Deliverable D4.4

Initial multi-cloud application deployment and adaptation

Editor(s):	Gema Maestro (Experis), Javier Gavilanes (Experis)
Responsible Partner:	Experis
Status-Version:	Final – v1.0
Date:	28/11/2017
Distribution level (CO, PU):	CO

Project Number:	GA 726755
Project Title:	DECIDE

Title of Deliverable:	Initial multi-cloud application deployment and adaptation
Due Date of Delivery to the EC:	30/11/2017

Workpackage responsible for the Deliverable:	WP4 - Continuous deployment and operation
Editor(s):	Gema Maestro (Experis), Javier Gavilanes (Experis)
Contributor(s):	Paolo Barone (HPE), Lorenzo Blasi (HPE
Reviewer(s):	Nicola Fantini (CB)
Approved by:	All Partners
Recommended/mandatory readers:	WP3, WP5

Abstract:	This initial version of the deliverable analyses the methods and the tools for implementing the automatic deployment and adaptation of multi-cloud applications and provides the technical description of the workflow manager implemented for supporting those activities.
Keyword List:	Cloud-native, microservice, REST, stateless, container, virtual machine, orchestration system, architecture, installation, packaging, usage.
Licensing information:	The document itself is delivered as a description for the European Commission about the released software, so it is not public.
Disclaimer	This deliverable reflects only the author's views and views and the Commission is not responsible for any use that may be made of the information contained therein

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	19/10/2017	First draft version	HPE
v0.2	14/11/2017	Sections completed and minor corrections	Experis
V1.0	29/11/2017	Ready for submission	TECNALIA

Table of Contents

List of Figures

List of Tables

Terms and abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CSP	Cloud Service Provider
DevOps	Development and Operations
DoW	Description of Work
EC	European Commission
GB	Giga Bytes
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KR	Key Result
MCSLA	Multi-Cloud Service Level Agreement
NCA	Native Cloud Application
NFR	Non-Functional Requirement
OS	Operating System
Protobuf	Protocol Buffers (Google's data interchange format)
QA	Quality Assurance
RAM	Random-Access Memory
REST	Representational State Transfer
SLA	Service Level Agreement
SLO	Service Level Objective
SQO	Service Quality Objective
SSH	Secure Shell
ToC	Table of Contents
UI	User Interface
URI	Unified Resource Identifier
URL	Unified Resource Locator
UC	Use Case
UUID	Universally Unique IDentifier
VM	Virtual Machine
WP	Work Package

Executive Summary

This deliverable describes the first implementation of ADAPT, DECIDE's continuous integration tool. The ADAPT prototype that will be developed during this first year is focused on the multi-cloud application deployment and adaptation functionalities, which will be carried out by the Deployment Orchestrator.

The Deployment Orchestrator is a component within ADAPT that is in charge of orchestrating the deployment lifecycle (deployment, un-deployment, user confirmation, redeployment) for the represented user application and its components. It takes as input a file that contains a description of the application (called *Application Description*) and communicates with ADAPT's Monitoring Manager to start and stop monitoring and with the GUI, for interaction with the user. It also interacts with ACSmI to start and release cloud resources when needed.

The actions that the Deployment Orchestrator sends to the different components are triggered by the Application Description. These actions belong to four categories: preparation, initialization, planning and execution. Since the services require infrastructure to run on, and ADAPT is in charge of provisioning and setting it up, these actions have to be applied twice: first for the infrastructure and then for the services.

The Deployment Orchestrator has been implemented as a microservice, packaged into a container image and running as a container, exposing a REST API to communicate with external components. It has been implemented by combining, customizing and even extending the following set of existing open source tools/apis: Flask [1], Werkzeug [2], Jinja 2 [3], Flask-RESTplus [4] and Terraform [5].

Since it is packaged as a Docker image, there are no specific steps to install the Deployment Orchestrator. The only requirements are to have access to a Linux-based system running a Docker daemon. The image can be found on an unofficial private repository, set up for DECIDE.

Regarding the usage of the component, once started, the Deployment Orchestrator provides endpoints for interacting with other DECIDE components. For evaluation purposes, it is possible to feed said endpoints manually. To successfully test the prototype, a user must have credentials for the Git repository where the Application Description is hosted and for the ACSmI in which resources will be provisioned.

1 Introduction

1.1 About this deliverable

The architecture of DECIDE ADAPT has been described in the deliverable D4.1 along with the relevant requirements. This deliverable focuses on the multi-cloud application deployment and adaptation functionalities of ADAPT. The document reports about what has been implemented in the first year of the project, how it fits into the overall DECIDE framework, and how it can be installed and used.

1.2 Document structure

The first section of the document (sect. 2) describes the expected deployment and adaptation functionalities of ADAPT, implemented by the Deployment Orchestrator component, and explains what has been provided in the first version, both from a functional and a technical standpoint.

This D4.4 is a software deliverable, therefore this document also provides details (sect. 3) about the released software: how it's structured, how it can be installed and used.

In the conclusions (sect. 4) indications are provided about the expected next steps for the implementation.

2 Implementation

2.1 Functional description

The main components of ADAPT are shown in Figure 1. This deliverable reports about the first implementation of the Deployment Orchestrator component.

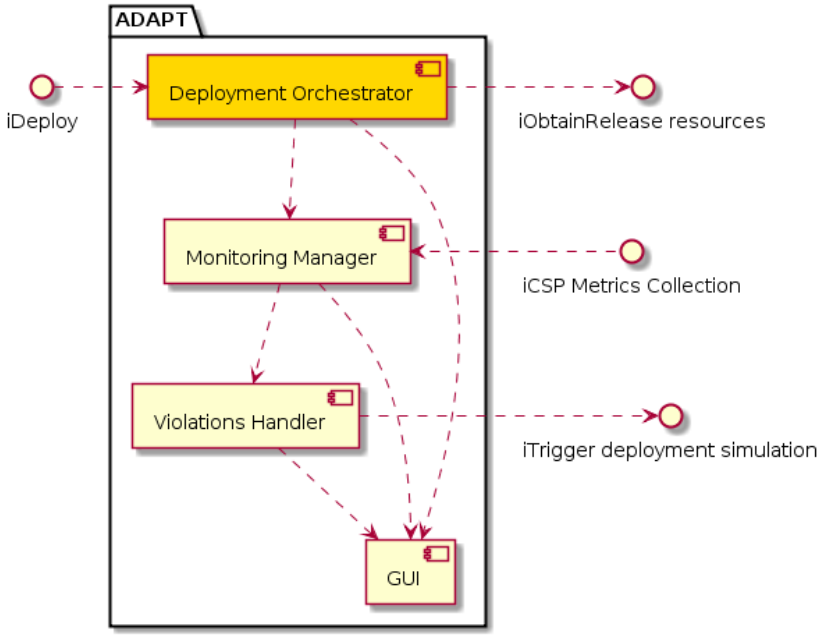

Figure 1 - DECIDE ADAPT components and interfaces

The Deployment Orchestrator, highlighted in the figure, takes as input an Application Description and is in charge of orchestrating the deployment lifecycle (deployment, un-deployment, user confirmation, redeployment) for the represented user application and its components.

The Application Description is a document including named fields describing the application to be deployed, its components, the cloud resources to be used and the applicable Multi-Cloud Service Level Agreement (MCSLA). The cloud resource provisioning is mediated by ACSmI, a cloud broker component which is the gateway to the Cloud Service Providers (CSPs) where the application components will be run. Therefore, the Application Description document includes also the information required to interact with the ACSmI API. A detailed definition of the Application Description is provided in deliverable D2.4 [6] and also in D4.1 [7] (sect. 5.1 and Appendix) for the specific sections relevant to the ADAPT component.

The main functionalities planned so far for the Deployment Orchestrator are the following.

F1. Read the input Application Description to extract all the needed information.
F2. If the application is already running and its level of technology risk is high the operator must explicitly confirm the redeployment through the GUI.
F3. Generate the deployment plan and scripts in the format understood by the selected underlying implementation technology.
F4. Obtain and start (through ACSmI Resources Management interface) the cloud infrastructure resources, typically Virtual Machines, from one or more providers as indicated in the Application Description.
F5. Configure the infrastructure resources, for example installing Docker on each VM.
F6. Verify the generated deployment plan.

F7. Execute the deployment plan to deploy and start all the application microservices indicated in the Application Description (each hosted within its own container and deployed potentially on a different provider).
F8. Trigger the monitoring functionality for the application and each of its components.
F9. In case of a redeployment un-deploy the previous instance of the application and release the unused infrastructure resources.
F10. Record the current deployment state.

As already indicated in D4.1 the deployment plan and scripts generation functionality (F3) has been moved from the Application Controller/OPTIMUS to ADAPT, since the plan and scripts are closely dependent on the technology selected for the ADAPT implementation (Terraform [5]).

The listed functionalities are implemented using an incremental approach over multiple releases. In this first release the following functionalities are implemented, using an example application (Socks Shop, see section 2.1.2) as a target: **F1**, **F3**, **F4**, **F5**, **F6**, **F7**, **F10**.

The following **¡Error! No se encuentra el origen de la referencia.** details the relationship between the deployment requirements indicated in deliverable D4.1 [7] and the implemented functionalities, with a description of the coverage for each functionality.

Table 1. Relationship between ADAPT functionalities and ADAPT requirements

Functionality	Req. ID	Coverage
F1	WP4-REQ34, WP4-REQ35, WP4-REQ36	An application description for the target application (see section 2.1.2) has been created by hand; the prototype reads it and extracts all the needed information.
F2	WP4-REQ14, WP4-REQ23, DEVOPS-REQF16	None.
F3	WP4-REQ3, WP4-REQ12	The prototype generates a Terraform deployment plan for the application based on the information read from the Application Description; the script needed to install Docker on each Virtual Machine is also generated from a template.
F4	WP4-REQ9, WP4-REQ28, DEVOPS-REQF6	The Virtual Machines indicated in the Application Description are started through the CloudBroker / ACSmI API interface from two Cloud providers: CloudSigma and AWS.
F5	WP4-REQ9, WP4-REQ28, DEVOPS-REQF6	The prototype runs the Docker installation script on each started Virtual Machine resource.
F6	WP4-REQ9, WP4-REQ28, DEVOPS-REQF6	The prototype verifies the generated Terraform deployment plan using the terraform plan command.
F7	WP4-REQ9, WP4-REQ28, DEVOPS-REQF6	The prototype executes the generated deployment plan using the terraform apply command, which starts all the application containers on the provider indicated in the Application Description.

Functionality	Req. ID	Coverage
F8	WP4-REQ1, WP4-REQ9, WP4-REQ34, DEVOPS-REQF16	None.
F9	WP4-REQ1, WP4-REQ21, WP4-REQ30, WP4-REQ44, DEVOPS-REQF16	None.
F10	WP4-REQ1, WP4-REQ15, WP4-REQ21, WP4-REQ30, WP4-REQ44, DEVOPS-REQF16	The technology selected for the ADAPT implementation (Terraform) already stores the current deployment state, as a starting point for any further commands. The history of deployment configurations produced by OPTIMUS, is stored by OPTIMUS itself.

The innovation of the implemented prototype lies in the model-based automatic deployment functionality (the Application Description represents the model) and in the automatic generation of a deployment plan from the model itself.

2.1.1 Fitting into overall DECIDE Architecture

ADAPT is the tool that drives the DECIDE operations lifecycle. As shown in Figure 2 it takes as input a model of the multi-cloud application to be deployed, the Application Description, and performs its deployment and monitoring. In case of a violation of the Multi-Cloud application SLA (MCSLA), ADAPT starts the adaptation cycle involving OPTIMUS (to simulate a new deployment configuration) and ACSmI (to find and contract the right CSP providers for the new configuration). The adaptation cycle then restarts with ADAPT redeploying the application according to the new configuration.

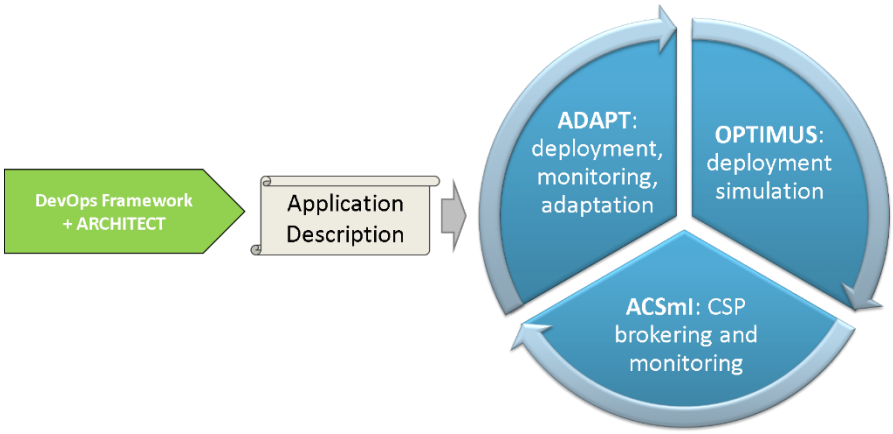

Figure 2. Application Description feeds the DECIDE operations lifecycle

The deployment and redeployment functionalities in ADAPT are performed by the Deployment Orchestrator component described in this deliverable.

As shown in Figure 3, the Deployment Orchestrator interacts with other two components of ADAPT: with the Monitoring Manager, to start and stop monitoring, and with the GUI, to obtain operator's confirmation in case of the redeployment of a high technology risk application.

Deployment Orchestrator also interacts with other tools in the DECIDE ecosystem: it is activated by the Application Controller and uses ACSmI's Resources Management interface to start and release cloud resources from any needed Cloud Service Provider.

