D2.5 Technical Description of the SIDE Subsystem

Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643963 (SWITCH project).

Start date of project: 01.02.2015. Duration: 36 months until 31.01.2018

Due Date:	31.07.2017
Delivery:	31.07.2017
Lead Partner:	CU
Dissemination Level*:	PU
Type:**	R
Status:	FINAL
Approved:	All partners
Version:	1.0.1

***Dissemination Level**	
PU	Public
CI	Classified, information as referred to in Commission Decision 2001/844/EC.
CO	Confidential, only for members of the consortium (including the Commission Services)
****Type**	
R	Document, report (excluding the periodic and final reports)
DEM	Demonstrator, pilot, prototype, plan designs
DEC	Websites, patents filing, press & media actions, videos, etc.
OTHER	Software, technical diagram, etc.

Contributors

The contributors to this deliverable are listed below.

Contributor	Role
Matej Cigale, Polona Štefanič, Andrew Jones (CU)	Editors
Matej Cigale (CU), Polona Štefanič (CU), Spiros Koulouzis (UvA), Sandi Gec (UL), Andrew Jones (CU), Ian Taylor (CU)	Authors
Paul Martin (UvA), Rui Amor (MOG), George Suciu (BEIA), Vlado Stankovski (UL)	Internal reviewers

Document History

Version	Date	Author	Description
0.0.1	22 May 2017	A Jones, M Cigale	Initial outline/synopsis
0.1	18 June 2017	A Jones, M Cigale, P Štefanič	Moved to Google Docs; assembled multiple sections …
0.2	26 June 2017	A Jones, M Cigale, P Štefanič	First draft available for review.
0.3	7 July 2017	P Štefanič	Convert Google Doc to Word Document for tracking changes.
0.4	10 July 2017	P Štefanič	Reorganization of the deliverable.
0.5	17 July 2017	P Štefanič	Generating Dynamic Smart Templates
0.6	18 July 2017	M Cigale	Revision
0.7	18 July 2017	P Štefanič	Revision and submission for internal review
0.8	20 July 2017	M Cigale	Incorporated reviewers' feedback
0.9	21 July 2017	P Štefanič	Minor revisions
0.10	24 July 2017	M Cigale	Incorporated additional reviewers' feedback
0.11	27 July 2017	M Cigale, A Jones	Incorporated Cardiff internal review feedback
0.12	27 July 2017	M Cigale, A Jones	Executive summary; minor updates
1.0	28 July 2017	M Cigale, A Jones	Minor updates; candidate version for delivery to European Commission
1.0.1	30 July 2017	A Jones	Minor updates; final version for delivery to European Commission

Keyword list

Component-based software engineering, TOSCA, Dynamic Smart Templating System, QoS modelling

Table of Contents

Executive Summary 4
1 Introduction 5
2 Role of the SIDE Subsystem within the SWITCH platform 6
2.1 Component-based and co-programming software engineering in SIDE 6
2.2 Relationship of SIDE to DRIP & ASAP subsystems 8
3 Technical implementation of SIDE 9
3.1 SIDE internal architecture and technologies 10
3.2 Dynamic behaviour and internal communication in SIDE: application walk-through 13
3.2.1 Component composition 16
3.2.2 Application composition 18
3.2.3 Instance management view 20
3.3 Dynamic Smart Templates 21
3.3.1 The Dynamic Smart Templating (DST) APIs 23
3.3.2 The Dynamic Smart Templating Form Generator 26
3.4 Integration of SIDE and DRIP subsystems 28
3.5 Integration of SIDE and ASAP Subsystem 31
4 Non-functional requirements in SWITCH SIDE 32
4.1 Assisting the user in choosing among conflicting NFRs 32
4.2 Quality of Service Models for informed NFR monitoring 33
5 Agenda for the final phase of SWITCH 35
5.1 Usability evaluation approach 36
6 Summary 38
6.1 Software functionality in public releases 38
6.2 Innovation 40
Bibliography 41
Abbreviations 42

Executive Summary

In the SWITCH platform, the SIDE (SWITCH Interactive Development Environment) sub-system is responsible for providing the front-end to the software developer which supports the entire time-critical cloud software life-cycle: it enables the creation, deployment and management of applications on the SWITCH platform. It works in conjunction with the DRIP (Dynamic Real-time Infrastructure Planner) and ASAP (Autonomous System Adaptation Platform) subsystems in order to provide a novel application-infrastructure co-programming software development metaphor, in which the developer's real-time application and the infrastructure are developed together, specifying performance and other non-functional requirements.

A number of key developments have been necessary in order to realise the SIDE subsystem, the "look-and-feel" of which has been discussed in previous deliverables. The present deliverable describes these key developments:

- An internal SIDE architecture which allows a responsive Web browser-based user interface to interact with complex state information regarding an application that is under development or deployed. State-of-the-art technologies such as EmberJS are used to facilitate this.
- A Dynamic Smart Template (DST) concept, which leverages the TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) standard, and provides a pluggable architecture for adding custom forms and GUIs for application components, allowing SWITCH application GUIs to be completely decoupled from the Dashboard provided by the SIDE subsystem.

It has also been necessary to extend the TOSCA standard as an exchange mechanism between the SIDE, DRIP and ASAP subsystems, supporting exchange of non-functional requirements, configuration information, etc. This is described briefly here, but has already been described in detail in an earlier deliverable (D2.4).

In addition to the core SWITCH functionality, experimentation has taken place in the following areas, and experimental integration of these features into SIDE will be undertaken in order to assess their effectiveness in relation to development of the SWITCH Use Cases:

- Use of Pareto front techniques to help users to choose their preferred trade-off between NFR constraints, which will be integrated into the SIDE system
- Automatic creation of Qualitative Metadata Markers (QMMs) which can be used to inform the Quality of Service model available to the SIDE system and identify the parameters which have the greatest influence on an application's QoS for a given software component.

This deliverable explains these developments and experiments; it summarises the integration of SIDE with the other SWITCH components; and it describes how SIDE will be used, in conjunction with DRIP and ASAP, in the forthcoming phase when the SWITCH Use Cases will be fully developed and deployed using the SWITCH platform.

1 Introduction

The following deliverable has been prepared as part of the SWITCH project Work Package 2 (WP2) tasks 2.4 (Time critical software co-programming tool) and 2.5 (The front end for testing, monitoring and steering). Its primary aim is to describe the internal architecture of the SIDE sub-system, but because SIDE interfaces to all sub-systems, it also describes the interfaces between SIDE and ASAP, and between SIDE and DRIP. It therefore needs to be read in conjunction with other deliverables for a full picture of the overall SWITCH architecture. A secondary aim is to provide a more general progress report on the SIDE system and to provide an update on material introduced in Deliverable D2.4 [1]: how the design and implementation of the SIDE GUI has developed since the time of writing of D2.4 [1], and research work that has been undertaken, especially in regard to specification of Non-Functional Requirements (NFRs), and how this will be further incorporated in SIDE during the remainder of the project.

To fully describe the SIDE sub-system, we provide an overview of the internal architecture, the operations, role and position of SIDE within the SWITCH system. To this end, the SWITCH user's perspective is included (i.e. the perspective of someone wanting to develop, deploy and manage an application using SWITCH), describing features and the co-programming model supporting the entire application life-cycle of time-critical cloud applications and services.

A recent key development has been the introduction of a so-called Dynamic Smart Templates (DST) system. The goal of this system is to provide a pluggable architecture for adding custom forms, GUIs and their corresponding actions for application/software components. The SIDE interfaces for defining the application components, the automatic form generation and external service workflow integration completely decouple SWITCH application GUIs from the SIDE Dashboard. Developers can develop a GUI component and Web form independently and then integrate seamlessly with SIDE when it is ready for integration.

The DST concept is based on the Topology and Orchestration Specification for Cloud Applications (TOSCA [2]) standard, which presents a language and terms to describe a topology of cloud based web services, their components and relationships. The aim of this approach is to develop a functionality that supports auto generation of fully functional forms based on a TOSCA specification from application/software components. TOSCA is also used as the exchange mechanism between SWITCH components, having been extended to represent the non-functional requirements and other SWITCH-specific information which needs to be exchanged between these components.

The rest of the deliverable is structured as follows. Section 2 explains the role of the SIDE subsystem in supporting component-based and co-programming software engineering, and positions SIDE within the SWITCH environment. Section 3 presents the internal architecture and technical

implementation of the SIDE subsystem. The dynamic behaviour of the SIDE system is illustrated by a walk-through of the application life-cycle. We include a description of the Dynamic Smart Template (DST) concept and show how DSTs are used within the software engineering process. The RESTful APIs which underpin the technical integration of SIDE with the DRIP and ASAP subsystems are also explained. Section 4 concentrates on two specific aspects of non-functional requirements: how we can further exploit the TOSCA extensions to deal with NFRs which were described in D2.4 and are currently used in the SWITCH system, and a method for accumulating Qualitative Metadata Markers that can inform the specification and use of NFRs. TOSCA is already used in SWITCH to exchange information, including NFRs, but we explain in this section how we plan to extend SIDE on an experimental basis to help users select between conflicting NFRs using a Pareto front concept, and to allow monitoring data to inform the selection and use of NFRs. Section 5 presents the agenda for the next phase of the SWITCH project, from the SIDE perspective, especially in relation to how the usability of SIDE will be evaluated. Section 6 concludes with a summary of SIDE software functionality in the public releases of SWITCH, and of the key innovations.

2 Role of the SIDE Subsystem within the SWITCH platform

2.1 Component-based and co-programming software engineering in SIDE

For time critical applications, such as the SWITCH use cases provided by BEIA [3], MOG [4] and WT [4] it is crucial and necessary that they operate in real-time, which means that these cloud applications need to be designed in such a way that they offer reliability, robustness, availability, and dependability. Hence, a successful deployment of real-time systems depends greatly on low development costs, a short time-to-market, and a high degree of configurability and Quality of Service (QoS) and Quality of Experience (QoE). This is where Component-Based Software Development (CBDS) can be of a great advantage. The introduction of CBSD into real-time and embedded systems development offers significant benefits, including [5]:

- *Rapid development and deployment:* Designed and verified components can be reused across applications, offering a reduction in development cost and time-to-market.
- *Reduction of complexity:* Software for a specific application can be configured by reusing components chosen from an existing library.
- *Design evolution:* Components can be replaced or added to the system as needed, allowing for continuous software/system development and co-programming.
- *Increased reliability and maintainability:* Each component can be tested independently, making problems easier to isolate and fix. In addition, bug fixes can benefit many projects, and reused components tend to be more stable and mature than any new development.

- *Higher developer efficiency:* Each development team can focus on its own specialized task—domain experts can concentrate on creating components, and integration experts on assembling those components into products [5].

In designing the SWITCH IDE, CBDS principles were followed, motivated by the above reasons. Consequently, the SWITCH IDE provides a software developer the ability to build entire (multi-tier) cloud applications from scratch by creating components and placing them on the SIDE canvas. *Co-programming* in SWITCH – programming infrastructure and requirements at the same time as application functionality – enables the developer to link behavioural properties, such as QoS attributes/constraints and monitoring requirements (throughput, latency, upload time, etc.) and adaptation and infrastructure requirements that allow the user to expose exactly how and where they want SWITCH to support their application at run-time. By placing components and related properties on the SIDE canvas, the software engineer can create a graph representing the entire, fully functional application, ready for planning, deployment and execution in the cloud infrastructure [1].

Besides creating components, software developers can also define the monitoring metrics, QoS and QoE constraints linked to those metrics, and also reconfiguration policies that can link to metrics, constraints and components. During the development phase each element (application component, dataflow link, quality constraint) is placed on the GUI and linked to the component to which it relates.

In Figure 1, a screenshot of a SIDE GUI is presented. The menu on the left allows users to choose the application components which have been created, quality constraints, monitoring metrics drag and drop those components to the canvas, and link them together as required.

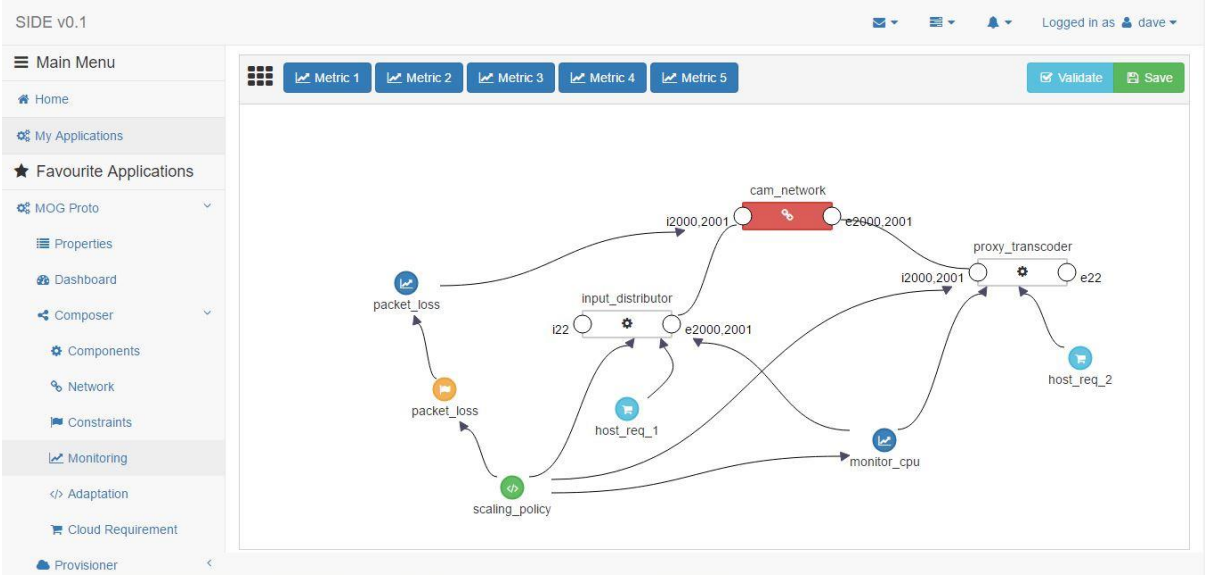

Figure 1: Creating application logic by connecting components and their QoS constraints and requirements on the SIDE GUI.

