resources arises, new nodes can be created in a relatively short amount of time, but there is still some room for standardization of how the system works allowing for an even greater level of flexibility. The SIDE Workbench is a flexible system which, at this point, relies on other systems to provide most of the functionality. Separation of concerns has been achieved, in that the exchange format, OASIS TOSCA, is an industry standard, and so SIDE could potentially be modified to work with other services and provide different capabilities for the system. The SIDE approach is therefore a robust way of specifying cloud applications that is still viable even if changes arise in the way applications are developed and something replaces the current technology of Docker containers.

There are many lessons that can be learned from the SIDE Workbench. However, one of the most important ones is that creating complex systems based on graphical interfaces quickly becomes unwieldy, necessitating the ability to partition the application into additional subsystems.

As mentioned, the SIDE Workbench provides a base that can be used to develop further systems in a similar vein. The connection of components is a powerful concept that can be further developed not only in the field of Microservices or Multi-Tier applications, but also in other aspects of application creation, such as creating Internet of Things (IoT) applications, functional programming and so on.

Another advantage of SIDE is that it is an inherently top-down system, allowing the creation of components on a larger scale and only then digging down into the details of the system; this is intended to be seen as an intuitive approach. From an industry perspective SIDE could be enhanced so that the system is more fully decoupled from the other systems. In this way the development can take a more robust approach to changing the system so that it can work with arbitrary systems using TOSCA. This sounds trivial, but would require a creation of a metalanguage to describe how and with which systems SIDE should communicate.

For the more immediate goals the validation functionality of the system is currently only at a basic level. Further work is needed to validate the applications and the connections between the components, so that communication between components can be even more flexible. Secondly integrated DevOps approaches need to be explored further and compared to the automatic planning and adaptation currently supported by the SWITCH system.

ACKNOWLEDGMENT

The research reported in this paper was funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 643963 (SWITCH project).

REFERENCES

[1] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, F. J. Hidalgo, G. Suciu, A. Ulisses, P. Ferreira, and C. d. Laat, "A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch)," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1181–1184.

[2] "Oasis topology and orchestration specification for cloud applications version 1.0," November 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf

[3] A. Brogi, J. Soldani, and P. Wang, *TOSCA in a Nutshell: Promises and Perspectives*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 171–186. [Online]. Available: https://doi.org/10.1007/978-3-662-44879-3_13

[4] M. Carlson, M. Chapman, A. Heneveld, S. Hinkelman, D. Johnston-Watt, A. Karmarkar, T. Kunze, A. Malhotra, J. Mischkinsky, A. Otto, V. Pandey, G. Pilz, Z. Song, and P. Yendluri, "Cloud Application Management for Platforms," December 2012. [Online]. Available: https://www.oasis-open.org/committees/download.php/47278/CAMP-v1.0.pdf

[5] Amazon Web Services, "AWS Cloud Formation documentation," December 2017. [Online]. Available: https://aws.amazon.com/cloudformation/

[6] OpenStack, "OpenStack Documentation," December 2017. [Online]. Available: https://www.openstack.org/

[7] ——, "Heat - OpenStack Orchestartion," December 2017. [Online]. Available: https://wiki.openstack.org/wiki/Heat

[8] WSBPEL TC., "Web services business process execution language version 2.0," December 2017. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

[9] OpsCode Inc., "Chef Documentation - Overview," December 2017. [Online]. Available: https://docs.chef.io/chef_overview.html

[10] G. Katsaros, M. Menzel, A. Lenk, J. R. Revelant, R. Skipp, and J. Eberhardt, "Cloud application portability with tosca, chef and openstack," in *2014 IEEE International Conference on Cloud Engineering*, March 2014, pp. 295–302.

[11] Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," June 2011. [Online]. Available: http://www.omg.org/spec/MARTE/1.1/

[12] SysML.org, "SysML Open Source Specification Project - What is SysML?" December 2017. [Online]. Available: http://sysml.org/

[13] ATOS, "Alien4Cloud 1.1 overview," December 2017. [Online]. Available: https://drive.google.com/file/d/0B-bJgbmOz4ipNlNfYkdsOUlocm8/view

[14] Apache Foundation, "AriaTOSCA," December 2017. [Online]. Available: http://incubator.apache.org/projects/ariatosca.html

[15] CELAR, "CELAR cloud project," December 2017. [Online]. Available: http://www.celarcloudproject.eu/

[16] Cloudify, "What is Cloudify - Documentation," December 2017. [Online]. Available: http://cloudify.co/

[17] Github Dicer project, "DICER project," December 2017. [Online]. Available: https://github.com/dice-project/DICER

[18] Open TOSCA, "OpenTOSCA goes Docker Compose," December 2017. [Online]. Available: http://www.opentosca.org/

[19] Ubicity, "TOSCA Validator," December 2017. [Online]. Available: https://ubicity.com/index.html

[20] ——, "TOSCA Types," December 2017. [Online]. Available: https://ubicity.com/types.html

[21] ——, "TOSCA Validator," December 2017. [Online]. Available: https://ubicity.com/validator.html

[22] Eclipse, "Eclipse Winery," December 2017. [Online]. Available: https://projects.eclipse.org/projects/soa.winery

[23] y. Uwe Breitenbücher and Tobias Binz and Oliver Kopp and Frank Leymann and David Schumm, booktitle=OTM Conferences, "Vino4tosca: A visual notation for application topologies based on tosca."

[24] K. Baxley, J. D. la Rosa, and M. Wenning, "Deploying workloads with juju and maas in ubuntu 14.04 lts," May 2014. [Online]. Available: https://linux.dell.com/files/whitepapers/Deploying_Workloads_With_Juju_And_MAAS-14.04LTS-Edition.pdf

[25] Fabric8, "Fabric8 documentation," December 2016. [Online]. Available: http://fabric8.io/guide/overview.html

[26] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses, and C. de Laat, "Developing and operating time critical applications in clouds: The state of the art and the switch approach," *Procedia Computer Science*, vol. 68, no. Supplement C, pp. 17 – 28, 2015, 1st International Conference on Cloud Forward: From Distributed to Complete Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915030653

[27] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao, "Planning virtual infrastructures for time critical applications with multiple deadline constraints," *Future Generation Computer Systems*, vol. 75, no. Supplement C, pp. 365 – 375, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17301905

[28] H. Zhou, Y. Hu, J. Wang, P. Martin, C. D. Laat, and Z. Zhao, "Fast and dynamic resource provisioning for quality critical cloud applications," in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2016, pp. 92–99.

[29] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Jcatascopia: Monitoring elastically adaptive applications in the cloud," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014, pp. 226–235.

[30] P. Štefanič, M. Cigale, A. Jones, and V. Stankovski, "Quality of service models for microservices and their integration into the switch ide," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sept 2017, pp. 215–218.