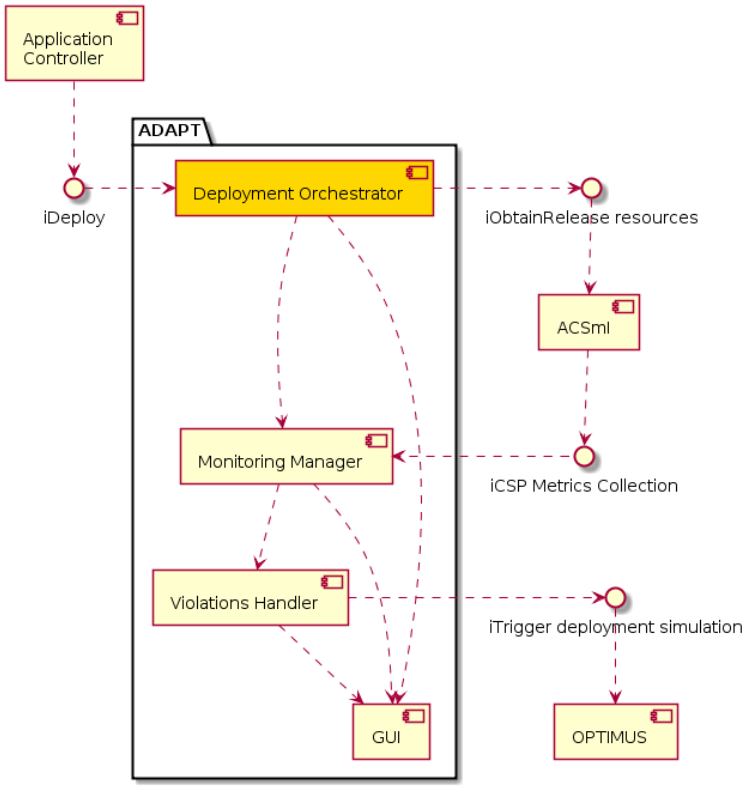

Figure 3. Deployment Orchestrator interactions and external interfaces

2.1.2 The target application: Socks Shop

The Socks Shop microservice application [8] available on the Internet from Weaveworks Inc., is often taken as a proof-of-concept reference in the microservices area. The application is provided by Weaveworks to show how a well-designed microservices application can be seamlessly deployed on different container runtime environments (Docker Compose, Docker Swarm, Kubernetes, Mesos, Amazon ECS, …) with very little effort, by just modifying deployment configuration files.

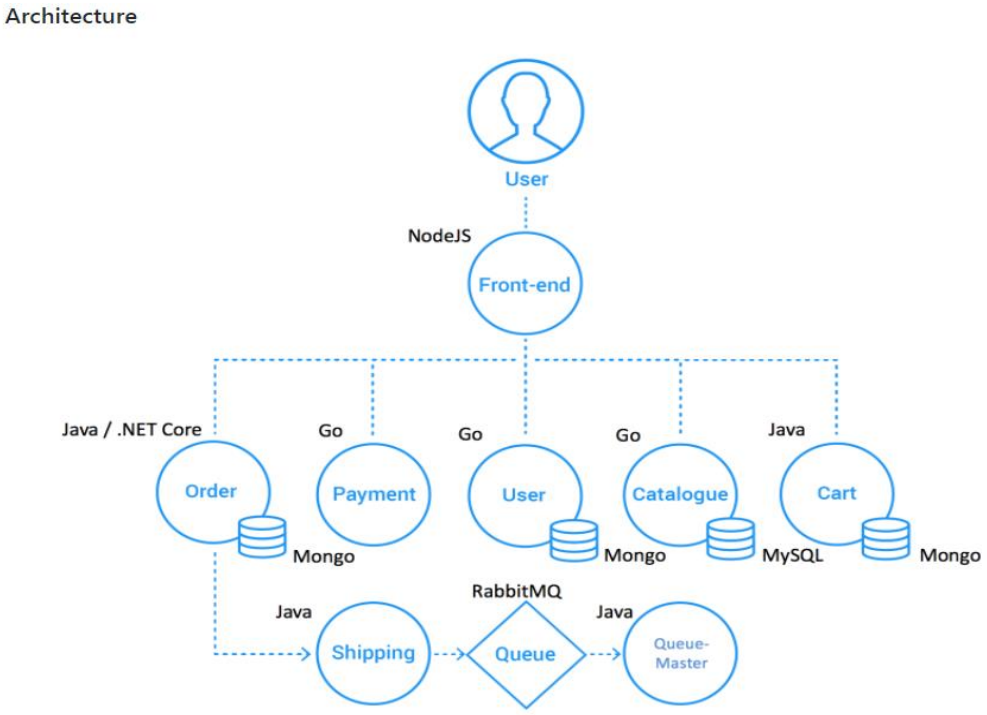

Figure 4. Socks Shop application architecture (source: *[9]*)

Figure 4 depicts the application architecture. For the purposes of this document, we don't need to gain in-depth knowledge of the application components, we rather need to know that every item consists of a software unit which is packaged into a Docker image and can be launched as a container by just specifying the image registry address, name and tag from any Linux machine or virtual machine running a Docker instance. The Socks Shop application is composed by 8 services provided by 13 containers (DBs and RabbitMQ are containerized as well).

Since in DECIDE we are specifically targeting multi-cloud deployment of Cloud Native Applications [10], we used such application as a reference to show how to deploy its microservices over *different* Docker nodes, running in *separate* virtual machines that are *automatically provisioned and configured* via Terraform.

The original deployment options provided by Weaveworks were intended for *cluster-based* technologies, which assume the Docker nodes running on the same cluster-level network; this means that they have to run on the same low-level infrastructure or, at least, in the same region of a specific Cloud Service Provider (CSP).

A sample deployment scenario based on this original approach, with a 3 nodes Docker Swarm cluster, is represented in Figure 5, where the microservices are spread over three different Linux virtual machines, each one running a Docker daemon, with service s1 being the Front-end and entry point for the application, acting as reverse proxy and orchestrating the application logic interactions towards the "backend" microservices. Since all the three virtual machines are running in the same CSP infrastructure, it is possible to use this deployment scenario based on a clustered technology.

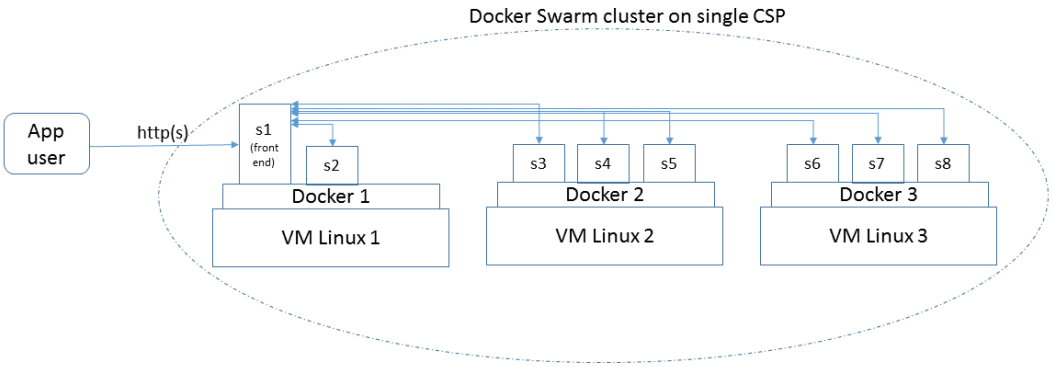

Figure 5. Sample deployment scenario based on a Docker Swarm cluster, running in a single CSP

In the DECIDE target scenario, we wanted to be able to split the services arbitrarily over different clouds, with the single microservice as the minimal deployment unit. Therefore, we had to change the application configurations:

- we changed the container launch parameters, so that every container can be run with no dependencies on a specific cluster technology or network (apart from the services connected to a DB/Queue, where the service container and the related DB/Queue container are run on the same Docker node and with a dedicated Docker interconnecting network for efficiency's sake);
- we modified the Front-end (which is based on Traefik [11], a latest generation reverse-proxy and load balancer) launch command so that the configuration is based on a K/V store (based on Consul [12]) in place of a static configuration file. This allows to change dynamically the reverse proxy addressing rules upon re-deployment of a service on a different docker node.

Figure 6 depicts one possible sample deployment for the target scenario, where the services are spread over 4 Docker nodes, each one running in a VM provided by a different CSP.

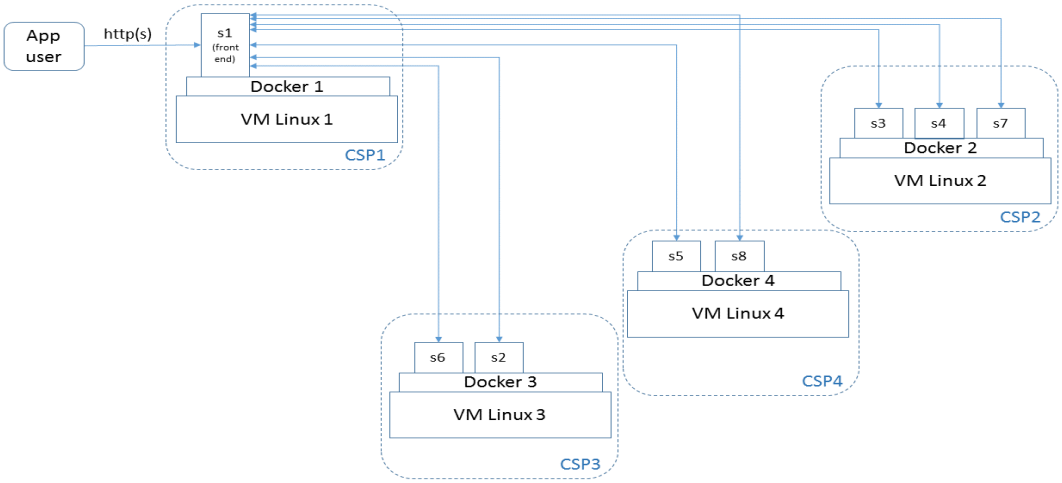

Figure 6. Sample deployment scenario with services spread over different cloud providers

To implement the deployment scenario depicted above, we must address the following tasks:

1. Provision the required set of Linux virtual machines. In this specific example, we are planning to leverage virtual machines started on a set of different Cloud Service Providers; but, in

principle, they can be as well physical servers hosted by a private infrastructure, or by a private cloud, or even a combination of them.

2. Set up the Linux machines with all the required tools and configurations. In particular, for each Linux machine, we have to:
    a. install Docker;
    b. create certificates and keys for securing the Docker socket (so that we can issue Docker commands remotely in a safe manner);
    c. install on every node a Consul agent, so that we can monitor the service status, and make each of them join a Consul master node in a WAN Consul cluster.
3. For every service, available as container image on the Docker Hub, we have to issue the Docker run command, with proper launch parameters, on the remote Docker instance planned by our deployment schema

Clearly, this means that we have to do a lot of manual operations and repeat them for every node. Moreover, if we want to modify the deployment plan, e.g. replacing or adding a virtual machine, we have to repeat all of them from scratch.

In the next sections, we show how we addressed this with the Terraform tool and how, after creating once for all the proper configuration items, this allowed us to provision all the desired elements from scratch with just a minimum set of commands. We choose to split the process into two separate operations: the *infrastructure provisioning* and the *services management*.

2.2 Technical description

The ADAPT Deployment Orchestrator has been implemented as a microservice, packaged into a container image and running as a container, exposing a REST API for remote invocation. The API endpoints allow external components to perform deployment requests and to get status information on the submitted operations. The ADAPT internals are based on custom code which extends, customizes and combines a set of open source tools, resulting in a framework that transforms high-level requests to the API into:

- technology-specific requests to the external ACSmI component for the provisioning of cloud resources;
- resource-specific operations on cloud resources for their configuration and set up, consisting of steps for preparing the execution environment for the application components;
- deployment and execution actions on the application components.

In the DECIDE framework, a container is the minimum deployment unit, and the applications are obtained by composition of services provided by a set of containers. Therefore, the application components that get deployed on cloud resources consist, as well as ADAPT, of microservices, packaged into container images and deployed as containers.

2.2.1 Prototype architecture

The ADAPT Deployment Orchestrator has been realized by combining, customizing and even extending the following set of existing open source tools/apis: Flask [1], Werkzeug [2], Jinja 2 [3], Flask-RESTplus [4], Terraform [13].

Flask is a framework written in Python [14], which combines Werkzeug, a utility library which implements the WSGI [15] specification, with Jinja 2, a templating engine for Python. This combination of tools builds up a framework for providing web services compliant to the WSGI specification and for dynamically generating files based on pre-configured templates. This mechanism is in particular suitable for creating configuration files at runtime.

Flask-RESTplus is an extension for Flask which allows to quickly implement http-based APIs based on the REST specification.

Terraform is an open source tool which allows to define infrastructures via configuration files, and to provision them via a set of commands. Terraform can interact with a huge set of cloud providers APIs and with configuration and management tools. It can also be extended with custom plugins to interact with custom providers and tools. One of the most powerful features of Terraform is that it manages automatically the infrastructure state and, whenever a change is made in the configuration, it can apply the updates to the part of the infrastructure involved, managing all the dependencies in the resources, and leaving unchanged the ones that are not affected. It also allows to save the infrastructure configuration and the related state into code versioning systems, hence fully implementing the IaC [16] paradigm: the infrastructure is really considered and managed as if it were a software code artifact.

In DECIDE, we extended Terraform by developing a specific plugin which allows the ADAPT Deployment Orchestrator to interact with the cloud broker component managed by ACSmI, so that we can use Terraform for instantiating cloud resources via the broker API. The custom plugin is considered as part of the "Multi-cloud application helpers" and its specification is documented in the dedicated Deliverable D4.10 [17].

Terraform provides a command line client, and the related operations are driven by the contents of *all the configuration files with ".tf"* extension *found in the directory from which the Terraform commands are launched*. Terraform configuration files can be written either in JSON format, or in a more readable and synthetic proprietary language called Hashicorp Configuration Language (HCL) [18], and the items defined there can be referenced by means of an interpolation language [13]. Terraform parses the configuration files and builds a dependency graph, which allows to automatically manage dependencies between the configured resources.

The main operations of the Terraform command line shell are:

- `terraform init`: parses the terraform configuration files and initialize an internal data structure corresponding to the initial infrastructure state.
- `terraform plan`: parses the configuration files and prepares a "plan" for building the infrastructure. In practice, it is a command which allows simulating the Terraform actions without applying them; it verifies if the configuration and the dependencies are consistent and allows you to check the operations before applying them.
- `terraform apply`: applies the plan, by issuing all the related commands, e.g. invoking a cloud provider API for starting a VM, etc.

Terraform is usually adopted to define statically the infrastructure configuration, and to exploit its powerful capabilities in terms of infrastructure management. You usually define in advance the infrastructure resources you want to get, then apply the configuration and Terraform takes care of the provisioning. Most noteworthy, if you change some part of your configuration afterwards, Terraform is able to apply that specific modification without impacting the other infrastructure resources.

The key point in the usage of Terraform within the ADAPT Deployment Orchestrator is that we pushed its usage some steps forward than usual: **in ADAPT we are creating the Terraform resources configuration files dynamically at runtime, depending on the model defined in the Application Descriptor**. This allowed us to apply a real multi-cloud application deployment at runtime, on different CSPs, without relying on specific clustering technologies which usually require at least to be run on a single cloud provider and in the same availability zone.