A more detailed description of a SIDE GUI which shows the phases of the application lifecycle along with a composition graph in a development and deployment phases are presented in D2.4 [1].

2.2 Relationship of SIDE to DRIP & ASAP subsystems

SIDE is the user-oriented Web interface that is provided to SWITCH users to enable them to interact with the rest of the components. It is built around the idea that simply defining the functionality of the system for designing distributed applications is no longer enough. Other requirements must be considered. The developer must make sure that an application not only performs the required function; an application must satisfy Quality of Service (QoS) constraints and, from the end-user's perspective, it must provide a satisfactory Quality of Experience (QoE). Although QoE and QoS are related, QoE is particularly challenging to measure. However, QoS and QoE are important metrics by which the success of a deployed application must be judged.

SIDE was designed with these constraints in mind. It enables the user to link up *application components*, each of which provides a certain element of functionality to the system. However, each component comes with a large set of additional tools that enable the user to specify the non-functional requirements, add monitoring or additional GUI functionalities.

Most of the functionality of the SIDE subsystem represents other parts of the SWITCH subsystems. For example,

- The DRIP Planner [6] [7] takes the developer's requirements into account to create a plan that returns the types of virtual infrastructure provided to the user and the position of components on these machines. The constraints are bound to an application component and are integrated inside TOSCA. (TOSCA is the de facto exchange format for SWITCH components, and more detail of how TOSCA has been adapted for use in SWITCH is provided in an earlier deliverable (D2.4).)
- The DRIP Provisioner [6] [7] can take this plan and provision machines on public or private clouds for the application with the right operating system and the interstitial network configured between machines in different data centers.
- The DRIP Deployment Agent [6] [7] can then set up the environment that is required for the system to operate it, bind several of machines together in a cluster, deploy specific services that are part of the SWITCH ecosystem and then deploy containers or other applications on the required machines.

The SIDE/DRIP integration is straightforward, because the steps from one step to another are well defined and cannot overlap or be taken in a different order. This makes the SIDE/DRIP integration

circular. When the application is deployed the developer can take the results into account and change some of the constraints of the system, re-plan the infrastructure and start a new deployment cycle.

The other part of SWITCH – ASAP – implements monitoring with alarm triggers, application modelling, and decision making [8] [9] [10]. Monitoring [8] and alarm triggers [9] are bound to the application components and can be changed during the creation of the applications. This information is also provided as part of the TOSCA description and submitted to the Monitoring component that uses it to set up the monitoring and alarms.

During the execution of the system, each component is monitored and should a component experience a violation, it can send an alarm to SIDE that reports this to the user so that he/she can take specific actions. These actions leverage the DRIP module to enable Start, Stop and Create actions on a container. The functionality is exposed as an example of the DST functionality (Section 3.3). Alternatively, the user can choose to defer the decision to adaptation module [10].

SIDE also provides the developer with an insight on requirements of the components he/she is using and provides hints on how he/she can provide a better performance of the application. This is accomplished by displaying the result of the modelling that is part of ASAP so that the user can have a better understanding what parameters affect the performance of the system (Section 4.2).

3 Technical implementation of SIDE

In this Section we commence by explaining the static architecture of the SIDE subsystem, and the technologies used. We then provide a SIDE-centric "walk-through" of a sample application (related to the MOG Use Case), from inception through to deployment. Dynamic Smart Templates (DST) are presented separately, and their role in a deployed system is explained. In the last two subsections we explain the RESTful APIs which underpin the integration of SIDE with the DRIP and the ASAP subsystems.

3.1 SIDE internal architecture and technologies

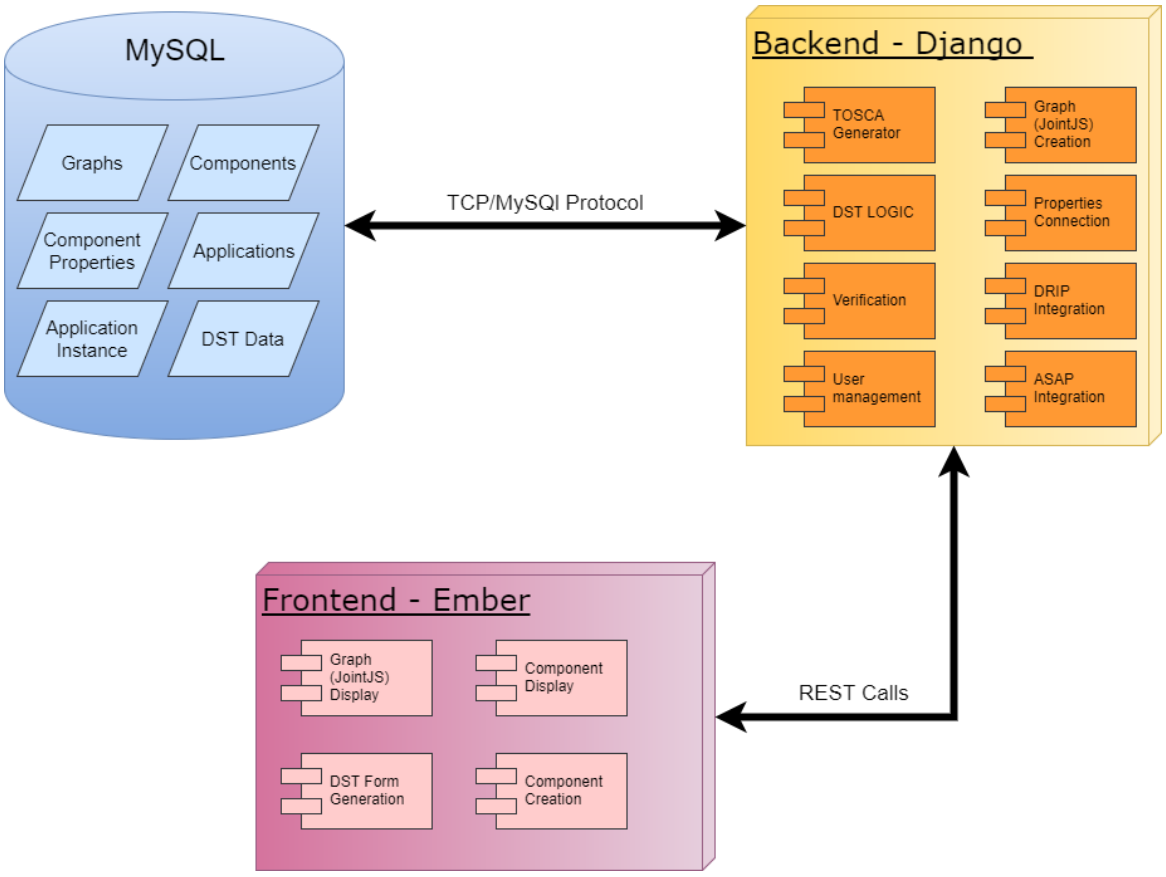

Figure 2 SIDE Architecture

The SIDE internal architecture is composed of three parts: the front-end is implemented using EmberJS; the back-end, which manages current SIDE state and provides APIs for communication with other SWITCH components; and the database, which stores persistent information needed by the front- and back-ends.

The front-end is implemented using EmberJS, with the central graph generation being facilitated by JointJS. The JointJS library takes JSON input that contains all the elements of the graph (the SWITCH Components, properties etc.) with their location and visual effects and the connections between them.

The rest of the communication between the front-end and back-end is achieved by REST calls with JSON as the exchange format. For instance, when a new component is dragged to the canvas a REST API *switchcomponentinstances* is called with the location and parameters of the component. The back-end creates a new instance of the component in the DB. If the component has associated parameters, instances of them are also created and linked to the component. The graph is updated with the new component instance and redrawn.

From the Django point of view these APIs are basic views that display or take JSON data. The rest of the APIs between the frontend and backend are shown in Table 1. All the APIs require Token Authentication, so they are user-specific. Missing from the table are the DST APIs, that are described in greater detail in section 3.3.

Table 1: Restful APIs between back-end and front-end.

Function name	method	Description
switchapps	GET POST	Set of methods deals with managing the application. It enables creating and updating from TOSCA, validation, planning, provisioning and deploying the application and enables storing the application to the Knowledge Base.
switchappinstances	GET	Returns the instances of the application.
switchcomponents	GET	Returns the appropriate components in the left-hand menu that can be added to the application.
switchcomponenttypes	GET	Returns the properties type components for component creation view.
switchcomponentinstances	GET POST	A set of methods dealing with creating and displaying components.
switchcomponentports	POST	Creates the component ports
switchdocuments	GET POST	Deals with creating and managing credentials for various systems.
switchservicelinks	GET POST	Creates and returns links between the components and manages component links
switchnotifications	GET POST	Enables the posting of notifications to the user. Functions as Alarm trigger endpoint.
switchartifacts	GET	Returns all the components available in the system
switchrepositories	GET	Returns the available repositories.
api/switchapps/{appID}/graph'	GET POST	Deals with displaying and storing the application graph
api/switchappinstances/{appInstanceID}/graph	GET POST	Deals with displaying and storing the application instance graph
api/switchcomponents/{componentID}/graph	GET POST	Deals with displaying and storing the component graph

The back-end stores all its data in a database, in our case this is MySQL. The creation of the data tables and communication is facilitated by Django. The communication between the back end and database is MySQL protocol over TCP. Django offers special functions to create the tables in the database. The tables of the database are shown in Figure 3.