The diagram in Figure 7 depicts the ADAPT Deployment Orchestrator architecture, where the above-mentioned tools have been extended and integrated to form three logical components: *ADAPT Deployment Orchestrator REST API, ADAPT Deployment Orchestrator Preparation Engine, ADAPT Deployment Orchestrator Execution Engine.*

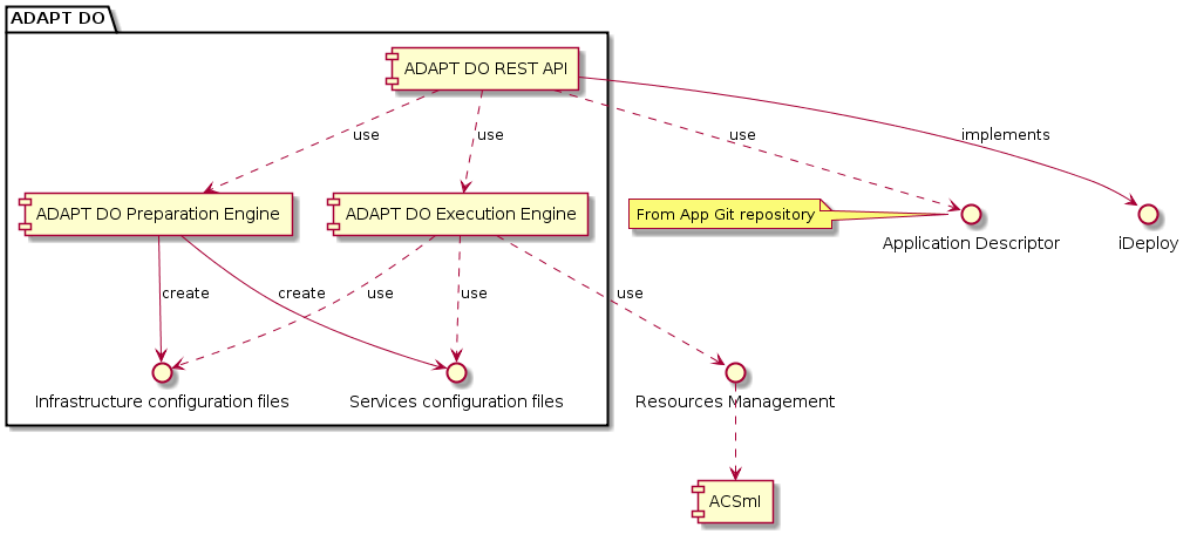

Figure 7. ADAPT Deployment Orchestrator architecture

The ADAPT Deployment Orchestrator REST API:

- implements a http(s) REST API, enabling WSGI endpoints and REST actions for invoking the offered functionalities;
- maps http(s) requests to ADAPT Engine Preparation and Deployment actions;
- maps actions result to http(s) responses.

The ADAPT Deployment Orchestrator Preparation Engine provides a set of custom templates for Terraform resources configuration files and, leveraging Flask and the Jinja 2 templating engine, creates dynamically upon request a set of Terraform configuration files, based on a description of resources contained in the Application Description document.

The ADAPT Deployment Orchestrator Execution Engine uses Terraform and the custom Terraform plugin to apply Terraform operations to the generated configuration files, resulting in the:

- Provisioning of cloud resources;
- Configuration and installation of pre-requisites on cloud resources;
- Deployment of application components on cloud resources ad their start-up.

2.2.1.1 ADAPT Deployment Orchestrator Interactions

This section describes the ADAPT Deployment Orchestrator interactions with the external components, and the flow of operations between its internal components.

ADAPT Deployment Orchestrator actions are triggered by the Application Controller and are related to the following categories of operations:

- *Preparation*: the creation of Terraform configuration files for both the *Infrastructure* (virtual machines) and *Services* (containers) environments.
- *Initialization*: the creation of an initial Terraform state file for each environment, which also verifies the availability of the required providers plugins.

- *Planning*: the creation of an execution plan on the configured resources, which implies a verification of the formal correctness and of the resources dependencies.
- *Execution*: the application of the execution plan to the configured resources prepared in the previous steps, for a target environment.

The infrastructure (consisting of a set of virtual machines) is a prerequisite for the running of services (the containers to be launched on the infrastructure). Therefore the above list of operations must be triggered for the infrastructure first; then, after the infrastructure is fully operational, the same steps can be applied to the deployment of the services.

2.2.1.1.1 Interactions for provisioning the infrastructure

Figure 8 describes the interactions for generating the configuration files for the provisioning of the infrastructure, which consists in a set of virtual machines on which the services will be deployed as containers. The preparation phase is triggered by the Application Controller via a request on the REST API (documented in section 2.2.2.1). The external component involved is the Git repository where the Application Description document resides. The operations between the internal ADAPT Deployment Orchestrator components are specified in detail in the dedicated section 2.2.1.2.

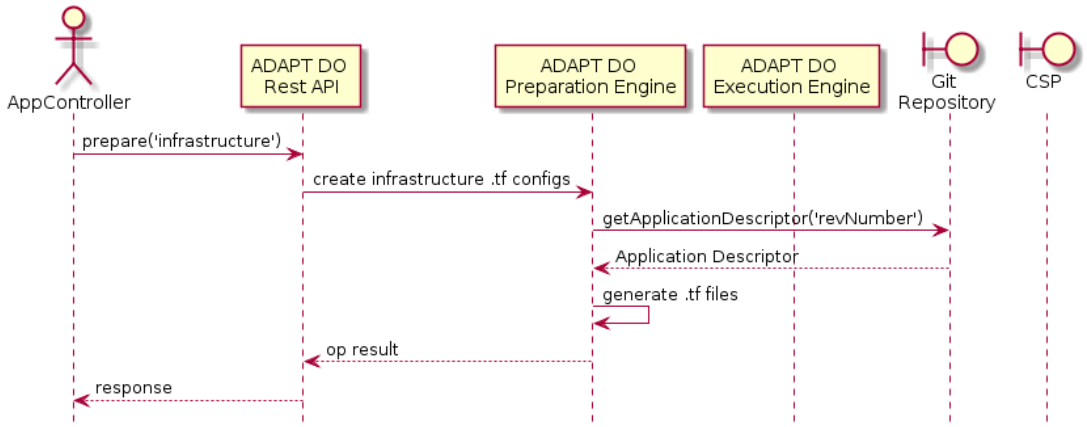

Figure 8. Sequence of steps to create configuration files for the infrastructure provisioning

Figure 9 shows the interactions for initializing the infrastructure. The Initialization phase is triggered by the Application Controller via a request on the REST API (documented in section 2.2.2.1) and applied to the configuration files generated in the previous step.

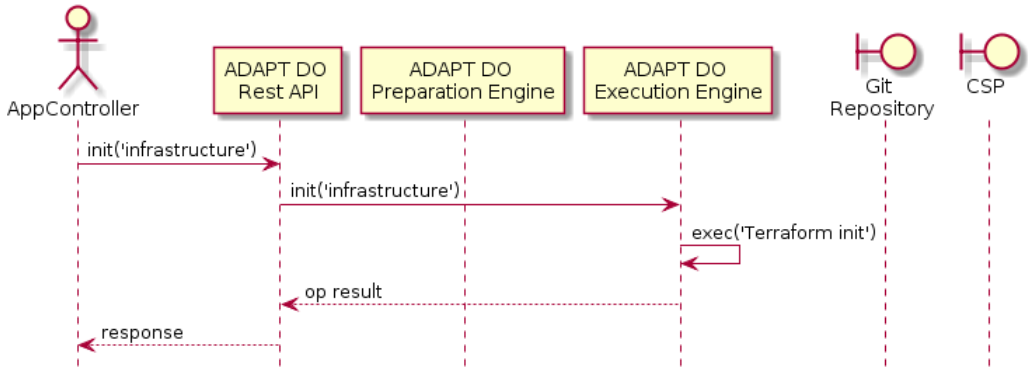

Figure 9. Initializing the Terraform state for the infrastructure

Once the initialization has taken place, the Application Controller can trigger the Planning action on the REST API, as shown in Figure 10.

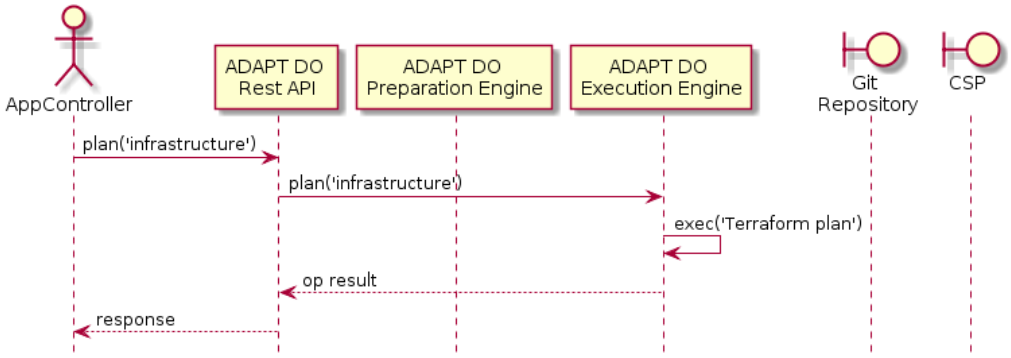

Figure 10. Generating an execution plan for the infrastructure

The operations between the internal ADAPT Deployment Orchestrator components for both the Initialization and Planning phases are specified in detail in the dedicated section 2.2.1.3.

Finally, the Execution step, the last one towards the real provisioning of the Infrastructure, is shown in Figure 11. The Application Controller triggers the request on the REST API (documented in section 2.2.2.1) and the Terraform 'apply' command is executed on the execution plan prepared in the previous step. This results in a set of actions towards the remote CSP, which are specified in detail in section 2.2.1.3.

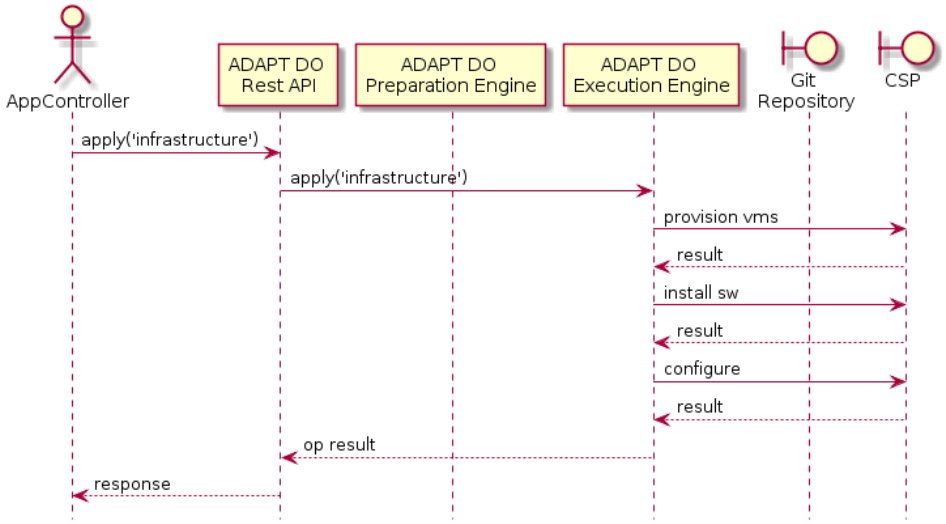

Figure 11. Applying the execution plan for the infrastructure

2.2.1.1.2 Interactions for provisioning the services

Figure 12 describes the interactions for generating the configuration files for the provisioning of the services, which consists in a set of containers deployed on the virtual machines provisioned in the previous steps. The preparation phase is triggered by the Application Controller via a request on the REST API (documented in section 2.2.2.1). The external component involved is the Git repository where the Application Description document resides. The operations between the internal ADAPT Deployment Orchestrator components are specified in detail in the dedicated section 2.2.1.2.

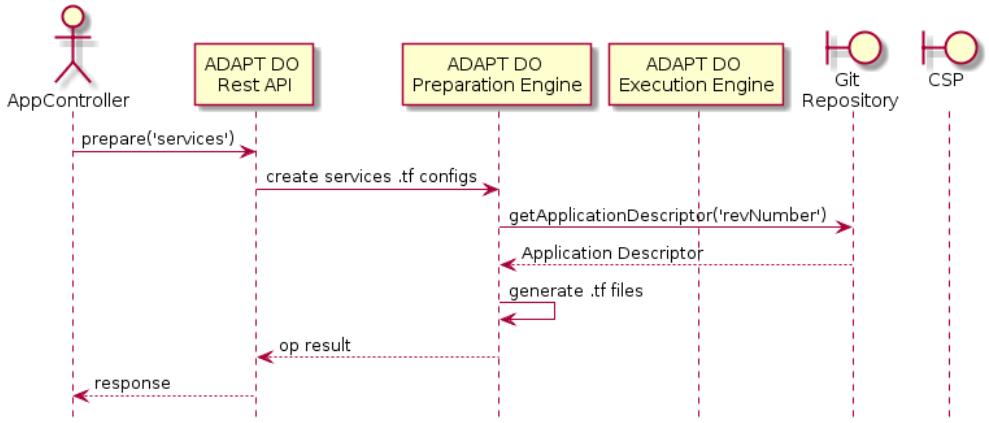

Figure 12. Sequence of steps for the creation of configuration files for the services provisioning

Figure 13 shows the interactions for initializing the services. The initialization phase is triggered by the Application Controller via a request on the REST API (documented in section 2.2.2.1) and applied to the configuration files generated in the previous step.

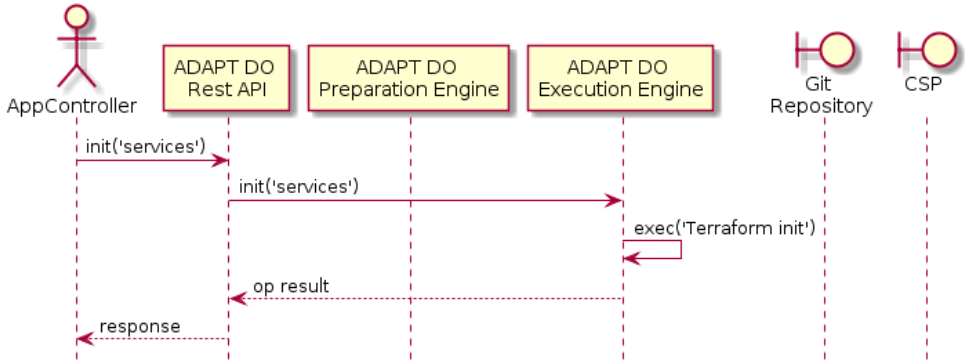

Figure 13. Initializing the Terraform state for the services

Once the initialization has taken place, the Application Controller can trigger the Planning action on the REST API, as shown in Figure 14.

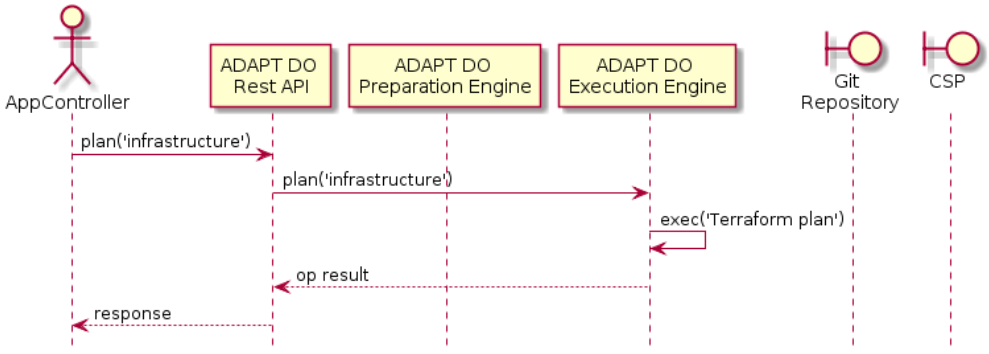

Figure 14. Generating an execution plan for the services

The operations between the internal ADAPT Deployment Orchestrator components for both the Initialization and Planning phases are specified in detail in the dedicated section 2.2.1.4.

Finally, the Execution step, the last one towards the real deployment and startup of the services, is shown in the next figure. The Application Controller triggers the request on the REST API (documented

in section 2.2.2.1) and the Terraform 'apply' command is executed on the execution plan prepared in the previous step. This results in a set of actions towards the remote CSP, which are specified in detail in section 2.2.1.4.

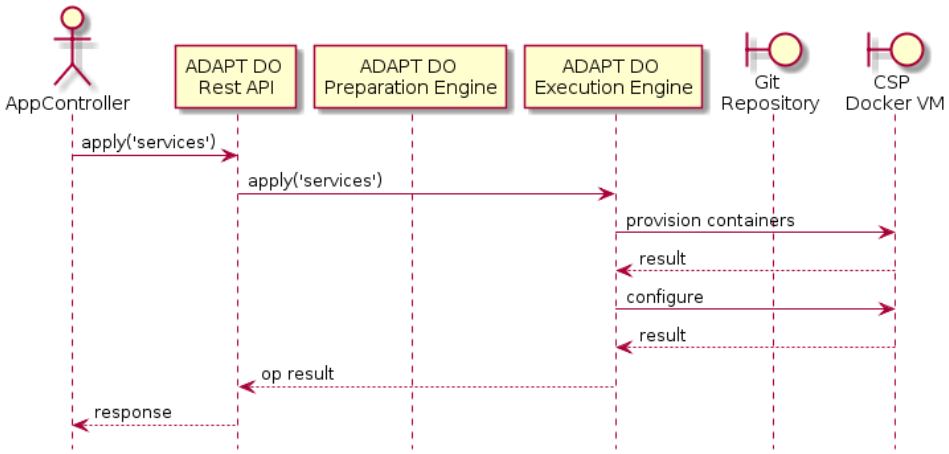

Figure 15. Applying the execution plan for the services

2.2.1.2 Dynamic creation of Terraform configuration files

Figure 16 depicts the process of creation of Terraform configuration files, that will be used for the provisioning of the cloud resources required by the application. A POST request is received on the REST API, specifying in the body the information required to retrieve the Application Descriptor file form a repository. Once retrieved, the Descriptor is passed to the ADAPT Deployment Orchestrator Preparation Engine where it is parsed and, for each cloud resource configuration section, one or more Terraform configuration files are generated. The outcome is a set of Terraform configuration files, placed in a dedicated directory and representing the "infrastructure" for the application runtime environment.

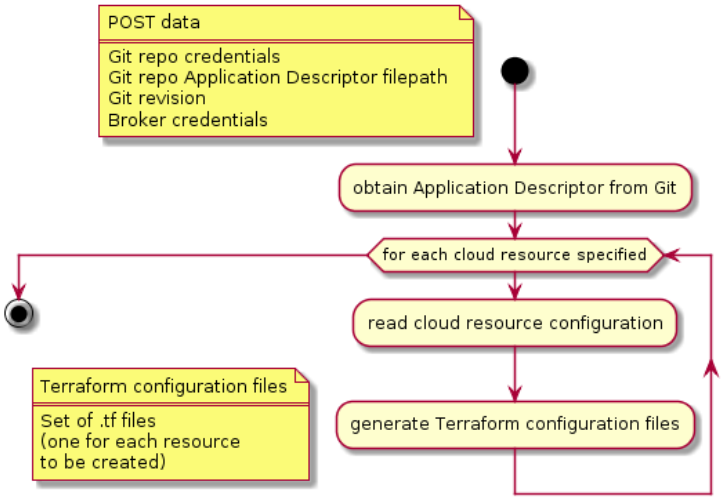

Figure 16. Preparation of configuration files for provisioning of cloud resources required by the application

In a similar manner, Figure 17 depicts the process of creation of a set of Terraform configuration files that will be used for the provisioning of the microservices composing the application, packaged as containers. Again, a POST request is received on the REST API, specifying in the body the information required to retrieve the Application Descriptor file. The Descriptor is passed to the ADAPT Deployment Orchestrator Preparation Engine where it is parsed and for each container section, one or more

Terraform configuration files are generated. The outcome is a set of Terraform configuration files, created in a dedicated directory and representing the microservices which implement the functionalities of the application.

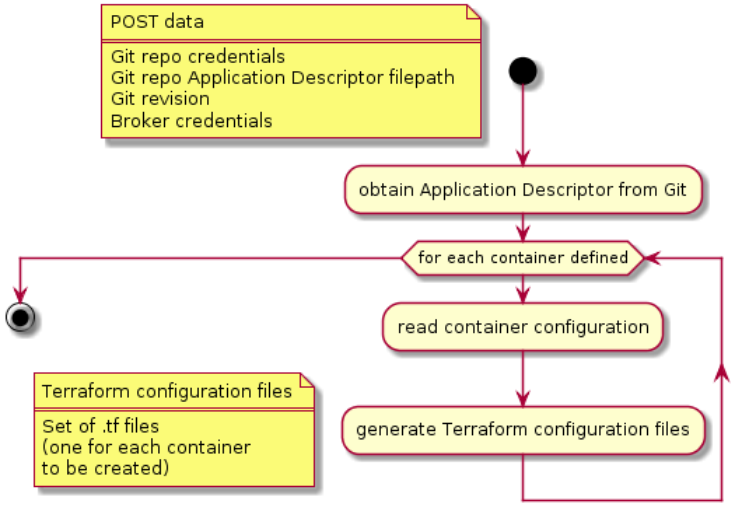

Figure 17. Preparation of configuration files for provisioning of microservices composing the application

2.2.1.3 Terraform operations for managing the "infrastructure" creation

Figure 18 shows the initialization phase for the Terraform execution environment dedicated to the provisioning of cloud resources. Upon the reception of a POST request for initialization of the "infrastructure" environment of a given application, for which a set of configuration files have been already generated, the API component routes it to the ADAPT Deployment Orchestrator Execution Engine, which applies the Terraform 'init' command on the set of files. The effect of the 'init' operation is the verification of local environment pre-requisites for the execution of future provisioning requests, namely the set of Terraform plugins needed to connect to the required cloud providers. In the case of DECIDE, it verifies the availability of the custom Terraform plugin for connecting to the broker API via ACSmI. If verification is successful, an initial Terraform 'state' data structure is created and used as reference for future Terraform operations.

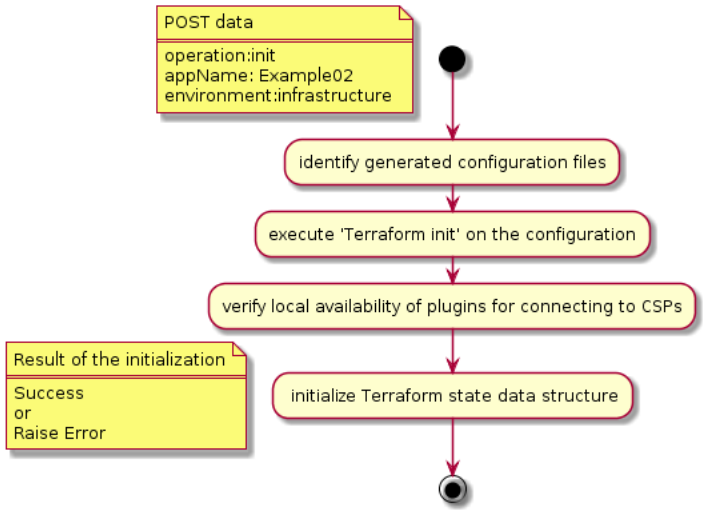

Figure 18. Initialization of the Terraform environment for the provisioning of cloud resources

Before creating real resources or applying modifications to existing infrastructures, it is good practice to verify whether a prepared set of Terraform configuration files are going to provide the expected results. For this, Terraform provides a 'plan' command which simulates the application of creation commands based on the set of configuration files provided and displays a report. While doing that, it verifies the formal correctness of the configurations, builds a resource dependency graph and performs a dependency check between all the resources involved. This process is depicted in Figure 19.

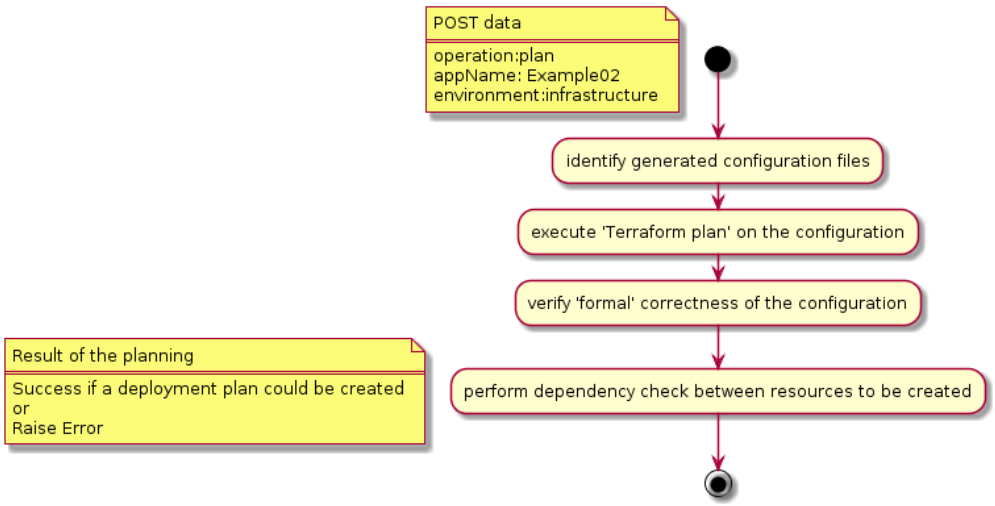

Figure 19. Verification step for the Terraform configuration files generated

Once the 'plan' operation is successful, it is possible to invoke on the ADAPT Deployment Orchestrator REST API the 'apply' operation, which allows the real provisioning of the cloud resources. As shown in ***Figure 20***, the REST API routes the 'apply' request to the ADAPT Deployment Orchestrator Execution Engine, which executes the Terraform 'apply' operation on the set of configuration files related to the provisioning of the application infrastructure. For each resource to be created, the corresponding plugin to connect to the involved CSP is invoked. The resource provisioning is a long-lasting operation whose duration depends on several factors such as the type of CSP, the type of resource, the network latency etc. As an example, if the resource to be started is a virtual machine the operation will be pending until the operating system has booted and a public IP address has been assigned, which may take minutes. Therefore, the resource creation operations submitted with the Terraform 'apply' command are delegated to background processes, while the 'apply' operation returns almost immediately. Before returning, the operation creates two log files: the first one represents the current status of the resource creation operations and contains a value which can be "running" if the creation is still ongoing, "0" if finished with success or a code greater than zero if an error occurred; the second one is the operation log itself created by Terraform, containing the operations taking place during the creation process.

After the creation of the log files, the operation returns providing in the response:

- the result of the "request submission" operation (code for success or failure);
- an endpoint that can be used to get the current status of the operation (taken from the operation status log file);
- an endpoint that can be used to get the current log of the operation.

A client (e.g. the Application Controller) can then use these endpoints to poll the ADAPT Deployment Orchestrator to detect if the operation has finished and if it was successful.