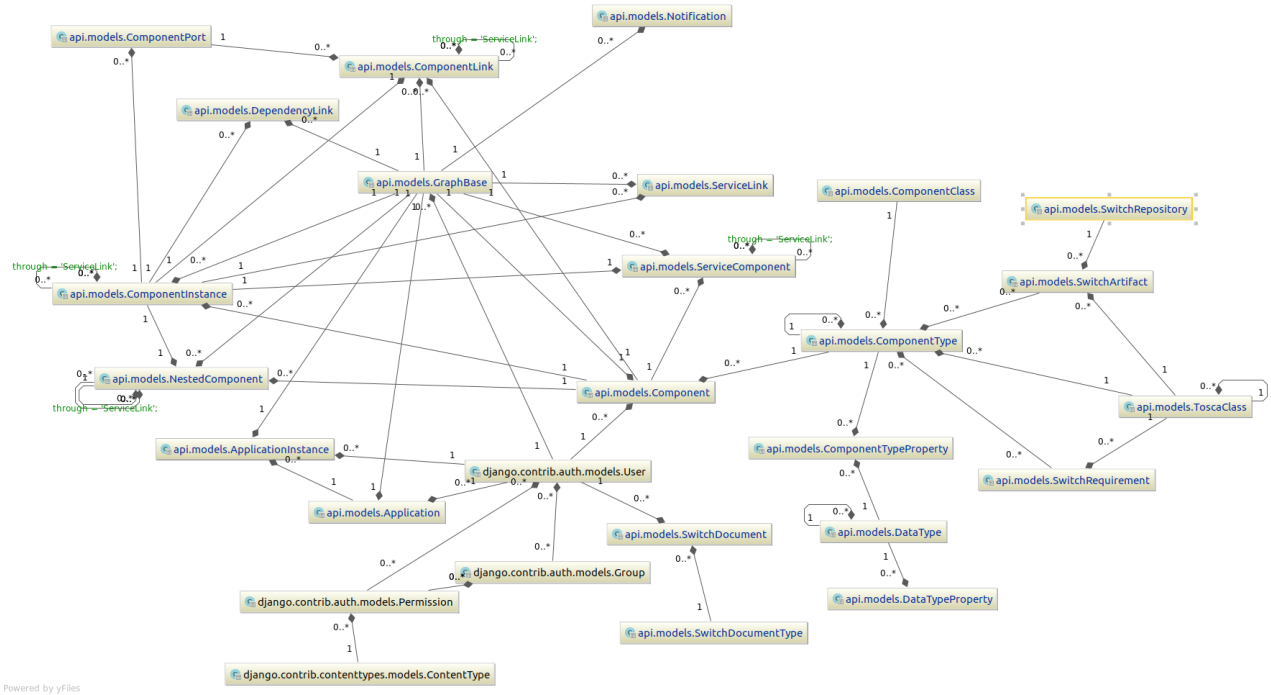

Figure 3 SIDE Database models

3.2 Dynamic behaviour and internal communication in SIDE: application walk-through

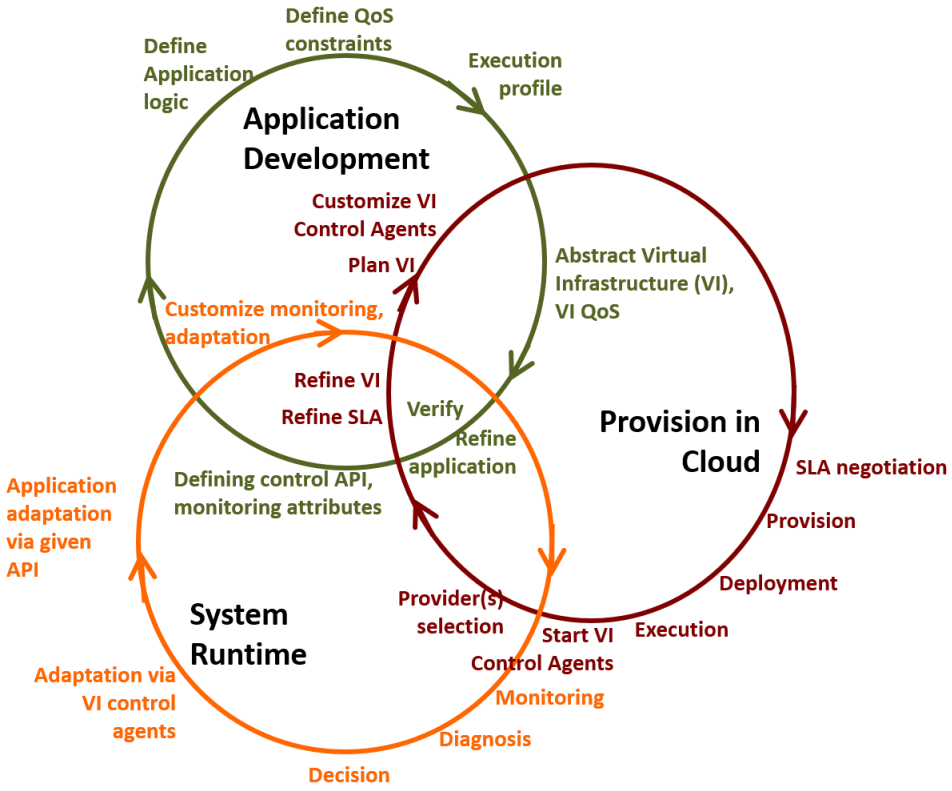

Figure 4 SWITCH application life-cycle: development, provisioning and runtime control.

This section provides a walk-through of the entire co-programming life-cycle shown in Figure 4, from the application developer's point of view. The SIDE GUI provides the user with the ability to drag and drop components onto a graph canvas, allowing the designer to build up a graph representing both the functional and network components that belong to an application, so that an application workflow can be defined. The user may also drag, drop and link Non-Functional Requirements and Quality-related properties. This allows the user to expose exactly how and where they want SWITCH to support their application at run-time, which provides a means of identifying QoS/QoE attributes, monitoring requirements, reconfiguration and adaptation abilities/requirements and infrastructure requirements. Figure 5 shows a list of graph Components, Properties and Provisioned Infrastructure and DST icons that are represented in the GUI. A *component* is the basic building block of an application, created by a third party. An example would be MOG's Proxy Transcoder. *Properties* are the co-programming archetypes available in SWITCH such as QoS constraints for the system; different hardware requirements (for example the amount of ram the user wants to specifically reserve for the system); monitoring parameters and alarm settings that enable the user to set up the monitoring system. *DST* is the UI creation component that enables rapid creation of interfaces for components.

(For a detailed explanation of DSTs, please refer to Section 3.3.) After the planning step, Provisioned Infrastructure icons are created on the canvas that show the VMs that will be used by the system (once the provisioning step has been performed).

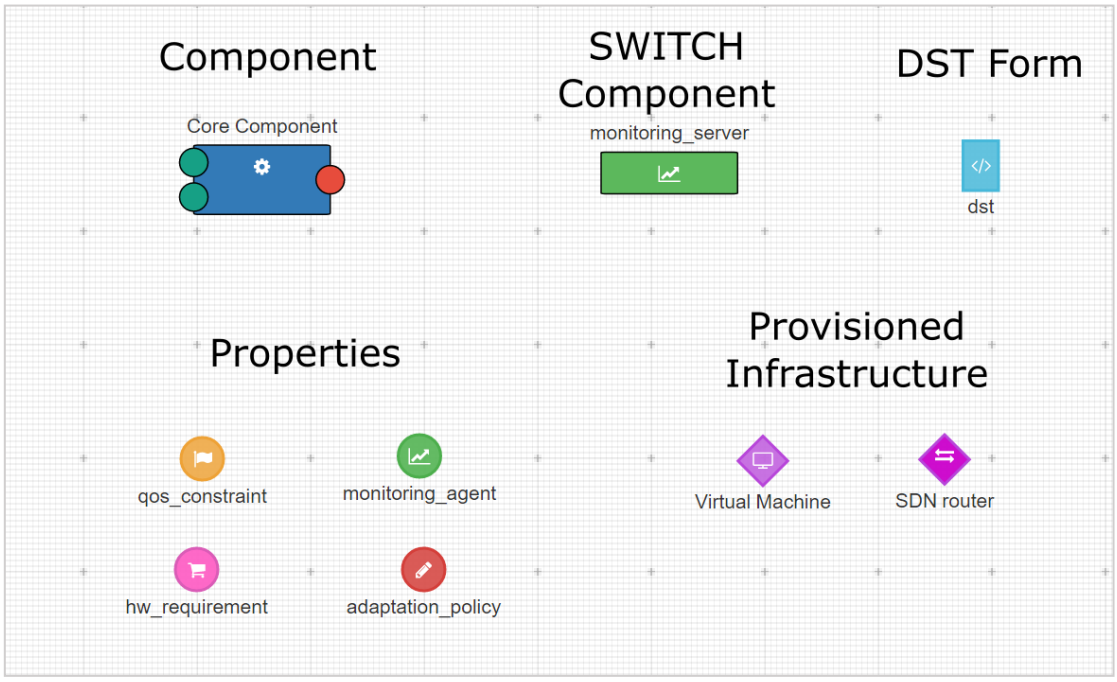

Figure 5 Graph elements present in SIDE used in application design and provisioning.

Figure 6 shows a screenshot of the initial interface of the SIDE GUI. The menu on the left allows users to find additional components to drag to the system. When a component is added to the Canvas the Properties associated with it are also displayed on the canvas. (To facilitate clarity of the image we removed some Properties.)

The image includes VMs and SDNs that were planned and provisioned by DRIP. Planning and provisioning needs the QoS Constraints component so it can complete its work. The monitoring server can be automatically added to the system if there is at least one monitoring agent present.

Finally, the "Director" Component is associated by the DST Service component that enables the user to access its functionality. In this example, it would be selecting the stream that would be displayed to the viewers of the stream.

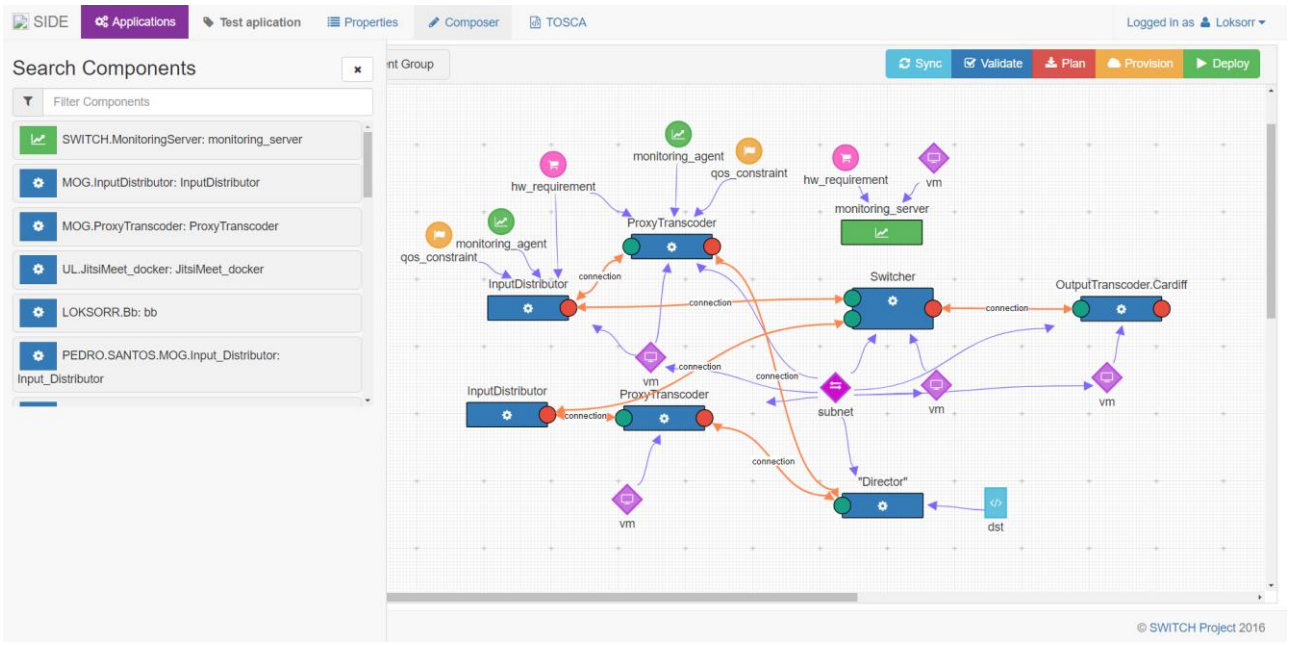

Figure 6 Defining application logic as a graph using the SIDE GUI.

The SIDE GUI has three main views that represent each of the three main phases of a SWITCH application: component development, application development with provisioning, and operation, as can be seen in Figure 7. Each of the main phases will render the application graph in a similar fashion, providing the user with a consistent understanding of how his or her application is configured, but with elements of the graph highlighted or manipulated in a way relevant to the current phase of development.

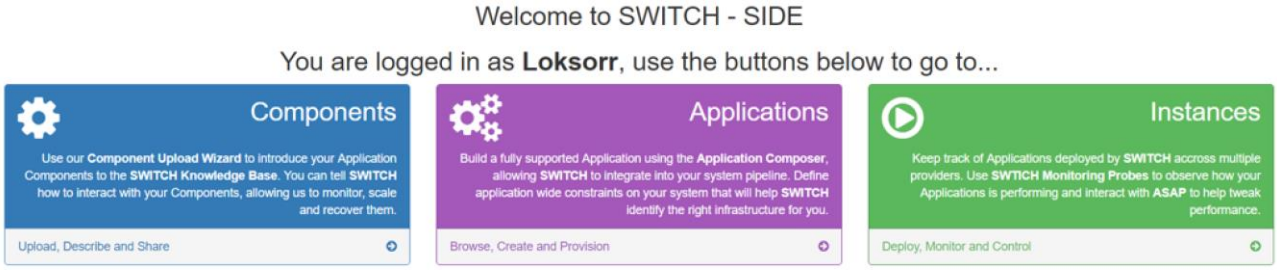

Figure 7 SWITCH workbench welcome screen.

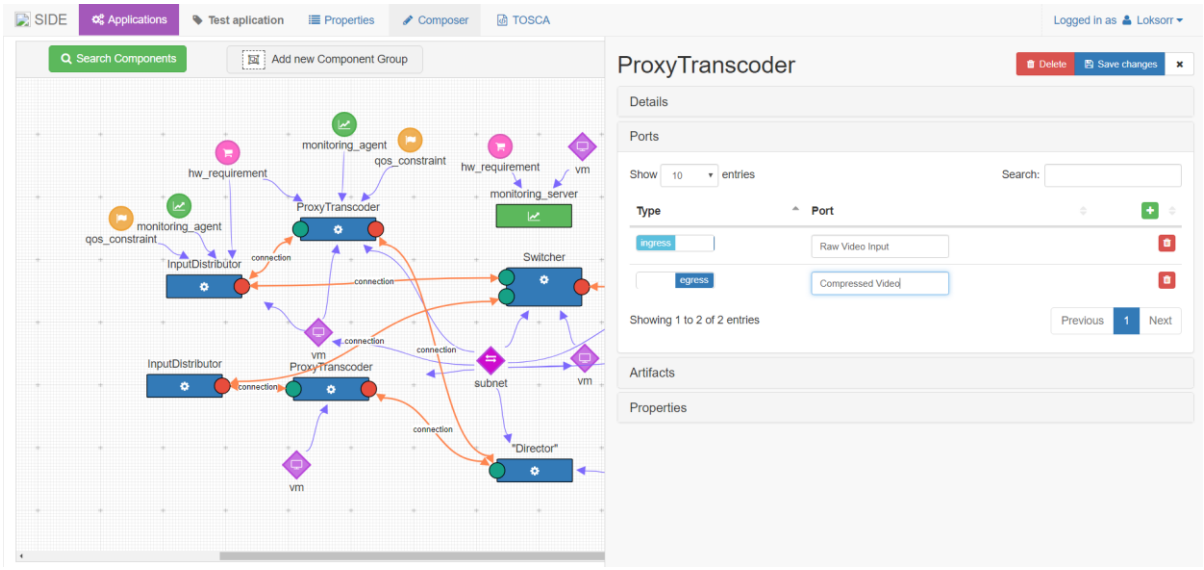

Figure 8 SIDE– Application view highlighting a sub-view for component description.

The three phases and their corresponding sub-views (for example the component details sub-view depicted in Figure 8) allow users to provide information in an iterative way, refining and extending the information provided in previous views by previous users, in such a way that at the end of each stage, SIDE has collected all the information necessary to be passed to DRIP or ASAP to initiate the following stage.

3.2.1 Component composition

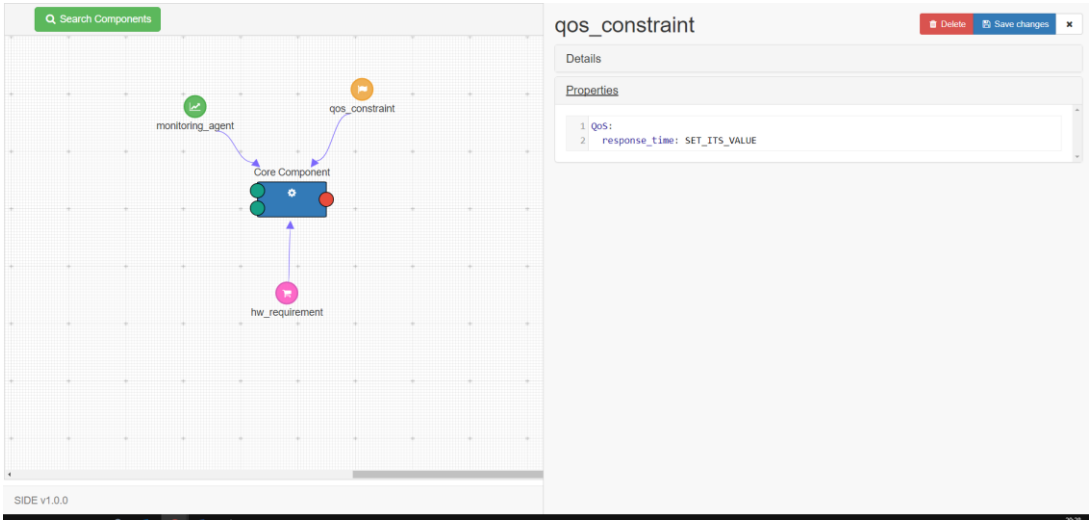

Figure 9 SIDE– Component composition view with QoS constraint parameters sub-view.

Component composition creates the components that can then be connected in the application view. Here the developer can specify some parameters of the system such as what metrics can be monitored

or what is the QoS constraint of the component. This part should be done by the component developer, as it requires in-depth understanding of the component and its features. The person describing the component is expected to know, for instance, what is the QoS metric or hardware constraints (Figure 9).

Part of the resulting YAML (which, in turn, is part of the TOSCA produced by the system) is shown below:

```
node_templates:
    "fdd7fa70-cf54-4fca-8b5b-0763bd3676a5":
      type: "Switch.nodes.MonitoringAgent"
      properties:
        agent_id: null
        probes:
          CPU_Probe:
            active: true
            metrics:
              cpuTotal_metric:
                thresholds:
                  threshold_0:
                    operator: greater_than
                    value: 80
                type: double
                name: cpuTotal
                unit: "%"
            static: false
            name: CPU
    "57ff2645-808b-4572-952b-3d2c74f5e984":
      artifacts:
        bb_image:
          type: "tosca.artifacts.Deployment.Image.Container.Docker"
          file: null
          repository: SWITCH_docker_hub
      requirements:
        - monitored_by: "fdd7fa70-cf54-4fca-8b5b-0763bd3676a5"
        - host:
            node_filter:
              capabilities:
                host:
                  cpu_frequency: "2 GHz"
                  mem_size: "2 GB"
                  num_cpus: 2
                  disk_size: "300 GB"
```

Note that not all the values are filled in, as these are the values of the system before deployment, so some values (such as agent_id) are null. The Monitoring agent and the Component are split into separate nodes because they are treated like that by the system. *"57ff2645-808b-4572-952b-3d2c74f5e984"* is the unique ID of a component. Similarly, *"fdd7fa70-cf54-4fca-8b5b-0763bd3676a5"* is the UID of the monitoring agent that can then be referenced by the main component. Here the component types are *"tosca.artifacts.Deployment.Image.Container.Docker"* for the main container image (defined in TOSCA standard) and *"Switch.nodes.MonitoringAgent"* for the monitoring agent (defined by the SWITCH system); these inform the system of what kind of node it is dealing with. Properties of the monitoring agent are specific to the monitoring system and are specified so as to conform to JCatascopia requirements. It will be noted that we include some NFRs for the main component – cpu frequency, etc.

3.2.2 Application composition

In the next step, the developer takes the components and connects them to create the final application as seen in Figure 6. This is the core of the SWITCH development. The user would search for the components he wants to use from the available components, as seen in Figure 8. The available components are queried from the database according to the permissions of the user. The user has access to public components and the components (s)he created.

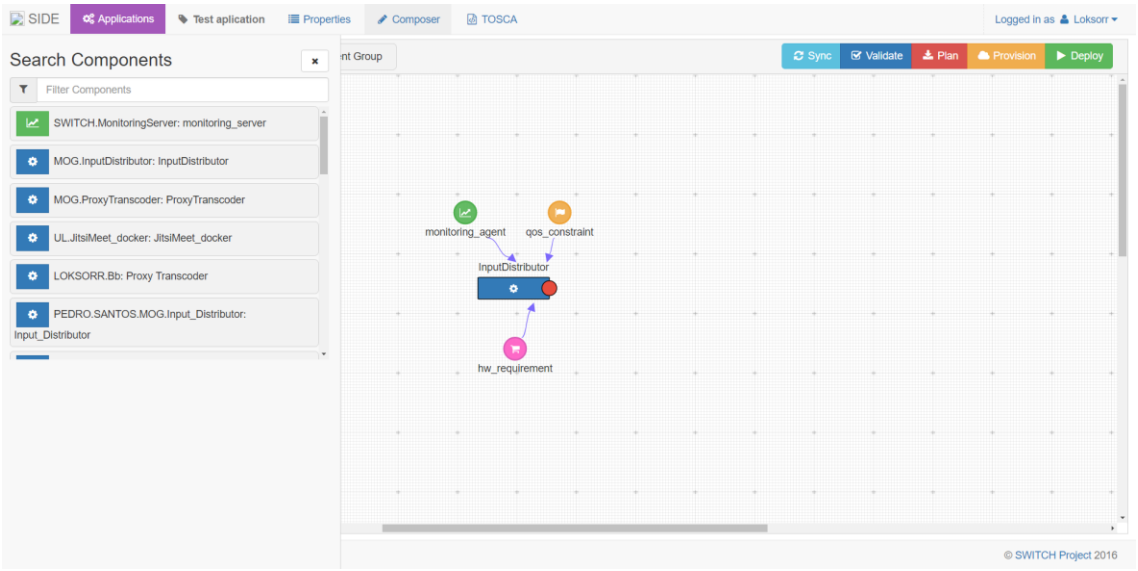

Figure 10 Adding InputDistributor component to application canvas.

After this (s)he can add additional components and connect them together (Figure 11). This creates an association of the components in the back-end that is defined in TOSCA. In order to connect the

components an egress must be defined for the component and another component must possess a compatible ingress. Connecting these components is subjected to validation based on their types.

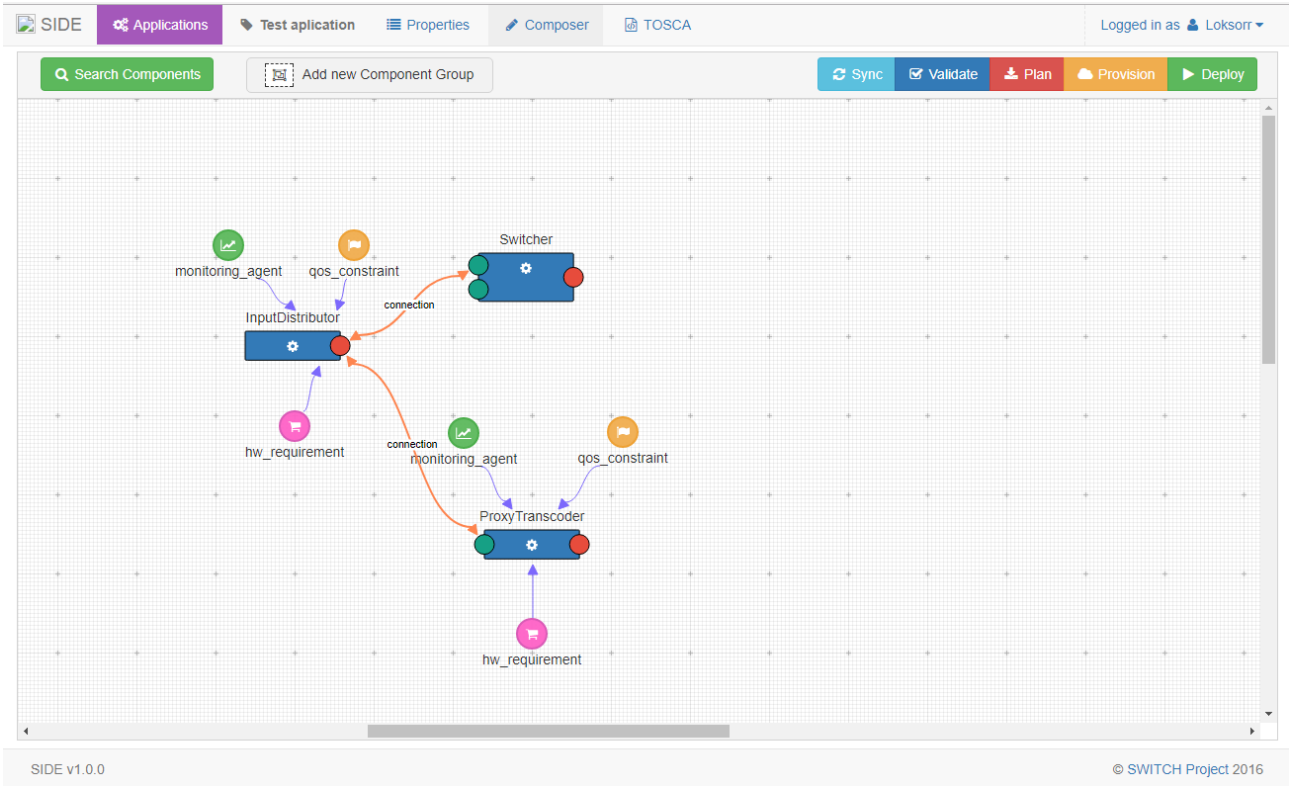

Figure 11 Connecting components.

He/she can update the constraints to suit his needs increasing their performance etc. (Figure 12) This is also the view that enables the developer to Verify, Plan, Provision and Deploy the application (see Section 3.4 for further details). Each step of the creation of the system can add additional components to the canvas. Validation can notice that the monitoring server is missing while there are monitoring probes and adds it. Planning adds VMs and their connections to the components. Provisioning adds additional information to the VM Sub-View such as the IPs of the machines, etc., until the final result is something similar to Figure 8.

Deployment is the final step of the application composition. After it is complete there is a new instance of the system available and ready for use. Further actions on the application can be observed in the Instance Management View.