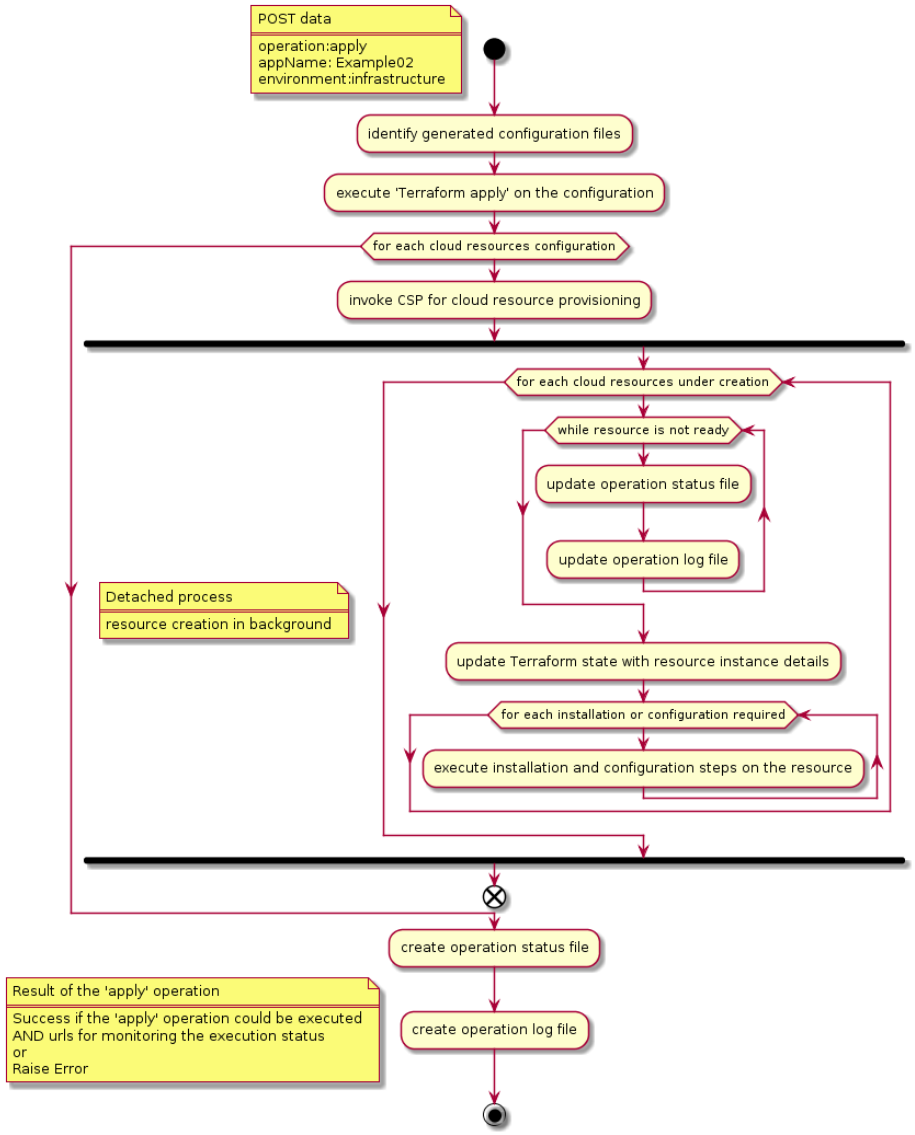

Figure 20. Execution of Terraform 'apply' action on the configuration files generated, to provision the cloud resources

In the meanwhile, the process of resource creation proceeds in the background according to the central section of Figure 20. Terraform manages the creation of the cloud resources and updates the creation steps accordingly in the operation log file. Once the creation has terminated, the Terraform state file is updated to register the new "snapshot" of the infrastructure. After the resource creation, a set of configurations and set up steps, if specified in the resource configuration files, are applied by Terraform. Typically, for the DECIDE case, they consist of the installation of software tools required to run the DECIDE applications such as Docker and Consul, and in the creation or configuration of ssh keys, certificates (e.g. to secure the Docker daemon socket, so that it can be reliably invoked remotely when deploying the containers representing the application microservices) and configuration files.

2.2.1.4 Terraform operations for managing the "microservices" creation

In the same way as for the creation of the infrastructure resources needed as runtime environment for the DECIDE applications, the software resources to be deployed and started on them are managed via the Terraform phases of initialization, planning and execution. Regarding the first two phases, they are identical to the ones described for the infrastructures and the corresponding action diagrams match the ones in Figure 18 and Figure 19.

Regarding the provisioning phase, as we deal with microservices packaged into containers, the steps are somehow different than the ones related to the creation of cloud resources because they imply the start of a container with proper start-up parameters, performed by issuing remote Docker commands to the target cloud resource. This is represented in Figure 21.

When the Terraform 'apply' command is issued on the configuration files related to the application services, for each container to create a first check is related to the origin of the related images. Images can be either public, meaning that they are hosted on the Docker Hub, or private, meaning that they are hosted on a private repository. In the latter case, the first step for getting the image is to perform authentication from the Docker host to the private registry with proper credentials. After that, the image can be pulled from the repository and the container started with the proper start-up parameters. This operation is repeated for all the containers to be started and, in the same way as for the creation of the infrastructure resources, it creates initial status and log files and returns call-back URLs for polling, and the container provisioning steps continue on the background. Once a container is started, it is registered as a service in the ADAPT consul registry for basic health and availability checks. ADAPT is able to provide, optionally and if specifically declared in the Application Descriptor, a reverse-proxy based on Traefik to be used as a single entry point to the application endpoints. If the Application Descriptor declares that the application wants to rely on the ADAPT reverse-proxy mechanism, the application endpoints provided by every container, together with the routing rules, are registered as key/value configuration items in the ADAPT Consul K/V store and used as remote configuration for an instance of the Traefik reverse-proxy and load balancer dedicated to the application, which will then be used as single entry point to the application services.

For this to happen, proper configuration rules must be specified in the Application Descriptor and the application logic must be implemented taking into account the forwarding rules defined in the configuration.

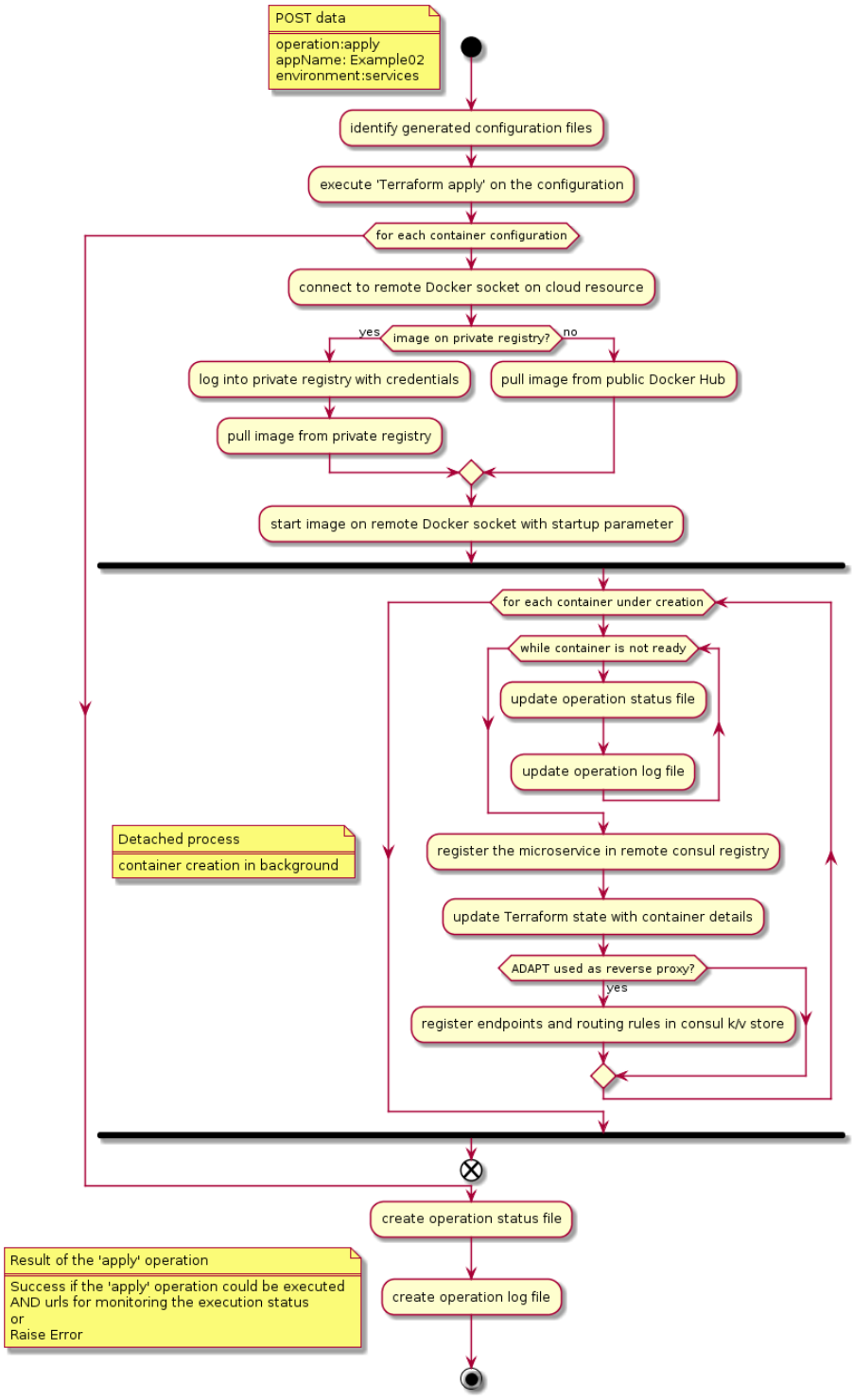

Figure 21. Execution of Terraform 'apply' action on the configuration files generated, to provision the containers

The remainder of this chapter describes in detail the three components of the ADAPT Deployment Orchestrator, and the actions involved in their operation.

2.2.2 Components description

2.2.2.1 ADAPT Deployment Orchestrator REST API

As introduced in section 2.2.1, the ADAPT Deployment Orchestrator REST API component is the interface of ADAPT Deployment Orchestrator to the other DECIDE components. The purpose of this interface is to provide http(s) endpoints that allow external actors to trigger actions related to the deployment of application components to cloud resources, based on an Application Description document.

The key-role of the ADAPT Deployment Orchestrator REST API is to expose the component endpoints and to manage the requests, by mapping properly the input parameters and the submitted contents, routing them to the proper lower level ADAPT Deployment Orchestrator Engine.

The requests can be based on POST or GET methods. POST requests trigger one of the following operations.

On the ADAPT Deployment Orchestrator Preparation Engine:

- dynamic generation of a set of configuration files representing the cloud resources to be provisioned for running the application;
- dynamic generation of configuration files representing the software components (microservices packaged into containers) to be deployed on the cloud resources.

On the ADAPT Deployment Orchestrator Execution Engine:

- Execution of Terraform commands to require to ACSmI the provisioning of the cloud resources for which configuration items were created, and to configure on them all the software tools, settings and credentials required by the application components.
- Execution of Terraform commands to deploy the application microservices (containers) into the cloud resources according to the deployment schema decided by the DECIDE upper layer components and described in the Application Description.

GET requests allow to retrieve on the ADAPT Deployment Orchestrator Execution Engine:

- The status of an operation submitted via a POST request. Some operations are time consuming, as they involve the provisioning of sets of cloud resources which require time to start up, to get public addresses, etc. Therefore, the POST operations reply with an operation identifier and a call-back URL, that can be used by the clients to get the status of the operation; this allow a client to implement a polling mechanism to get the status of the operation until it returns a result corresponding to a final or error state.
- The log of an operation submitted via a POST request. In the same way as for the operation status, this allows a client to get the current execution log of the POSTed operation.

In the following, we list the endpoints developed so far with their specification and description

The first endpoint, shown in Figure 22, allows to prepare the configuration files required by a specific deployment action. DECIDE applications are based on compositions of microservices provided by containers running on cloud resources. Therefore, this preparation phase is dedicated to the definition of two possible environments:

- The “infrastructure” environment, consisting of the set of virtual machines to be started on a corresponding set of cloud providers, where the software components will be deployed.

- The “services” environment, consisting of the set of containers and the references to the corresponding images and start parameters, that will be deployed on the virtual machines.

As a consequence, the allowed values for the path parameter {environment} are: “infrastructure”, “services” or “all”, meaning both of them. In addition to the path parameter, the endpoint requires also a JSON input specifying: the endpoint and credentials for connecting to the cloud broker; the endpoint and credentials for accessing the Git repository where the Application Description resides; the corresponding revision and filepath to pull from the repository.

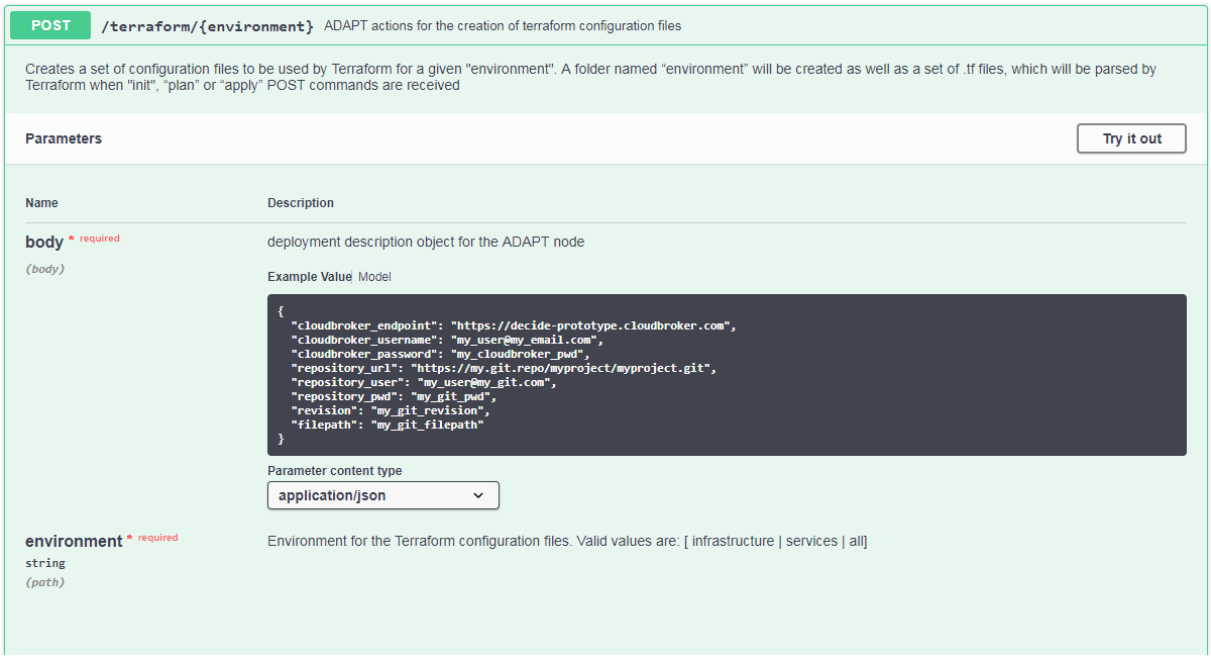

Figure 22. Specification of the POST request for the preparation of Terraform configuration files

The call returns an http response code 202 (request submitted) in case of success or 400 in case of error. It also returns a JSON data structure containing textual description of the result.