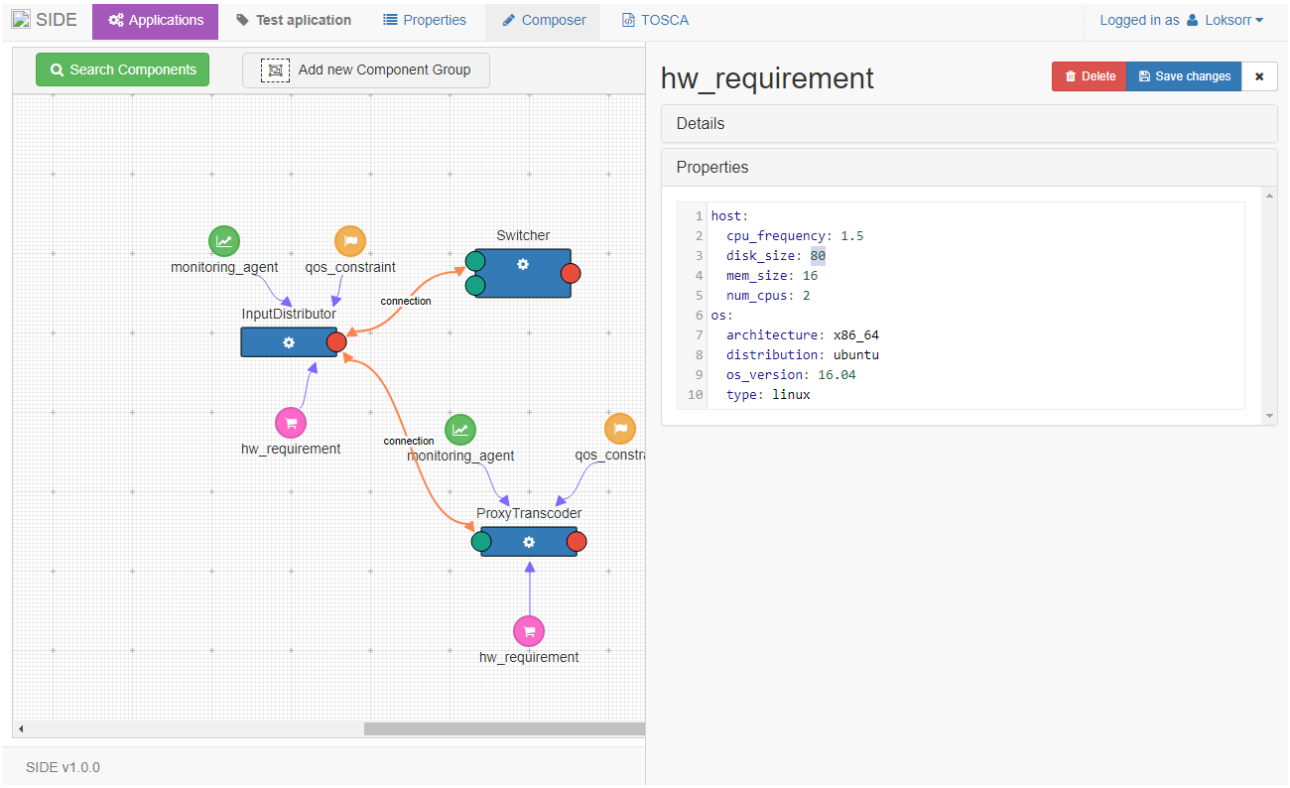

Figure 12 Changing properties of the component.

3.2.3 Instance management view

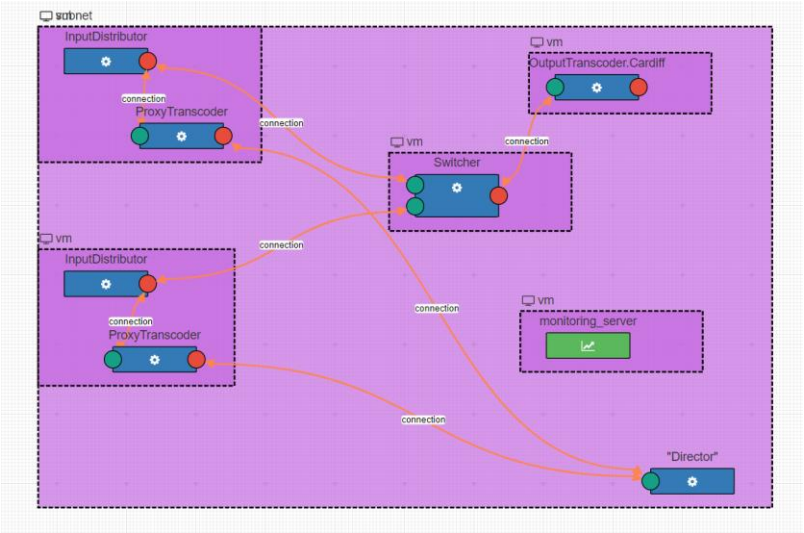

Figure 13 SIDE – instance management view with QoS constraint parameters sub-view.

The Instance management view (Figure 13) shows the state of the system. It enables the user to monitor the state of the system and access the DST functionalities that were developed. This view is important, as it enables control of multiple instances of the same application – for instance, multiple

videoconferencing services that service different users across the world with different requirements and network characteristics.

Monitoring, processing of alarms, and steering of a deployed application is facilitated by Dynamic Smart Templates, which we shall now cover in detail.

3.3 Dynamic Smart Templates

The goal of the Dynamic Smart Template (DST) system is to provide a pluggable architecture for adding custom forms and GUIs for application components. The SIDE interfaces for defining the application components, the automatic form generation, and external service workflow integration completely decouple SWITCH application GUIs from the Dashboard. Accordingly, developers can develop a GUI component and Web form independently and then integrate seamlessly with SIDE when it is ready for integration. Example integrations using this approach include:

- ASAP Integration 1: Network metrics monitoring form: Latency, throughput, upload time, upload speed, RTT, hops
- ASAP Integration 2: Container-based metrics monitoring form: CPU, Memory, reads/writes

Customized application GUIs can be integrated using this approach e.g.:

- the MOG Video Viewer

The aim of the approach is to develop a functionality that supports auto-generation of fully functional forms based on TOSCA/YAML definition from application components. The TOSCA/YAML definition must cover GUI components including their type and optionally default values, REST-based requests linked to components (usually on buttons) and be able to include external complex components such as monitoring graph component. An important aspect that must be covered is the data delegation between subsystems.

The input GUI is a form based on a YAML specification and resulting output GUIs is delivered from an external Web GUI, e.g. the monitoring Web interface, which is iFramed (another HTML form (web page) is embedded into Web site of the SIDE GUI) into the dashboard. Data can also come in natively into the dashboard and logged to a text area (like a console log). In the future, in the context of the SWITCH Use Cases, we may support other native GUI components and ingest real time data also. Figure 14 shows the general scheme of the system.

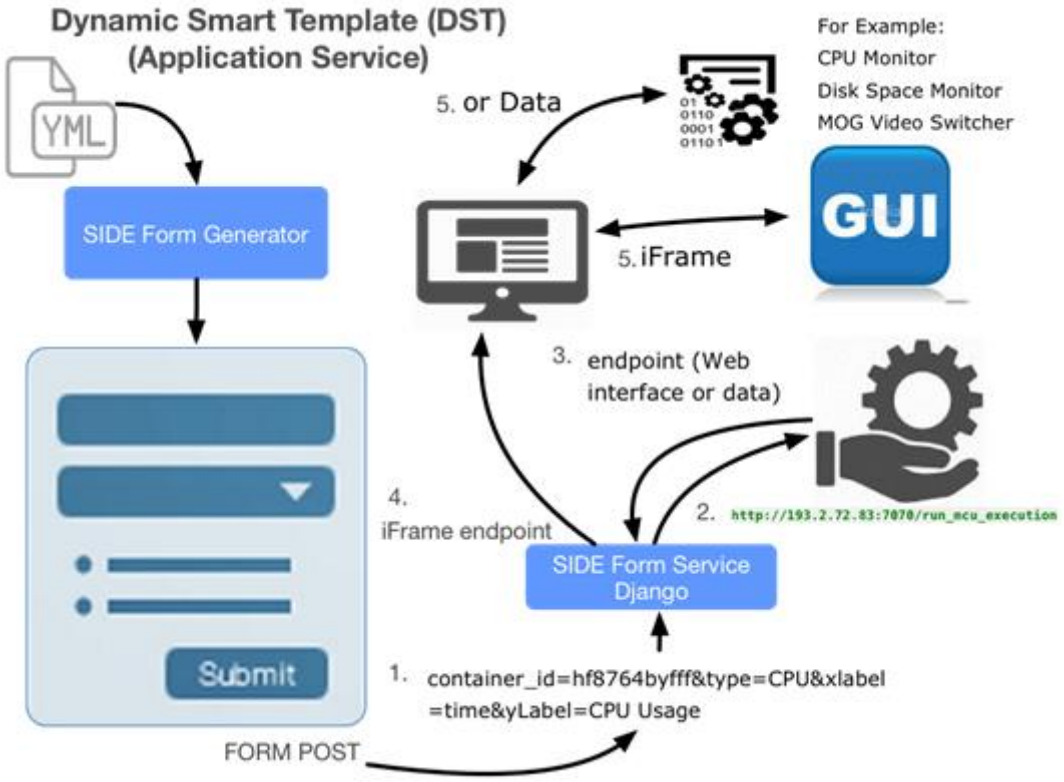

Figure 14: State diagram of the DST system.

The flow proceeds as follows:

- DST YAML (TOSCA) template component is written and stored in SIDE
- DST application component is registered (attached) to a component (i.e. a container)
- On running the application, the DST is translated using EmberJS into a Web Form.
- When the user fills in the form and hits “Submit”, SIDE sends FORM POST to the backend, which proxies it to the external service to configure the Web interface. It outputs an endpoint where the Web interface or data is.
- SIDE renders the GUI from the URL in an iFrame or outputs the data into a text area, depending on the configuration.

In order to facilitate this flow several REST APIs are required, presented in Figure 15.

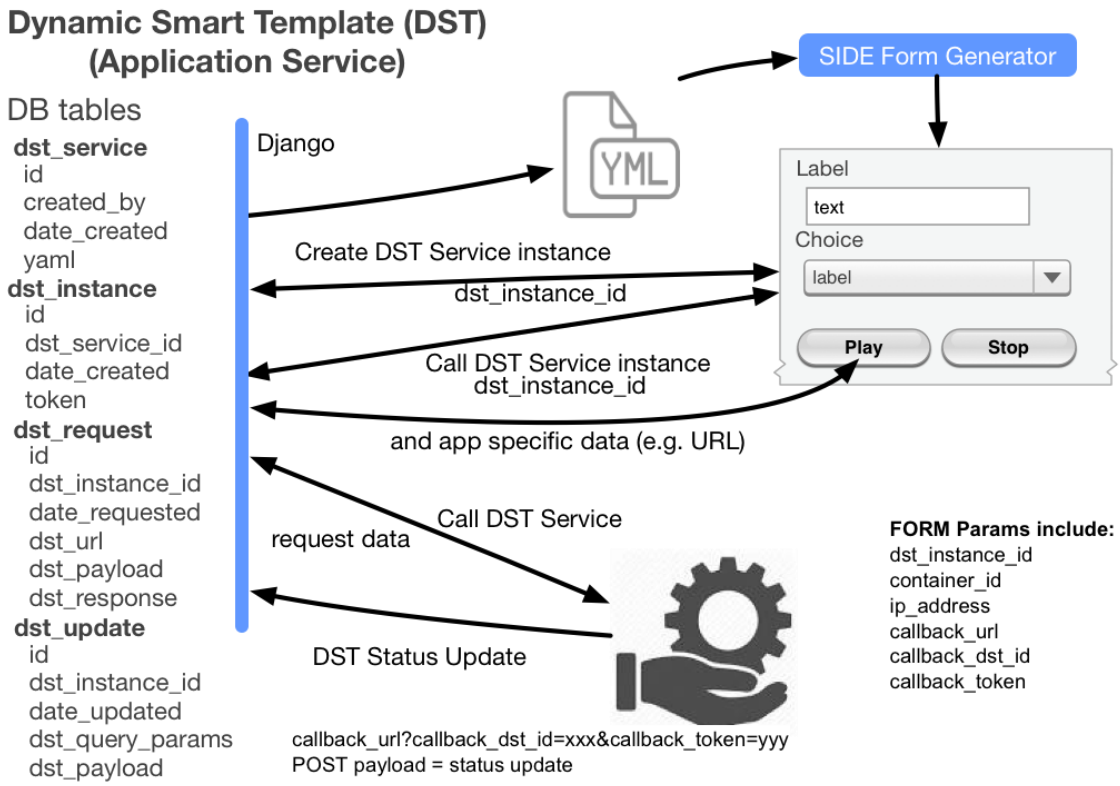

Figure 15: Schema of the APIs provided by the DST System.

3.3.1 The Dynamic Smart Templating (DST) APIs

Table 2 summarizes the services available by DST. The DST Services endpoint is the first step of the DST. It is the part of the GUI where the user defines the YAML that will be used to generate the forms and the endpoints of the REST Calls that will be used. The user specifies this in a special template component that is connected to the core component.

Table 2: Restful APIs between SIDE and ASAP subsystems.

Function name	method	Description
`dstService`	`GET`	Get the YAML that is needed to create the form
`dstInstance`	`GET`	Once a new Form is generated a new DST instance is instantiated to rack the plugin.
`dstRequest`	`POST`	Takes the data collected by the form and forwards it to the destination service.
`dstUpdate`	`GET` `PUT`	A service receives a token and an endpoint where it can publish updates. These updates can then be queried by the from and displayed.