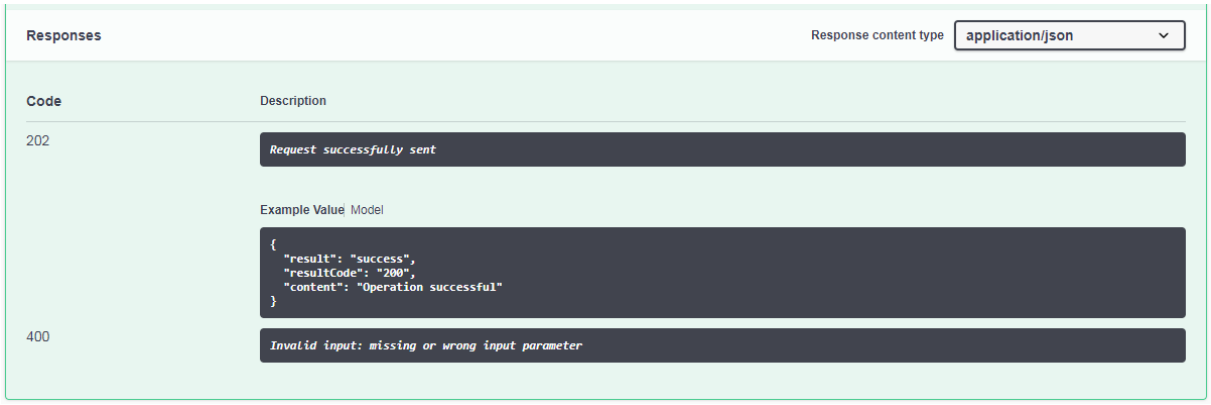

Figure 23. Response of the POST request for the preparation of Terraform configuration files

As described in section 2.2.1, the execution actions that can be submitted to the ADAPT Deployment Orchestrator via REST maps the three Terraform operation ‘init’, ‘plan’ and ‘apply’.

Accordingly, the endpoint documented in Figure 24 allows to submit via POST method one of those operations by specifying the proper {operation} path parameter. Regarding the target application and

environment for the operation, they are specified respectively in the {appName} and {folder} parameters. This information allows to detect the location of the Terraform resource configuration files to be used for the operation.

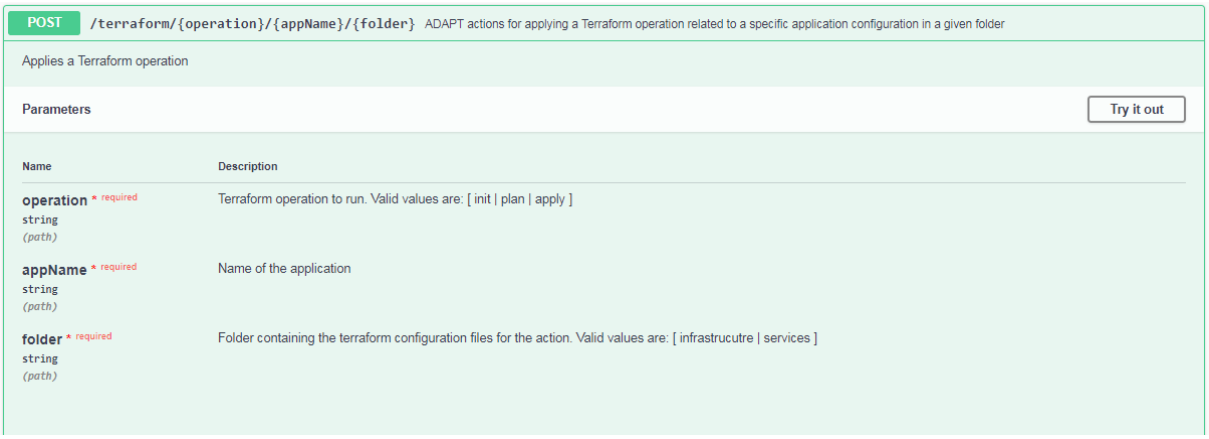

Figure 24. Endpoint for submitting a request for a Terraform operation on a pre-generated set of configuration files

The response, as shown in ***Figure 25***, returns the http response code 200 in case of success, together with a JSON data structure containing the URLs for getting the current status of the submitted operation and for getting the corresponding log. In case of error, it returns the http response code 400.

Figure 25. Response of the request for a Terraform operation

The URL returned in the 'get_log_url' field, represents the endpoint specified in ***Figure 26*** based on GET method. By passing the path variables {operation}, {appName}, {folder} and {op_id} (the unique identifier of an operation, returned by the previous POST operation), it is possible to get the current log of the operation.

Figure 26. Endpoint for getting the log of a submitted operation

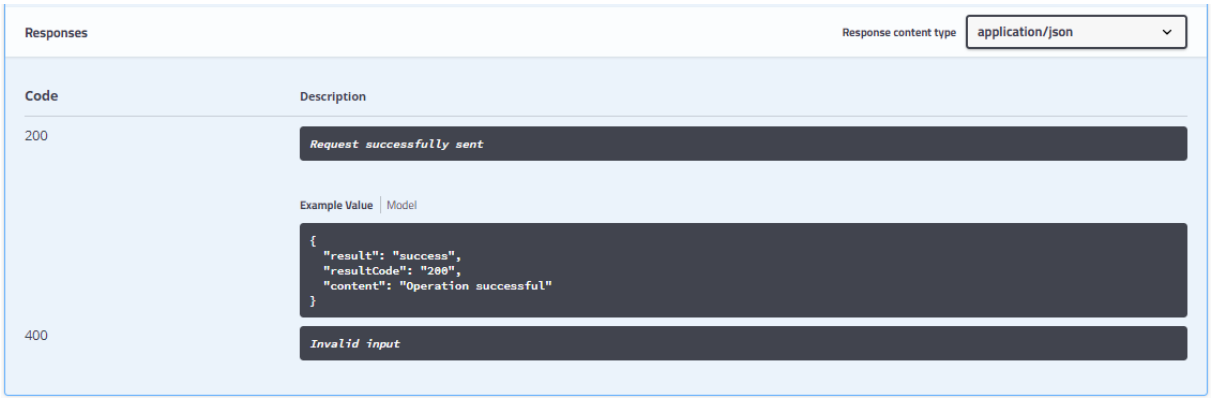

Figure 27. Response for the GET log operation

The log is encapsulated in the "content" field of a JSON response data structure (Figure 27). An example of a real response is shown in the snippet below, which is not much human-readable because of its encapsulation as string into JSON:


```
{"result": "success", "resultCode": "0", "content": "Refreshing Terraform state in-memory
prior to plan...\nThe refreshed state will be used to calculate this plan, but will not
be\npersisted to local or remote state storage.\n\n\n------------------------------------------
------------------------------\n\nAn execution plan has been generated and is shown
below.\nResource actions are indicated with the following symbols:\n  + create\n\nTerraform
will perform the following actions:\n\n+ module.Example02-node-1.cloudbroker_instance.decide-
vm\n      id:                   <computed>\n      disable_autostop:     \"true\"\n
external_ip_address: <computed>\n      instance_type_id:     \"e3ca8e4c-0f91-4e83-9bd9-
4cef88d054a8\"\n      internal_ip_address: <computed>\n      isolated:             \"false\"\n
key_pair_id:          \"fb40a1f3-86b1-4e68-9ab3-049b664139e7\"\n      name:
\"Example02-node-1\"\n      opened_port:          \"22,80,8000-9000,9411\"\n      region_id:
\"4265ddb9-e862-4814-82a4-d6b92f25e8e5\"\n      resource_id:          \"18d07329-07f6-4d59-
b1c1-676f64d1663f\"\n      software_id:          \"21b7ebed-5076-43b6-8351-0e06cf16eedc\"\n\n
+ module.Example02-node-2.cloudbroker_instance.decide-vm\n      id:
<computed>\n      disable_autostop:     \"true\"\n      external_ip_address: <computed>\n
instance_type_id:     \"e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8\"\n      internal_ip_address:
<computed>\n      isolated:             \"false\"\n      key_pair_id:          \"fb40a1f3-86b1-
4e68-9ab3-049b664139e7\"\n      name:                 \"Example02-node-2\"\n      opened_port:
\"22,80,8000-9000,9411\"\n      region_id:            \"4265ddb9-e862-4814-82a4-
d6b92f25e8e5\"\n      resource_id:          \"18d07329-07f6-4d59-b1c1-676f64d1663f\"\n
software_id:          \"21b7ebed-5076-43b6-8351-0e06cf16eedc\"\nPlan: 2 to add, 0 to change, 0
to destroy.\n\n------------------------------------------------------------------------
\n\nNote: You didn't specify an \"-out\" parameter to save this plan, so Terraform\ncan't
guarantee that exactly these actions will be performed if\n\"terraform apply\" is subsequently
run.\n\n"}}
```

The same content extrapolated from the JSON 'content' field and displayed as text output is shown below.

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
------------------------------------------------------------------------
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
+ module.Example02-node-1.cloudbroker_instance.decide-vm
      id:                  <computed>
      disable_autostop:    "true"
      external_ip_address: <computed>
      instance_type_id:    "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8"
      internal_ip_address: <computed>
      isolated:            "false"
      key_pair_id:         "fb40a1f3-86b1-4e68-9ab3-049b664139e7"
      name:                "Example02-node-1"
      opened_port:         "22,80,8000-9000,9411"
      region_id:           "4265ddb9-e862-4814-82a4-d6b92f25e8e5"
      resource_id:         "18d07329-07f6-4d59-b1c1-676f64d1663f"
      software_id:         "21b7ebed-5076-43b6-8351-0e06cf16eedc"
  + module.Example02-node-2.cloudbroker_instance.decide-vm
      id:                  <computed>
      disable_autostop:    "true"
      external_ip_address: <computed>
      instance_type_id:    "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8"
      internal_ip_address: <computed>
      isolated:            "false"
      key_pair_id:         "fb40a1f3-86b1-4e68-9ab3-049b664139e7"
      name:                "Example02-node-2"
      opened_port:         "22,80,8000-9000,9411"
      region_id:           "4265ddb9-e862-4814-82a4-d6b92f25e8e5"
      resource_id:         "18d07329-07f6-4d59-b1c1-676f64d1663f"
      software_id:         "21b7ebed-5076-43b6-8351-0e06cf16eedc"
Plan: 2 to add, 0 to change, 0 to destroy.
------------------------------------------------------------------------
Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

The second URL returned by the POST operation in Figure 25 allows to get the status of the operation. The request must specify the unique operation id of the operation and the result is either 'success', 'running' (if the operation is still ongoing) or a code greater than zero, identifying an exit error code of the requested operation.

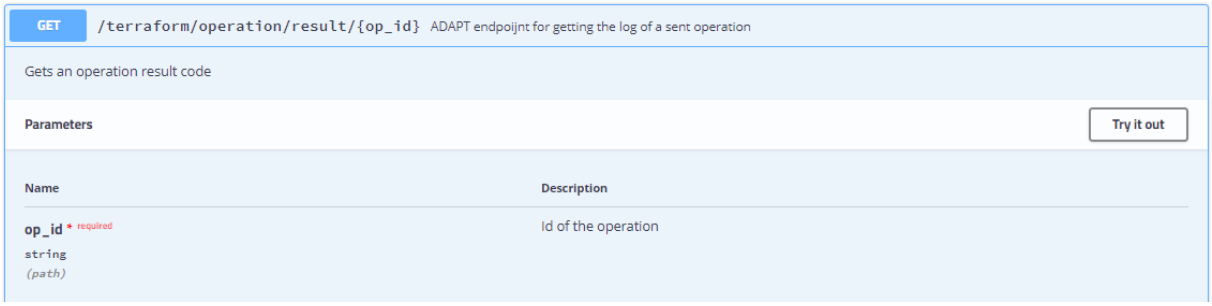

Figure 28. GET Request for getting the result of a submitted operation

2.2.2.2 ADAPT Deployment Orchestrator Preparation Engine

The ADAPT Deployment Preparation Engine is responsible for translating the relevant parts of the Application Descriptor into Terraform configuration files. As introduced in section 2.2.1, this process is carried on leveraging the Jinja 2 templating engine from custom code developed in Python within the Flask framework. ADAPT provides a set of templates defining the overall structure of the Terraform configuration files required to run the DECIDE applications.

As we deal with applications obtained as composition of microservices, each one running in Docker containers, the set of templates is related to the provisioning of virtual machines, the installation of Docker and tools on them and the execution of containers.

Jinja 2 templates can contain placeholders for variable substitution and tags that allow to define logic operations on the sections, so that part of the configuration files can be generated or not based on specific conditions, evaluated and decided by the custom Python code.

In the following, just to give an idea about how the mechanism works, we provide a set of simplified code snippets that describe how a Terraform resource configuration for the creation of a virtual machine can be obtained dynamically from a Jinja 2 template by parsing the Application Descriptor document. We will show it step by step, by first showing the format of a simplified, "static" Terraform configuration file (the target of the transformation), then describing how to obtain it automatically via transformation.

2.2.2.2.1 Static Terraform configuration files

As already described, Terraform provides a command line client, and the related operations are driven by the contents of *all the configuration files with ".tf"* extension *found in the directory from which the Terraform commands are launched*.

To start a VM with Terraform you have to create a configuration file with ".tf" extension similar to the one below, where you define:

- the type of *provider*, with the related endpoint and credentials;
- the set of *resources* you want to create on that provider, with the required parameters.

In the example below, we want to create a virtual machine resource on the `cloudbroker` provider, hence we specify all the required parameters such as: `software_id` (corresponding e.g. to Ubuntu Linux 16.04 64bit), `region_id`, `key_pair_id` (related to your cloud user profile), etc.

In the excerpt below, written in HCL, the values enclosed in "`${}`" tags are parsed by Terraform and interpolated according to the HCL syntax. The fields prefixed with "`var.`" are taken from environment variables or from command line parameters.

```
provider "cloudbroker" {
  username    = "${var.cloudbroker_username}"
  password    = "${var.cloudbroker_password}"
  endpoint    = "${var.cloudbroker_endpoint}"
  timeout     = 60
  max_retries = 5
}

resource "cloudbroker_instance" "my-virtual-machine" {
  software_id = "${var.vm_software_id}"
  resource_id = "${var.vm_resource_id}"
  region_id = "${var.vm_region_id}"
  instance_type_id = "${var.instance_type_id}"
  isolated = "false"
  key_pair_id = "${var.key_pair_id}"
  disable_autostop = "true"
  opened_port = "${var.opened_port}"
  name = "${var.node_name}"
}
```

If we fed `terraform apply` with the above configuration file (and with the related input parameters), it would instantiate a VM on the selected cloud provider. In addition, Terraform would create a file representing the current state of the infrastructure, used as reference for every future operation on the set of configured resources.

One of the many good features of Terraform is that, once we have created such configuration file and made it fully parameterized via variables, we can use it as an importable package (as if it was a function or package in software code) and reuse it from elsewhere.