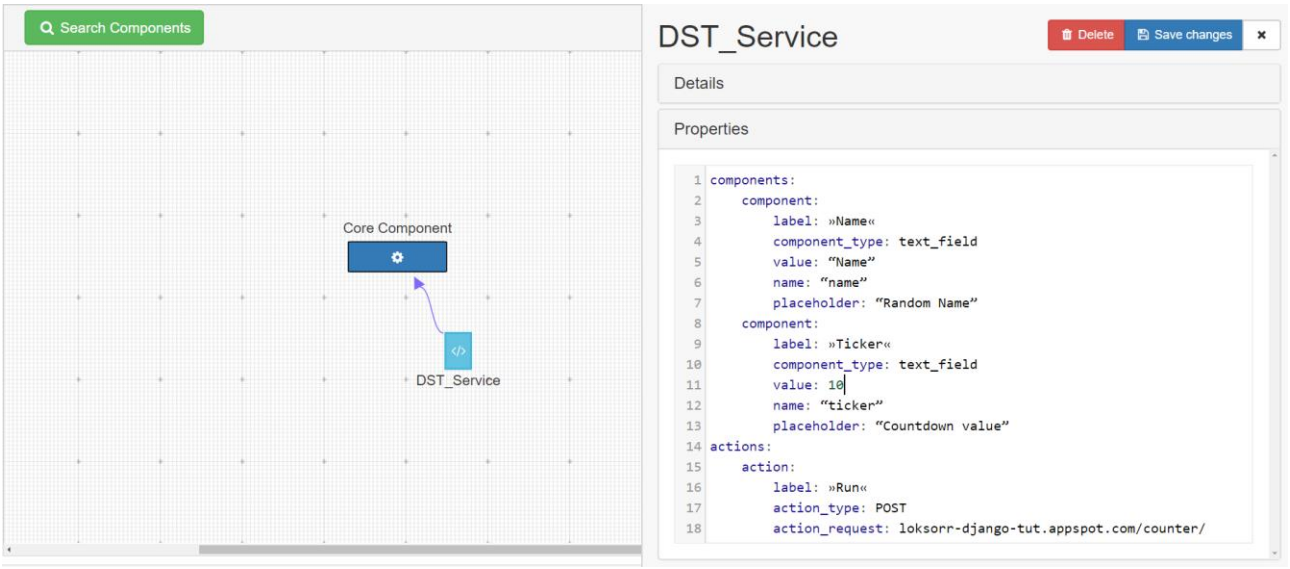

Figure 16: SIDE adding and editing the DST_service.

The YAML must follow certain specifications for it to work. Each component must be defined by a label, type, value (Default Value), name (name that will be parsed by the service) and optionally a placeholder that displays a hint to the user. Actions must contain the label, type and request URL.

```
components:
      component:
            label: »Name«
            component_type: text_field
            value: "Name"
            name: name
            placeholder: "Random Name"
      component:
            label: »Ticker«
            component_type: text_field
            value: 10
            name: ticker
            placeholder: "Countdown value"
      component:
            label: »Console«
            component_type: text_box
            value: ""
            name: conslole
            placeholder: ""
actions:
      action:
            label: »Run«
            action_type: POST
            action_request: loksorr-django-tut.appspot.com/counter/
            output_component: console
```

Each time a user creates a new instance of the form, a new entry is created in the dst_instance DB, using the dst_instance identifier table to track the plugin. Once an instance is created a user can interact with the DST form and customize the plugin.

The DST_Request API is a REST call that takes the data collected by the form and forwards it to the destination URL. It stores the response of the call (assumed to be a string containing the URL to the developed webpage for the request.

```
{ "dst_url" :  "loksorr-django-tut.appspot.com/counter/",
  "dst_instance_id" :  "From another API"
  "action_type" :  POST
  "dst_payload" :  {
              "name" : "matej",
              "ticker" : 200,
              "container_id" : "Bla",
              "ip_address" : " Bla",
              "callback_dst_id" : "Bla",
              "callback_token" : " Bla",
              "callback_url" : " Bla",
              "dst_instance_id" : 1
            }
}
```

The Update API has two endpoints. This is the POST endpoint where the external service can submit its results based on the *callback_dst_id*, *callback_token* and *callback_url* values of the system. This

result is then polled by the form and displayed in the component denoted by the *output_component* tag.

3.3.2 The Dynamic Smart Templating Form Generator

As can be seen in Figure 13, the user defines YAML for a specific component by double clicking the DST component (light blue rectangular on the canvas). After that, a right menu slides out and shows the additional properties that can be done for the specific component. In a text area below *properties* user can paste YAML and by clicking the button *Generate Form* (see Figure 14) a Form Template is generated out of a YAML (see Figure 15).

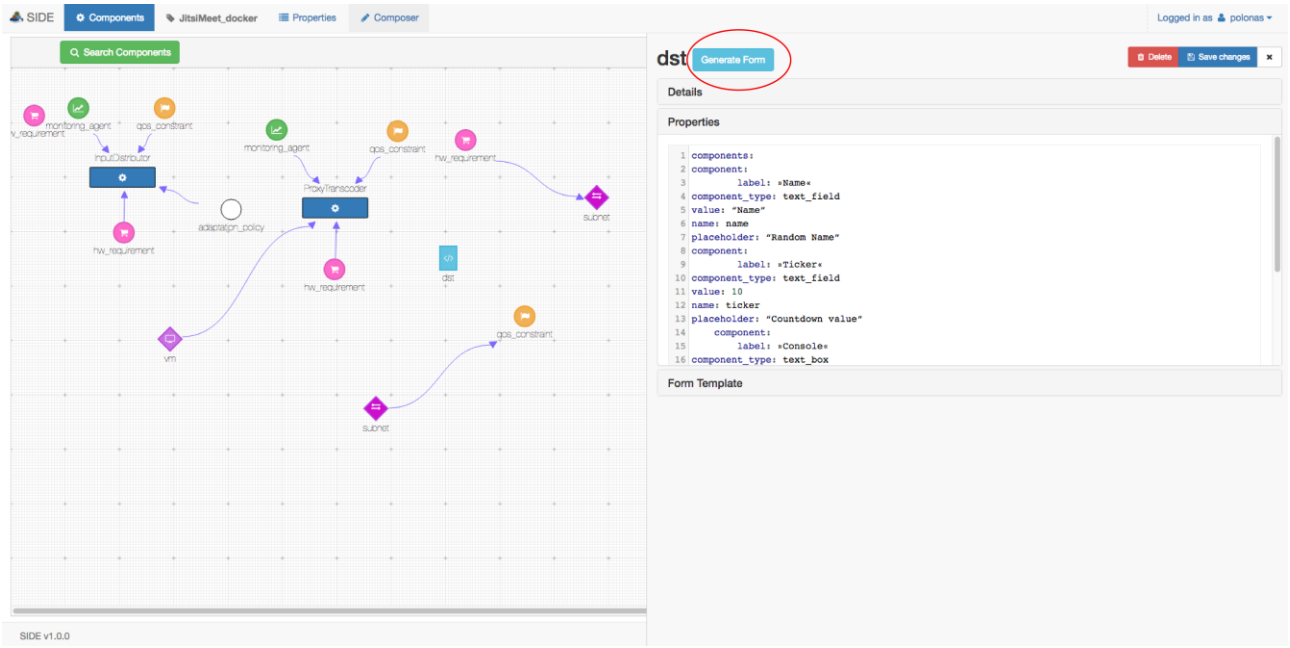

Figure 17: By double clicking the light blue DST component on the canvas a right menu slides out with possibility for the user to insert YAML into text area below properties.

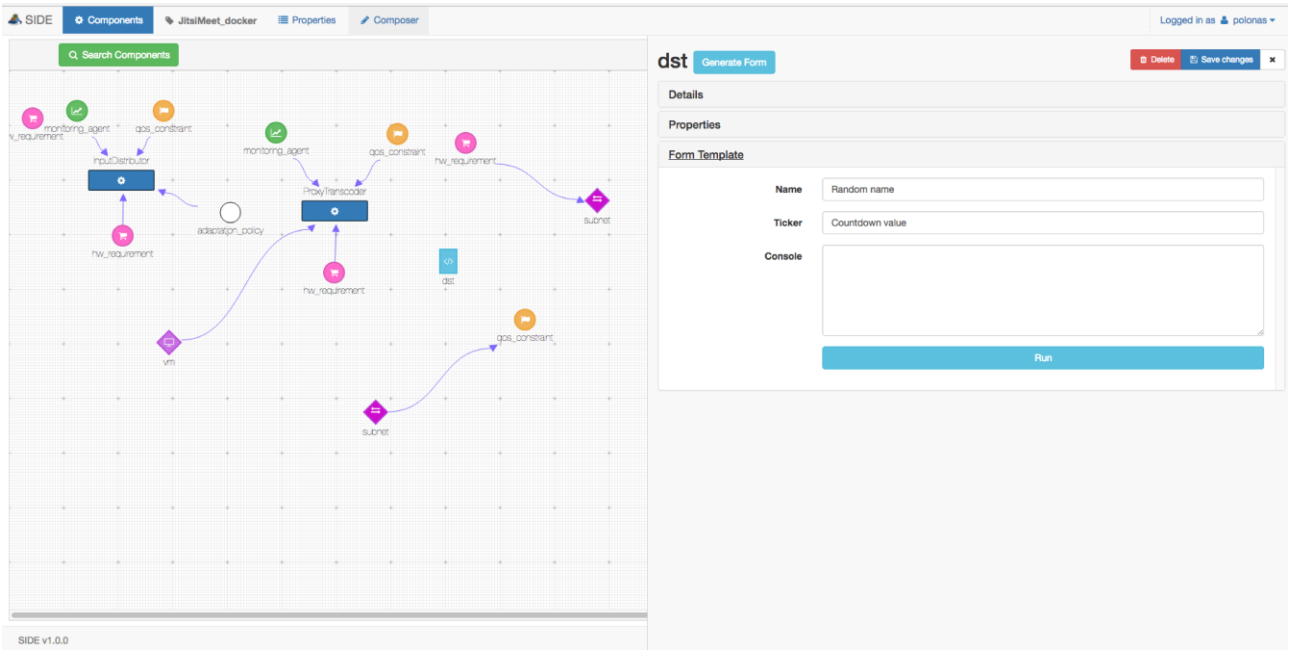

Figure 18: By pressing the button Generate Form a Form Template on the right menu opens that is generated from a YAML representation

3.4 Integration of SIDE and DRIP subsystems

DRIP provides the planning, provisioning and deployment functionality to SWITCH and by extension to the user via SIDE. The exchange format between DRIP and SIDE is TOSCA, that contains the current state of the system at each step. The steps do not overlap, as can be seen from the sequence diagram in Figure 19.

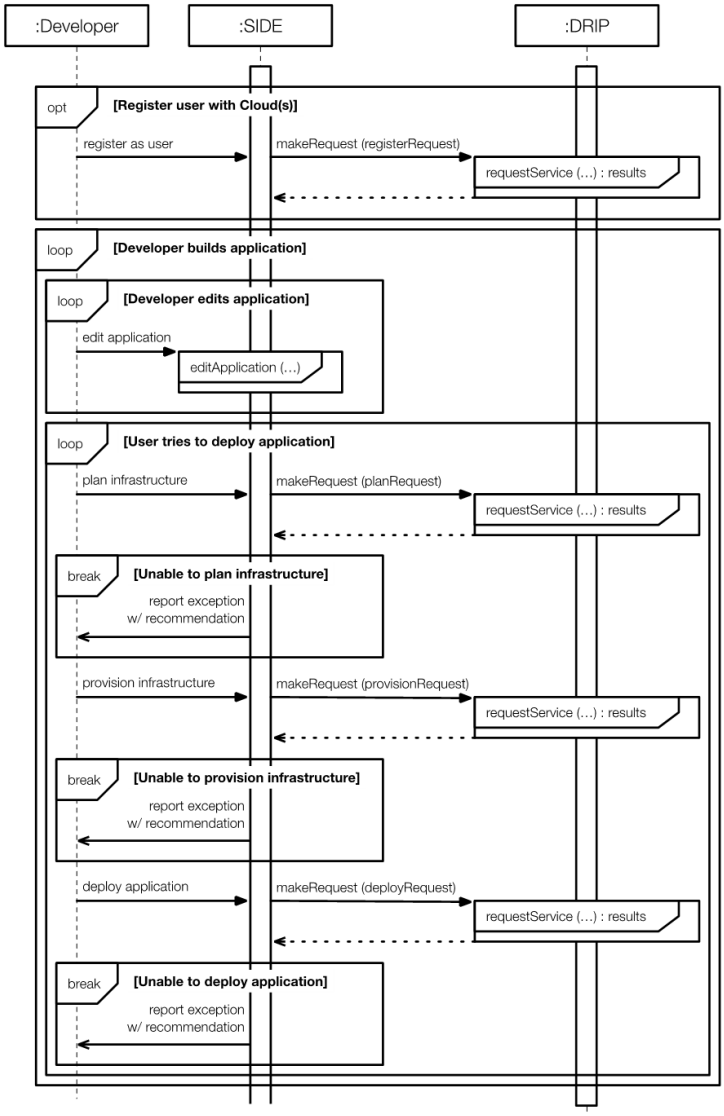

Figure 19 SIDE– DRIP sequence diagram.

This section contains a list with brief description of Restful APIs that connect SIDE and DRIP subsystems. Table 3 contains the list of Restful APIs among SIDE and DRIP. For each API the following parameters are listed: function name, short description, HTTP method (e.g. GET, PUT),

input and output parameters. A more detailed explanation of the system can be found in the respective deliverables and in the Integration plan document delivered to the Commission [11] .

Table 3: RESTful APIs between SIDE and DRIP subsystems.