Therefore, assuming we have saved the above file in the folder `/home/ubuntu/terraform/modules/docker-node`, we can create another Terraform configuration file and reference such package N times, once for every virtual machine we want to start in our deployment schema. Terraform configuration syntax allows also to organize configuration sections into "modules". As an example, if we wanted to start two virtual machines, we would end up with a file similar to the following:

```
module "myapp-node-1" {
  source = "/home/ubuntu/terraform/modules/docker-node"
  node_name = "myapp-node-1"
  cloudbroker_endpoint = "https://xxxx.provider.com",
  cloudbroker_username = "myuser@mycompany.com",
  cloudbroker_password = "YYYYYYYYYYYYYY",
```

```
  vm_software_id = "21b7ebed-5076-43b6-8351-0e06cf16eedc",
  vm_resource_id = "18d07329-07f6-4d59-b1c1-676f64d1663f",
  vm_region_id = "4265ddb9-e862-4814-82a4-d6b92f25e8e5",
  instance_type_id = "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8",
  key_pair_id = "fb40a1f3-86b1-4e68-9ab3-049b664139e7",
  app_name = "myapp",
  opened_port = "22,80,8000-9000,9411"
}
module "myapp-node-2" {
  source = "/home/ubuntu/terraform/modules/docker-node"
  node_name = "myapp-node-2"
  cloudbroker_endpoint = "https://xxxx.provider.com",
  cloudbroker_username = " myuser@mycompany.com ",
  cloudbroker_password = "YYYYYYYYYYYYYY",
  vm_software_id = "21b7ebed-5076-43b6-8351-0e06cf16eedc",
  vm_resource_id = "18d0573d-07f6-4d59-b1c1-4d43ef3aeafc",
  vm_region_id = "4265ddb9-e862-4814-82a4-d6b92f25e8e5",
  instance_type_id = "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8",
  key_pair_id = "fb40a1f3-86b1-4e68-9ab3-049b664139e7",
  app_name = "myapp",
  opened_port = "22,80"
}
...
```

As it can be noticed, in the file above we import with the `source` element the folder where the relevant configuration for the VM creation was stored, and we just assign values to the parameters related to the new VM we want to start.

Now, we can copy/paste a `module` section in the above file and change only the configuration parameters for all the VMs we want to start. The `terraform apply` command will start automatically all of them on the cloud provider. Moreover, if we changed just one of them (e.g. removing a VM element by deleting a `module` section and adding a new one) and run again the `terraform apply` command, Terraform would shut down automatically the deleted VM and start up the new one, leaving unchanged the other ones.

If we consider that we have our infrastructure creation depending from this couple of text files only, we can see how powerful this mechanism can be, as we can put them under code version control and replicate (e.g. restore to a previous one) the state of an infrastructure in the same way as we do for software artifacts.

2.2.2.2.2 Configuration templates

In the case of ADAPT, we want to generate the above configuration at runtime, taking the Application Descriptor as input and Jinja 2 templates as reference.

A corresponding ADAPT template for the resource configuration snippet under consideration is shown below:

```
module "{{ appName }}-{{ dockerHostNodeName }}" {
  source = "/app/modules/decide-docker-node"
  node_name = "{{ appName }}-{{ dockerHostNodeName }}"
  cloudbroker_endpoint = "{{ cloudbrokerEndpoint }}",
  cloudbroker_username = "{{ cloudbrokerUsername }}",
  cloudbroker_password = "{{ cloudbrokerPassword }}",
  vm_software_id = "{{ vmSoftwareId }}",
  vm_resource_id = "{{ vmResourceId }}",
  vm_region_id = "{{ vmRegionId }}",
  instance_type_id = "{{ instanceTypeId }}",
  key_pair_id = "{{ keyPairId }}",
  app_name = "{{ appName }}",
  opened_port = "{{ openedPort }}",
  consul-join-ip = "{{ consulJoinIp }}",
  docker-private-registry-ip = "{{ dockerPrivateRegistryIp }}"
  docker-private-registry-port = "{{ dockerPrivateRegistryPort }}"
}
```

In Jinja 2 templates, sections surrounded by double curly brackets are considered placeholders, that can be filled in via substitution from Python code.

In addition to substitution, templates allow a syntax for more complex logic, such as “if” conditions:

```
...
{% if dockerPrivateRegistryIp and dockerPrivateRegistryPort and dockerPrivateRegistryUser and
dockerPrivateRegistryPassword %}
  image =  "{{ dockerPrivateRegistryIp }}:{{ dockerPrivateRegistryPort }}/{{ imageName }}:{{
imageTag }}"
{% else  %}
...
```

Or “for” loops:

```
...
{% for port in portMapping %}
  ports {
    internal = {{ port['containerPort'] }}
    external = {{ port['hostPort'] }}
  }
{% endfor -%}
...
```

This syntax options are largely used in the ADAPT templates but are not shown in the current example for the sake of simplicity in the exposition.

2.2.2.2.3 Transformation from Application Description to configuration files

The templates are prepared to map the information contained in the Application Description document into Terraform configuration files.

Here follows a snippet of an Application Descriptor related to the creation of virtual machines:


```
{
        "name": "Example02",
        "virtualMachines": [
        {
                "id":  "VM001",
                "cspName": "CloudSigma",
                "cspId": "CSP002",
                "RAM": "1 GB",
                "cores": 1,
                "storage": "20 GB",
                "image": "Ubuntu 16.04",
                "cloudbrokerEndpoint": "https://decide-prototype.cloudbroker.com",
                "cloudbrokerUsername": "myuser@mycompany.com",
                "cloudbrokerPassword": "mypwd",
                "vmSoftwareId": "21b7ebed-5076-43b6-8351-0e06cf16eedc",
                "vmResourceId": "18d07329-07f6-4d59-b1c1-676f64d1663f",
                "vmRegionId": "4265ddb9-e862-4814-82a4-d6b92f25e8e5",
                "instanceTypeId": "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8",
                "keyPairId": "fb40a1f3-86b1-4e68-9ab3-049b664139e7",
                "openedPort": "22,80,8000-9000,9411",
                "consulJoinIp": "54.172.38.173",
                "dockerPrivateRegistryIp": "54.172.38.173",
                 "dockerPrivateRegistryPort": "8200",
                 "dockerHostNodeName": "node-1"
        },
        {
                "id":  "VM002",
                "cspName": "CloudSigma",
                "cspId": "CSP003",
                "RAM": "1 GB",
                "cores": 1,
                "storage": "20 GB",
                "image": "Ubuntu 16.04",
                "cloudbrokerEndpoint": "https://decide-prototype.cloudbroker.com",
                "cloudbrokerUsername": "myuser@mycompany.com",
                "cloudbrokerPassword": "mypwd",
                "vmSoftwareId": "21b7ebed-5076-43b6-8351-0e06cf16eedc",
```

```
            "vmResourceId": "18d07329-07f6-4d59-b1c1-676f64d1663f",
            "vmRegionId": "4265ddb9-e862-4814-82a4-d6b92f25e8e5",
            "instanceTypeId": "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8",
            "keyPairId": "fb40a1f3-86b1-4e68-9ab3-049b664139e7",
            "openedPort": "22,80,8000-9000,9411",
            "consulJoinIp": "54.172.38.173",
            "dockerPrivateRegistryIp": "54.172.38.173",
            "dockerPrivateRegistryPort": "8200",
            "dockerHostNodeName": "node-2"
        }
        ],
...
```

As we can notice, the above JSON defines the parameters for the creation of two virtual machines. Such parameters are the ones required by the template for generating the Terraform configuration files.

A corresponding Python code snippet which manages the Application Descriptor and the template is shown below:

```
...
    #data_loaded is the Json Object representing the input Application Descriptor document
    appName = data_loaded["name"]
    [...]
    cloudbrokerEndpoint = data_loaded["cloudbrokerEndpoint"]
    cloudbrokerUsername = data_loaded["cloudbrokerUsername"]
    cloudbrokerPassword = data_loaded["cloudbrokerPassword"]
    for element in data_loaded["virtualMachines"]:
        fname = infrastructure_dir+"/"+appName+"-vm-"+element["dockerHostNodeName"]+".tf"
        context = {
                'appName': appName,
                'dockerHostNodeName': element["dockerHostNodeName"],
                'cloudbrokerEndpoint': cloudbrokerEndpoint,
                'cloudbrokerUsername': cloudbrokerUsername,
                'cloudbrokerPassword': cloudbrokerPassword,
                'vmSoftwareId': element["vmSoftwareId"],
                'vmResourceId': element["vmResourceId"],
                'vmRegionId': element["vmRegionId"],
                'instanceTypeId': element["instanceTypeId"],
                'keyPairId': element["keyPairId"],
                'openedPort': element["openedPort"],
                'consulJoinIp': element["consulJoinIp"],
        }
```

```
        with open(fname, 'w') as f:
                tf = render_template('vm-tpl.tf', context)
                f.write(tf)
...
```

The snippet code manages a data structure (`data_loaded`) which contains the Application Description in form of a JSON Object. The relevant parameters are taken from the Application Description and put into the '`context`' variable, which maps elements of the descriptor to elements of the Jinja 2 template. Finally, the Jinja 2 '`render_template`' method applies the transformation using the template file (`vm-tpl.tf`) and the `context`, and a configuration file is written to the filesystem into a dedicated folder (see '`fname`' variable above: e.g. it will result, for the first virtual machine defined, in the file '`infrastructure_dir/Example02-vm-node-1.tf`').

The process described above is highly simplified with respect to the actual one, and is only meant to give an idea of what is going on behind the scenes during the generation of configuration files. A lot of additional items are configured for the installation of software on the virtual machines, for the management of certificates, for the setup of a K/V store, etc.

2.2.2.3 ADAPT Deployment Orchestrator Execution Engine

The ADAPT Deployment Execution Engine comes into play when a Terraform action must be executed, on behalf of the REST API component, on the appropriate set of configuration files previously generated by the Preparation Engine. As a matter of fact, it consists of the execution of a 'init', 'plan' or 'apply' Terrafom command from the proper folder where the configuration files reside, and in the returning of the operation result.

As we did in section 2.2.2.2, we provide some further snippets from the configuration files generated in the preparation steps, to show samples of the actions performed by the Execution Engine.

Let's consider below the extended version of the resource definition presented in paragraph 2.2.2.2.1, where we left out for the sake of clarity the sections already defined. Here we add a 'connection' and three 'provisioner' items. A 'connection' item tells Terraform about how to connect remotely to a resource after it has been created. A 'provisioner' is a Terraform component that is used to execute scripts on a local or remote machine as part of resource creation or destruction. Our snippet defines:

- A 'ssh' connection type, and the related user and ssh private key location;
- A 'file' provisioner, which is a Terraform component that allows to copy files from the machine executing Terraform to the remote resource;
- A 'remote-exec' provisioner, which allows to execute commands on the remote virtual machine;
- A 'local-exec' provisioner, which allows to run local commands on the machine executing Terraform.

```
resource "cloudbroker_instance" "my-virtual-machine" {
  software_id = "${var.vm_software_id}"
  [ ...already defined configuration items skipped...]
  connection { #defines the connection type – ssh – and the credentials
      type     = "ssh"
      user     = "ubuntu"
```

```
      private_key =  "${file("/home/ubuntu/terraform/keypairs/test-kp/private-key-openssh")}"

  }

  #copies files from 'source' - local to the Terraform machine - to 'destination' - the VM -

  provisioner "file" {

    source      = "/home/ubuntu/terraform/scripts"

    destination = "/tmp"

  }

  provisioner "remote-exec" { #executes command on the remote VM

    inline = [

      "chmod -R +x /tmp/scripts",

      "sleep 60 && sudo apt-get clean && sudo apt-get -f install",

      "/tmp/scripts/init-vm.sh ${cloudbroker_instance.my-virtual-machine.name}
${cloudbroker_instance.my-virtual-machine.external_ip_address} ${cloudbroker_instance.my-
virtual-machine.internal_ip_address} ${var.consul-join-ip} ${var.docker-private-registry-ip}
${var.docker-private-registry-port}" ]

  }

  provisioner "local-exec" { #executes local commands

    command = <<CMD

      mkdir -p /home/ubuntu/terraform/certs/${cloudbroker_instance.my-virtual-
machine.external_ip_address} \

        && scp -i /home/ubuntu/terraform/keypairs/test-kp/private-key-openssh -
oStrictHostKeyChecking=no -oUserKnownHostsFile=/dev/null -r ubuntu@${cloudbroker_instance.my-
virtual-machine.external_ip_address}:~/.docker/client/keys/
/home/ubuntu/terraform/certs/${cloudbroker_instance.my-virtual-machine.external_ip_address}

CMD  }

}
```

If we submit an 'apply' command with the above configuration Terraform, after the virtual machine creation, connects to it via ssh and copies a set of scripts to the /tmp folder. Such scripts are used to perform installation and configuration steps on the virtual machine, namely the installation of Docker and Consul, the configuration of certificates, etc. The set of scripts is considered part of the ADAPT "helpers" and are documented in the dedicated deliverable D4.10.

After copying the scripts, Terraform uses the 'remote-exec' provisioner to perform a set of remote commands: change rights on folders, update software packages and, finally, execute the scripts that were just copied there with proper parameters.

Finally, Terraform uses the 'local'exec' provisioner to copy back into a dedicated local folder the set of certificates created on the remote virtual machine during the installation and configuration steps, to secure out the Docker daemon socket. Such certificates are needed to run Docker commands on the virtual machine remotely from the Terraform machine (ADAPT), in a secure way.

Terraform operations can be time-consuming, especially the 'apply' operation, because it implies connection to a provider and the time needed for the provisioning of a resource and its setup. Therefore, the Execution Engine applies operations in form of background processes, so that a client can get immediately a response related to the result of the 'request submission' operation. Together with that information, the Engine provides a way to get the current status of the operation and the related logs, as described in section 2.2.2.1.

2.2.2.4 A special component role: ADAPT as ADAPT Deployer

So far we have described the ADAPT Deployment Orchestrator functionalities in terms of the actions that allow the provisioning and deployment of infrastructures and services required to run a DECIDE application. We have extensively documented that the ADAPT Deployment Orchestrator is a microserivce packaged into a container image and run as a Docker container. Anyway, we did not mention anything about a fundamental point: it is clear that ADAPT is a component required in advance, prior to start the whole process of infrastructure and services provisioning. Currently, two possible deployment alternatives have been designed for the ADAPT component (see deliverable D4.1, Chapter 7): one scenario plans to deploy one instance of ADAPT per application, together with the application components; another scenario envisions ADAPT running as a centralized service, and used to manage the operations related to multiple applications.