Function name	**methods**	**Description**
AnsibleOutputController	GET DELETE GET GET DELETE GET	This controller is responsible for showing the output from ansible executions.
BenchmarkController	GET DELETE GET DELETE GET	This controller is responsible for handling cloud benchmark tests like sysbench.
CloudConfigurationController0	POST POST	This controller is responsible for handling cloud credentials used by the provisoner to request for resources (VMs).
CloudCredentialsController	POST DELETE GET GET DELETE GET POST	This controller is responsible for handling CloudCredentials. CloudCredentials are a representation of the credentials that are used by the provisoner to request for resources (VMs).
ConfigurationController	DELETE GET POST DELETE GET	This controller is responsible for storing PlayBook descriptions that can be used by the planner.
DeployController	DELETE POST GET GET DELETE GET	This controller is responsible for deploying a cluster on provisioned resources.
DeployController0	POST POST	This controller is responsible for deploying a cluster on provisioned resources.

Function name	methods	Description
KeyPairController	POST DELETE GET GET DELETE GET	This controller is responsible for handling user public keys. These keys can be used by the provisioner to allow the user to login to the VMs from the machine the keys correspond to.
PlannerController	DELETE GET POST DELETE GET GET POST GET POST	This controller is responsible for planning the type of resources to be provisioned based on a TOSCA description.
PlannerController0	POST	This controller is responsible for planning the type of resources to be provisioned based on a TOSCA description.
ProvisionController	DELETE GET POST GET DELETE GET	This controller is responsible for obtaining resources from cloud providers based the plan generated by the planner.
ProvisionController0	POST POST	This controller is responsible for obtaining resources from cloud providers based the plan generated by the planner and uploaded by the user
ScriptController	POST DELETE GET GET POST DELETE GET	This controller is responsible for handling user scripts. These scripts can be used by the provisioner to run on the created VMs.
ToscaController	DELETE GET POST POST DELETE GET	This controller is responsible for storing TOSCA descriptions that can be used by the planner.

Function name	methods	Description
UserController	GET GET POST POST DELETE GET	This controller is responsible for handling user accounts.
UserController0	POST	This controller is responsible for handling user accounts.
UserPublicKeysController0	POST	This controller is responsible for handling user public keys. These keys can be used by the provisioner to allow the user to login to the VMs from the machine the keys correspond to.
UserScriptController0	POST	This controller is responsible for handling user scripts. These scripts can be used by the provisoner to run on the created VMs.

3.5 Integration of SIDE and ASAP Subsystem

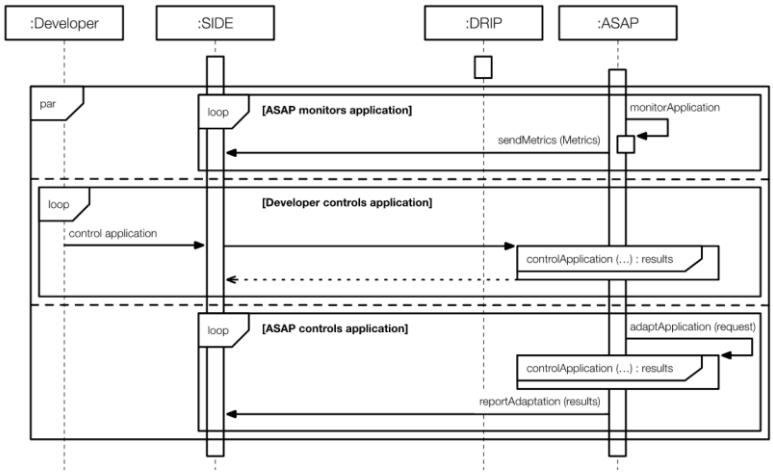

Figure 20 SIDE– DRIP sequence diagram.

ASAP by necessity is highly coupled to the application. As such the interaction between ASAP and SIDE is more set-up and reporting-based. The SIDE enables the user to define the parameters for the monitoring and Alarm triggering components and receives messages form ASAP on reconfiguration that were done (i.e. location of the services). A list of Restful APIs between SIDE and ASAP is presented in Table 4.

Table 4: Restful APIs between SIDE and ASAP subsystems.

Function name	method	Description
`getMonitoringInfo`	`GET`	Get basic monitoring information.
`getAvailableMonitoringMetrics`	`GET`	Get all available monitoring metrics.
`applyMonitoringMetric`	`POST`	Start monitoring the selected monitoring metric.
`getLastMetricValue`	`GET`	Get last metric value to be shown on a graph.
`stopSinglePod`	`GET`	Stop selected Kubernetes pod instance on demand.
`getKubernetesPods`	`GET`	Get all running Kubernetes pods.
`getAvailableAsapClusters`	`GET`	Get the list of all available and running ASAP cloud clusters.
`getAlarmTriggerInput`	`GET`	Get the input (YAML file) for the Alarm-Trigger component.

4 Non-functional requirements in SWITCH SIDE

SIDE as part of its development features some novel concepts. We have already described the Dynamic Smart Template (DST) concept, which decouples application/component-specific user interfaces from the SIDE GUI. Also, in D2.4 we explained how we have extended TOSCA for use in SWITCH, in particular so that it can handle Non-Functional Requirements, and this has been further illustrated in the walk-through presented in the current deliverable. In this Section we mention some current experimental work that takes forward the NFR-related aspects already incorporated into the SWITCH platform. We discuss some experimental Pareto front-related work which we plan to integrate into SIDE before the end of the SWITCH project, to assist the user in selecting NFRs. We also discuss experimentation that has been undertaken to create QoS models based on Qualitative Metadata Markers for software components, generated while the components are deployed. Again, this is something which we plan to integrate into SIDE before the SWITCH project ends. Initial publications relating to both these experiments have been produced ([12] [13]).

4.1 Assisting the user in choosing among conflicting NFRs

SWITCH uses the TOSCA standard for the application modelling because it is a widely-used standard, and because it supports the description of the application logic, the ability to use virtual images and containers as implementation artefacts and enables the description of QoS through

policies and management of the entire application lifecycle including continuous deployment, integration, monitoring and adaptation [2]. However, besides topology and management aspects, we also use TOSCA in SWITCH as a means of exchanging Non-Functional Requirements (NFRs) and other quality constraints expressed by application developers in the design phase, representing generic requirements such as throughput, latency and memory, and also application-specific requirements such as minimum frame rate. It should be noted that this is a significant development; in the literature it is noted that there is a lack of definition of Non-Functional Requirements and of quality constrains such as those we have needed to address in SWITCH [14] [15].

Recently we have described a novel approach that allows to software engineer to study conflicting NFRs trade-off possibilities for each application tier or software component during the development of multi-tier cloud applications. The process is managed by software engineer who is the decision maker. As an optimization method, we have used the Pareto front decision making method [12]. Our future work (to be undertaken during the remainder of the project) is in three parts. Firstly, we will extend the SIDE system on an experimental basis to allow users to visualise and explore this Pareto front in a manner that is integrated with the SIDE IDE. Secondly, we will further extend our use of TOSCA so that the conflicting constraints can be represented and trade-offs can be computed automatically from conflicting NFRs written in TOSCA policies, so they can be used in the automated deployment to a cloud environment. Thirdly, we will perform experimentation with the providers of the SWITCH Use Cases in order to assess the usability and utility of this approach.

4.2 Quality of Service Models for informed NFR monitoring

The SWITCH SIDE sub-system as a software engineering tool allows efficient creation of cloud native applications and micro-services at each stage of their lifecycle. On the SIDE canvas, a software engineer can create fully operational cloud applications and micro-services by defining software components (e.g. micro-services, software assets, etc.). In some cases, an individual component can represent a fully operational stand-alone simple application that can be deployed to the cloud/edge infrastructure. Furthermore, software components can be suitably combined to form larger, more extended and fully operational cloud applications.

A question that the developer might have some difficulty answering is: *how can one determine which are the most important Quality of Service parameters, and how do they actually affect performance?* Our second area of research is to address this issue with a novel QoS modelling approach that additionally equips software components with Qualitative Metadata Markers (QMM). These markers are part of the adaptation models created in ASAP based on the monitoring data. More information about these models can be found in respective deliverables [10]. The concept assumes that component providers will generate these models. To create QMMs, software components or micro-services should be subject to preliminary testing in a native cloud environment. Those preliminary tests should consider necessary constraints of a component (e.g. minimum CPU and memory needed), their NFRs

(e.g. level of importance of availability and security needed) and monitoring metrics, such as desired latency, throughput, upload speed, upload time and so on. QMMs present probabilities relating to the influence of these metrics or NFRs on the QoS of the software component and QoE of the end user.

As the application is deployed the software engineer can oversee an application's life-cycle in a concise manner and change the characteristics of the system if necessary. By continuously monitoring deployed software components, in our approach there is a continuous supply of probability values relating to which parameters have the biggest influence on the QoS of the deployed software component [13]. The concept is presented in Figure 21.

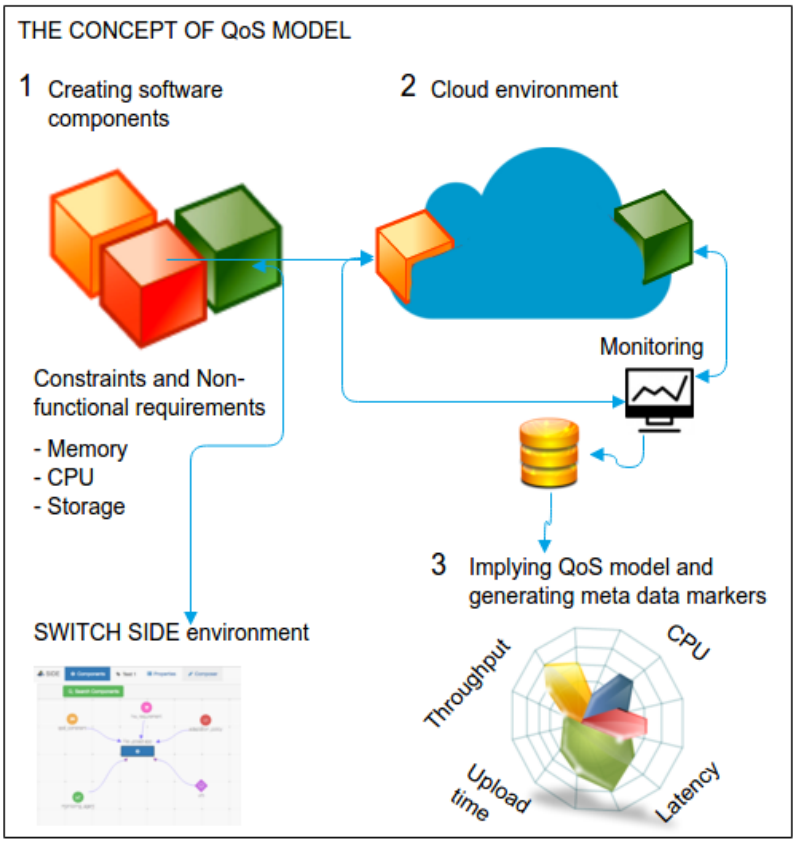

Figure 21: Steps of our augmented deployment system. (1) On SWITCH SIDE software components are created with defined constraints and NFRs; (2) Software components are deployed to a cloud environment and monitored; (3) monitoring data and constraints are used by a QoS model that generates Qualitative Metadata Markers.

Our approach has three steps:

Step 1: Creating a software component / micro-service: The software engineer can create software components from scratch using the SWITCH SIDE environment.

Step 2: Defining constraints, collecting monitoring metrics for a specific component: Here the minimal constraints and NFRs for a specific software component are defined. In order to obtain monitoring metrics, the software component is preliminarily deployed and monitored in the cloud environment.

Step 3: Creating Qualitative Metadata Markers: Data gathered in Step 2 is fed into a Model Maker (MM) component. The MM creates the differentials for all pairs of collected metrics. It then compares the amount of positive correlations (i.e. increasing the metric's value increases the QoS) and negative correlations. The QMM of the metric is the number of positive correlations minus the number of negative correlations. The result is treated as the probability value that determines the parameters (in our case monitoring metrics, such as throughput or latency or software component constraints) which have the greatest influence on the application's QoS for the software component [13].

As with the Pareto front techniques, generation and use of QMMs has not yet been fully integrated into SWITCH; the current status is that we have demonstrated the feasibility of the technique and it is presented in a workshop paper. Future work in this area to be undertaken during the remainder of this project is again in three parts. Firstly, we will extend the SIDE system on an experimental basis to allow users to inspect QMMs and use them to inform specification of QoS requirements. Secondly, we will establish how these markers can be stored and exchanged using the TOSCA orchestration standard. Thirdly, we will perform experimentation with the providers of the SWITCH Use Cases in order to assess the effectiveness of the QMM capture techniques, and their usability and utility as a source of information to inform the choice of QoS requirements.

5 Agenda for the final phase of SWITCH

The most important remaining priorities in relation to SIDE are as follows:

- Use of the SWITCH system (including SIDE) to support the entire life cycle of the SWITCH Use Cases entirely within the SWITCH platform
- Addressing any bugs, integration problems, etc., which come to light during these experiments
- Making minor adjustments to SIDE if necessary in order better to support the SWITCH Use Cases (for example, if it turns out to be unexpectedly difficult to specify a particular QoS requirement or monitor the relevant components)
- Integration into an experimental version of SIDE of the NFR-related developments described in Section 4, and engagement with the SWITCH community to evaluate their effectiveness
- Usability evaluation of SWITCH, as perceived through the SIDE GUI, as a means of supporting the co-programming concept in the context of time-critical cloud applications

This last point is critical, as it is effectively evaluating how successful the project partners have been in implementing the fundamental SWITCH concept. In the remainder of the present section we shall discuss approaches that we shall adopt in order to evaluate the usability of SIDE (and hence of SWITCH).

5.1 Usability evaluation approach

There are many things that should be considered when developing system or software. However, one of the most important aspects is the usability of a system. This can be considered to fall under the following headings: learnability, efficiency, satisfaction and errors. Learnability is a degree that shows how easy a new user can accomplish tasks the first time he uses the software or system. Efficiency is how quickly users can complete tasks after they are familiar with its use. Satisfaction is whether users enjoy the design of the software; and errors refers to the number of errors users make when they use the software, the severity of the errors and how easy they are to recover from.

When testing the SWITCH SIDE software engineering tool for usability we will first compare SIDE environment with similar software engineering tools that are used for creation of micro-services and cloud applications such as Juju [16] [17] and Fabric8 [18]. Juju is an open source universal component-based graphical modelling tool for service oriented architectures and application deployments. It offers also sets of predefined software assets and relationships and configurations among them that come with a knowledge of how to properly deploy and configure selected services in the cloud and relationships among them [17]. Fabric8 is an open source platform that is based on Docker as virtualization, and Kubernetes as orchestration technology. Fabric8 provides a developer console for creating, building and deploying micro-services and run and manage them with continuous improvement [18]. The comparison will be at an abstract level, using human processor models such as GOMS [19].

In contrast, usability testing is a technique used to evaluate a product by testing on users. In order to perform usability tests, we will first create some scenarios whereby users will perform a list of tasks, such as (1) creating a software component on a SWITCH GUI, (2) adding QoS constraints, Non-Functional Requirements, DSTs, (3) creating the composition, the provisioning graph, etc., and observing the dashboard with monitoring services and notifications, (4) deployment to the cloud environment and associated tasks. When users perform tasks, we will observe them, taking notes.

We will use a range of usability testing methods, including as many of the following as proves feasible in the available time:

- ***Hallway Testing*:** using random people to test the website rather than people who are trained for testing such environments.

- ***Expert Review*:** An expert in the field will evaluate the usability of the software engineering tool. Sometimes the expert is brought to a testing facility, while other times the tests are conducted remotely and automated results are sent back for review. Automated expert tests are typically not as detailed as other types of usability tests, but their advantage is that they can be completed quickly. This is probably the easiest kind of testing to do in SWITCH, as the Use Cases are well understood by the SMEs that are providing them to the project.
- ***Questionnaires and Interviews:*** interviews enable the observer to ask direct questions to the users. Similarly, the observer can also ask questions by means of questionnaires. The advantage of questionnaires is that they allow more structured data collection.
- ***Do-it-Yourself Walkthrough:*** in this technique, the observer sets up a usability test situation by creating realistic scenarios. He or she then walks through the work themselves just like a user would.
- ***Automated Usability Evaluation:*** Various academic papers and prototypes have been developed in order to try and automate website usability testing, all with various degrees of success. One interesting approach is Justin Mifsud's USEFul Framework [20].

After performing usability tests, we will compile the information and take note of any issues that testers had in common. We will consider things such as the amount of time needed to perform a task in a scenario, number of errors that occurred, and the users' "happiness" and satisfaction [21].

6 Summary

6.1 Software functionality in public releases

The components of the SIDE subsystem are essentially fully implemented at the time of writing, but further integration testing and enhancement to be fully compatible with the SWITCH Use Cases will be necessary in the coming months. The following table summarises the software functionality available in the two public releases of SWITCH.

Table 5: SWITCH SIDE subsystem functionality in public releases.

Architecture components (defined in D2.3)	**Functionality in V1**	**Functionality in V2**	**Key Performance Indicators (KPI)**	**Current status**
SWITCH Workbench	Yes	Yes	User activities permitted by SWITCH components that can be conducted via SIDE	Mostly integrated with DRIP/ASAP; some integration currently being completed.
Front-end	Yes	Yes	Responsiveness to user input. Properties/constraints supported by DRIP or ASAP components expressible in IDE	Implemented and performance appears adequate; full performance and user testing to be completed. Properties/constraints can be expressed, but in some cases currently only via free-text metadata
Back-end	Yes	Yes		Integrated with front-end; TOSCA-based REST APIs supported for interactions with other SWITCH subsystems
Internal database	Yes	Yes		Provides the required support to back-end

Architecture components (defined in D2.3)	Functionality in V1	Functionality in V2	Key Performance Indicators (KPI)	Current status
SIDE Collaborative Management System	Yes	Yes		User management & project management implemented; Partial version control - changes are stored, but no roll back functionality.
Formal Reasoner/ Verifier	Yes	Yes	Types of QoS/QoE constraint that can be validated; Capture of un-satisfiable constraints prior to submission of application specification to DRIP	Partial – some verification is done, identifying infeasible application deployments, but rudimentary. Experiments in assisting with choice of NFRs (Pareto front-based; Qualitative Metadata Markers) have led to methods to be incorporated in order to complement the formal reasoning/verification of QoS/QoE constraints

6.2 Innovation

We have discussed a number of SIDE innovations in the present deliverable. The following table summarises our key innovations in relation to the current state of the art.

Table 6: SWITCH SIDE subsystem innovations.

Component	Current state of the Art	Innovation
Application composition	Many suites provide integrated facilities for Cloud planning and provisioning.	SIDE (as part of the SWITCH platform) is the only application featuring the ability to combine the infrastructure requirements and functionality in the system, realising the SWITCH co-programming metaphor.
Dynamic Smart Templates	There are ways to create UI for applications and to communicate with them with REST API	DSTs enable the developer rapidly develop GUI for testing or control of his or her applications. Can be used to interface many different components in a rapid manner. The result can be stored inside TOSCA.
Extending TOSCA with NFR	There is a lot of research done on NFR for applications.	Developments in SIDE as part of the SWITCH platform comprise the first attempt to codify the logic for management of NFR inside TOSCA.
Informing the user what influences the performance of the application.	A lot of work is being done on application modelling. Specifically, for the purposes of automatic control.	Our two new approaches described in Section 4 give the user information what influences the performance of a component or an application, and also the interactions between the parameters, so that (s)he knows what parameters of the virtual infrastructure to concentrate on.
Application verification	Verification of applications is an important field in computer language design, but is a developing area in relation to micro service composition tools.	We provide a system for “type checking” and general deployability of the connected application components.

The SIDE subsystem is used in the project together with the other two SWITCH subsystems to implement the industrial pilot Use Cases. In the last phase of the project, exploiting SIDE within the integrated SWITCH environment will be highlighted. A detailed exploitation plan and report will be presented in D6.4 “Report on dissemination, communication, collaboration, exploitation and standardization V3”.

Bibliography

[1] F. Quevedo, D. Rogers, P. Martin, A. Taal, Y. Hu, A. Jones and I. Taylor, D2.4 Concept description for application-infrastructure co-programming, SWITCH consortium, 2016.

[2] OASIS TOSCA Specification v1.0, "TOSCA 1.0 (Topology and Orchestration Specification for Cloud Applications), Version 1.0," 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf.

[3] G. Suciu Jr. and V. Suciu, D5.2 Technical and functional specifications for the elastic early warning system, SWITCH consortium, 2017.

[4] A. Ulisses, P. Ferreira, P. Santo, D. Costa and R. Amor, D5.3 Technical and functional specifications for the Cloud virtual studio for directing and broadcasting live events, SWITCH consortium, 2017.

[5] mendix developer, "Best practices for Component-Based Development," 2 July 2017. [Online]. Available: https://docs.mendix.com/howtogeneral/bestpractices/best-practices-for-component-based-development. [Accessed 20 July 2017].

[6] P. Martin, A. Jones, C. Rodrigo, R. Marcos and Z. Zhao, D3.3 Semantic linking model for SWITCH V2, SWITCH consortium, 2016.

[7] P. Martin, C. de Laat, J. Wang, H. Zhou, Y. Hu, A. Taal, S. Koulouzis and Z. Zhao, D3.4 Drip Technical Description, SWITCH consortium, 2017.

[8] S. Taherizadeh, J. Trnkoczy, U. Paščinski, M. Breška and V. Stankovski, D4.1 Prototype runtime monitoring system, SWITCH consortium, 2015.

[9] M. Cigale, J. Trnkoczy, S. Taherizadeh, S. Gec and V. Stankovski, D4.2 Design specification of the ASAP Subsystem, SWITCH Consortium, 2016.

[10] M. Cigale, J. Trnkoczy, P. Kochovski, T. Salman, S. Gec, U. Paščinski, P. Štefanič, V. Poenaru and V. Stankovski, D4.3 Technical Documentation of the ASAP Subsystem, SWITCH project, 2017.

[11] P. Martin and Z. Zhao, SWITCH Integration delivrable, SWITCH Consortium, 2017.

[12] P. Štefanič, D. Kimovski, G. Suciu Jr. and V. Stankovski, "Non-Functional Requirements Optimisation for Multi-Tier Cloud Applications: An Early Warning System Case Study," in *The 3rd IEEE Conference on Cloud and Big Data Computing* , San Francisco, California, 2017.

[13] P. Štefanič, M. Cigale, A. Jones and V. Stankovski, "Quality of Service models for Micro-services and their integration into the SWITCH IDE," in *1st IEEE Workshop on Autonomic Management of large scale container-based systems*, Tuchson, Arizona, 2017.

[14] P. Hirmer, U. Breitenbücher, T. Binz and F. Leymann, "Automatic Topology Completion of TOSCA-based Cloud Applications," in *GI Jahrestagung*, Stuttgart, 2014.

[15] T. Waizenegger, M. Wieland, T. Binz, U. Breitenb{\"u}cher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak and S. Wagner, "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences: Confederated International Conferences: CoopIS, DOA-Trusted Cloud, and ODBASE*, Graz, Austria, 2013.

[16] Juju, “Juju Docs,” 1 December 2016. [Online]. Available: https://jujucharms.com/docs/1.25/about-juju. [Accessed 19 July 2017].

[17] K. Baxley, J. De la Rosa and M. Wenning, “Deploying workloads with Juju and MAAS in Ubuntu 14.04 LTS,” 5 May 2014. [Online]. Available: http://docplayer.net/12356952-Solution-brief-ca-service-management-service-catalog-can-we-manage-and-deliver-the-services-needed-where-when-and-how-our-users-need-them.html. [Accessed 18 July 2017].

[18] Fabric8, “Fabric8,” 2 December 2016. [Online]. Available: http://fabric8.io/guide/overview.html. [Accessed 19 July 2017].

[19] B. E. John and D. E. Kieras, “The GOMS family of user interface analysis techniques: comparison and contrast,” *ACM Transactions on Computer-Human Interaction (TOCHI),* vol. 3, no. 4, pp. 320-351, 1996.

[20] J. Mifsud, “USEFul - A Framework To Automate Website Usability Evaluation (Part 1),” 30 January 2012. [Online]. Available: http://usabilitygeek.com/useful-a-framework-to-automate-website-usability-evaluation-part-1/. [Accessed 18 July 2017].

[21] T. Churm, “An Introduction To Website Usability Testing,” 9 July 2012. [Online]. Available: http://usabilitygeek.com/an-introduction-to-website-usability-testing/. [Accessed 20 July 2017].

[22] OpenTOSCA, “TOSCA and OpenTOSCA: TOSCA Introduction and OpenTOSCA Ecosystem Overview,” 8 November 2013. [Online]. Available: http://www.slideshare.net/OpenTOSCA/tosca-and-opentosca-tosca-introduction-and-opentosca-ecosystem-overview.

[23] K. Evans, A. Jones, F. Quevedo, D. Rogers, I. Taylor, J. Wang, Y. Hu, H. Zhou, A. Taal, P. Martin, S. Taherizadeh and J. Trnkoczy, D2.3 SWITCH Architecture Design, SWITCH Consortium, 2016.

Abbreviations

Abbreviation	Expansion
API	Application Programming Interface
ASAP	Autonomous Self-Adaptation Platform
BEPL	Business Process Execution Language
BPMN	Business Process Management Notation
CSAR	Cloud Service Archive
DRIP	Distributed Real-time Infrastructure Planner
GUI	Graphical User Interface

Abbreviation	Expansion
LQN	Layer Queueing Network
MVC	Model-View Controller
NFV	Network Function Virtualization
OASIS	Organization for the Advancement of Structured Information Standards
QoE	Quality of Experience
QoS	Quality of Service
REST	Representational state transfer
SIDE	Switch Interactive Development Environment
SDN	Software Defined Networking
SLA	Service Level Agreement
SWITCH	Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications
TOSCA	Topology and Orchestration Specification for Cloud Applications
TSDB	Time Series Database
YAML	YAML Ain't Markup Language
DST	Dynamic Smart Templates