To fulfill the first scenario, we have designed and implemented a special endpoint which allows a user or a system (e.g. a continuous integration system) to run a volatile instance of ADAPT (on a known virtual machine or on a cloud provider), which is used only to deploy another instance of ADAPT itself, dedicated to a specific application. The deployment of this ADAPT instance happens in the same way as for any other application component, by submitting a dedicated Application Descriptor which only contains information related to a virtual machine to be provisioned for ADAPT, and the settings of the related ADAPT container to launch. Once the ADAPT instance is in place, all the application lifecycle interactions related to the deployment are sent to such instance.

Figure 29 shows the special endpoint which allows the preparation of the deployment of an ADAPT instance.

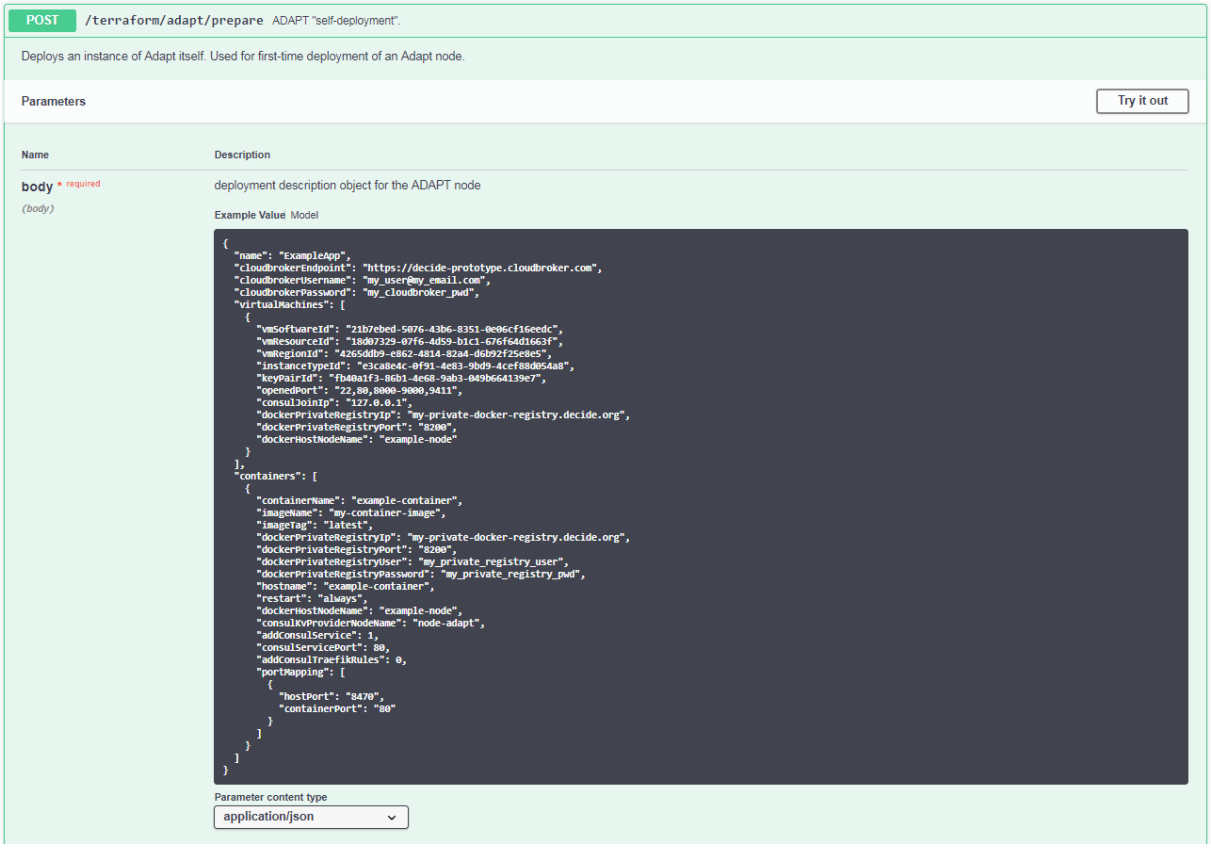

Figure 29. Endpoint for the deployment of an ADAPT instance, performed from another 'volatile' ADAPT instance

The endpoint accepts a POST request and requires a body in Json format, consisting in the description of one virtual machine and one container. A sample of the body content to submit is shown below:

```
{
        "name": "Example02",
        "cloudbrokerEndpoint": "https://decide-prototype.cloudbroker.com",
        "cloudbrokerUsername": "myuser@mycompany.com",
        "cloudbrokerPassword": "mypwd",
        "virtualMachines": [
        {
                "vmSoftwareId": "21b7ebed-5076-43b6-8351-0e06cf16eedc",
                "vmResourceId": "18d07329-07f6-4d59-b1c1-676f64d1663f",
                "vmRegionId": "4265ddb9-e862-4814-82a4-d6b92f25e8e5",
                "instanceTypeId": "e3ca8e4c-0f91-4e83-9bd9-4cef88d054a8",
                "keyPairId": "fb40a1f3-86b1-4e68-9ab3-049b664139e7",
                "openedPort": "22,80,8000-9000,9411",
                "consulJoinIp": "127.0.0.1",
                "dockerPrivateRegistryIp": "54.172.38.173",
                "dockerPrivateRegistryPort": "8200",
                "dockerHostNodeName": "node-adapt"
        }
        ],
        "containers":[
        {
                "containerName":"adapt",
                "imageName":"adapt",
                "imageTag":"latest",
                "dockerPrivateRegistryIp": "xxx.yyy.xxx.yyy",
                "dockerPrivateRegistryPort": "8200",
                "dockerPrivateRegistryUser": "decide-user",
                "dockerPrivateRegistryPassword": "myregistrypwd",
                "hostname": "adapt",
                "restart": "always",
                "dockerHostNodeName": "node-adapt",
                "consulKvProviderNodeName": "node-adapt",
                "addConsulService": 1,
                "consulServicePort": 80,
                "addConsulTraefikRules": 0,
                "portMapping": [
                {
                        "hostPort": "8472",
                        "containerPort": "80"
```

```
                    }
                    ]
            }
            ]
}
```

The POST request has the effect of creating Terraform configuration files for both the infrastructure (the single virtual machine) and the services (the single container) to be created. In this case, we can create all of them in one single step because we know in advance their number and relationship.

Once the configuration files are created, the phases of Initialization, Planning and Execution can be carried on in the same exact way as for a DECIDE application, by invoking the same REST API endpoints described in section 2.2.2.1 for infrastructure and services and the related processes in sections 2.2.1.3 and 2.2.1.4.

The response, depicted in Figure 30 has the same format as the REST API endpoints already introduced.

Figure 30. Response for the ADAPT deployment preparation action

2.2.3 Technical specifications

The current prototype has been developed by integrating the frameworks, tools and libraries listed in section 2.2.1, that we list here for convenience with related versions:

- Flask 0.12.2, comprising Werkzeug and Jinja 2;
- Flask-RESTplus 0.10.1;
- Nginx [19] 1.9.11 http server;
- Terraform 0.10.7;

and requires

- Docker 17.06.2-ce;
- Ubuntu Linux 16.04.

The integration is written in Python 3.6 via a common text editor, and is packaged into a Docker image pushed into a private DECIDE registry. e Terraform configuration files generated automatically are based on the Hashicorp Configuration Language for Terraform 0.10.7.

The REST API documentation has been rendered with Swagger UI [20], deployed on a Docker container and written via .yaml [21] descriptors.

The Terraform plugin for interacting with the cloud broker ACSmI API jhas been developed in the Go Programming Language [22] and is documented in the dedicated deliverable D4.10.

The optional reverse-proxy capability is based on Traefik, and is documented in the dedicated deliverable D4.10 [17].

3 Delivery and usage

3.1 Package information

The ADAPT Deployment Orchestrator is packaged as a Docker image, to be launched as a container on any system running a Docker instance.

The image is currently available on an "unofficial" private Docker registry that we have set up for DECIDE, for development and testing purposes. Credentials are needed to login to the registry and to pull images. There are plans to set up publicly available, official repositories in the next iteration of the project for accessing code, images and all the needed material for running DECIDE components.

3.2 Installation instructions

Being packaged as a Docker image, the ADAPT Deployment Orchestrator does not require specific installation steps.

The only pre-requisite is to have access to a Linux-based system running a Docker daemon.

Currently, there are three options available to get the ADAPT Deployment Orchestrator image:

1. Get the image from the private repository, in case you have been assigned credentials.
2. Get the image from a tar archive, available for download in case you have been authorized.
3. Build the image directly from the ADAPT project code.

Requests for credentials, download urls and access to systems can be submitted by filling the form at the following url: https://www.decide-h2020.eu/contact

3.2.1 Using the image from the private repository

For users who have access to the private repository where the ADAPT Deployment Orchestrator image is stored, the operations to perform to start it are:

- Log into the repository (only for first-time access) with the dedicated Docker command:

```
shell> docker login [registryIp]:[registryPort] -u registryUser -p registryPassword
```

- Execute the Docker 'run' command on the image, mapping a proper port which is open on the host and defining the –d (daemon) flag:

```
shell> docker run -p 8473:80 -d --name adapt [registry_ip]:[registry_port]/adapt:m12
```

- Verify that the container is up and running with the 'ps' command:

```
shell> docker ps
REPOSITORY                    TAG   IMAGE ID      CREATED      SIZE
aaa.bbb.ccc.ddd:nnnn/adapt    m12   40fd3162530b  2 weeks ago  803MB
```

3.2.2 Extracting the image from a tar.gz archive

Images that are saved as tar archives, can be loaded on a local machine with the Docker 'image load' command.

- Download the 'adapt.tar.gz' file from the url received from your request submission
- Unzip the archive with the 'gunzip' tool:

```
shell> gunzip adapt.tar.gz
```

- Load the image from the Docker shell with the following command:

```
shell> docker image load -i /home/ubuntu/adapt.tar

2c40c66f7667: Loading layer [==================================================>]
129.3MB/129.3MB

654f45ecb7e3: Loading layer [==================================================>]
45.45MB/45.45MB

f3ed6cb59ab0: Loading layer [==================================================>]
126.8MB/126.8MB

5616a6292c16: Loading layer [==================================================>]
326.7MB/326.7MB

97108d083e01: Loading layer [==================================================>]
8.043MB/8.043MB

815acdffadff: Loading layer [==================================================>]
62.18MB/62.18MB

325b9d6f2920: Loading layer [==================================================>]
4.608kB/4.608kB

2548e7db2a94: Loading layer [==================================================>]
5.591MB/5.591MB

9a9ce7dcd474: Loading layer [==================================================>]
8.599MB/8.599MB

df08e2c3d6fe: Loading layer [==================================================>]
5.209MB/5.209MB

3c0d8f1e556d: Loading layer [==================================================>]
3.584kB/3.584kB

87e2b99e95df: Loading layer [==================================================>]
3.584kB/3.584kB

5babbba9a986: Loading layer [==================================================>]
3.072kB/3.072kB

433a67f63093: Loading layer [==================================================>]
3.584kB/3.584kB

394c0a98982c: Loading layer [==================================================>]
3.072kB/3.072kB

461de7fb06ff: Loading layer [==================================================>]
5.346MB/5.346MB

b61bb40af46f: Loading layer [==================================================>]
3.584kB/3.584kB

40dc546a568a: Loading layer [==================================================>]
2.048kB/2.048kB

357d65e53b78: Loading layer [==================================================>]
2.048kB/2.048kB

66ddad2c15b4: Loading layer [==================================================>]
3.584kB/3.584kB

31b3b1f0ab7b: Loading layer [==================================================>]
5.5MB/5.5MB

bba5e66e1bc9: Loading layer [==================================================>]
3.072kB/3.072kB

0889a339caa2: Loading layer [==================================================>]
3.072kB/3.072kB

475e51c5e84e: Loading layer [==================================================>]
3.584kB/3.584kB

297a10341f47: Loading layer [==================================================>]
67.77MB/67.77MB
```

```
3cb698cec5c8: Loading layer [==================================================>]
45.06kB/45.06kB

be6e62470c3b: Loading layer [==================================================>]
17.58MB/17.58MB

b8983b6f72e0: Loading layer [==================================================>]
10.25MB/10.25MB

Loaded image: 54.172.38.173:8200/adapt:test
```

- Verify the image is available from the set of local Docker images:

```
shell> docker images

REPOSITORY                      TAG          IMAGE ID            CREATED             SIZE

aaa.bbb.ccc.ddd:nnnn/adapt    m12          40fd3162530b        2 weeks ago         803MB
```

- Run the image:

```
shell> docker run -p 8473:80 -d --name adapt aaa.bbb.ccc.ddd:nnnn/adapt:m12
```

- Verify that the container is up and running with the 'ps' command:

```
shell> docker ps

REPOSITORY                      TAG    IMAGE ID       CREATED      SIZE

aaa.bbb.ccc.ddd:nnnn/adapt    m12   40fd3162530b   2 weeks ago 803MB
```

3.2.3 Building the image from code

If you have access to the code repository, the ADAPT Deployment Orchestrator Docker image can be easily built from the Dockerfile available in the 'Adapt_deployment/adapt-do' folder of the 'WP4' Gitlab project.

- Download the project code from Gitlab as zip or archive file at the url received from your request submission.

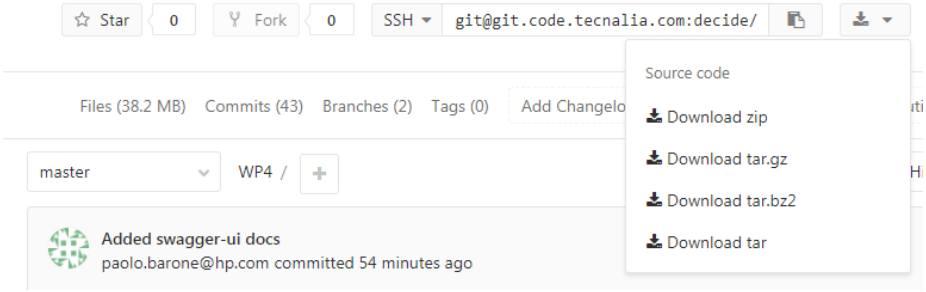

Figure 31. Downloading project code from Gitlab

- Extract the archive into a directory of your choice (e.g. '~/tmp) and go into the directory 'WP4/Adapt_deployment/adapt-do'
- Run the command:

```
shell> sudo docker build -t adapt:m12
```

The above command builds the image 'adapt' with tag 'mytest'.

- Verify the image is available from the set of local Docker images:

```
shell> docker images
```

```
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
adapt                         m12                 a247026d9236        2 minutes ago       811MB
```

- Run the image:

```
shell> docker run -p 8473:80 -d --name adapt adapt:m12
```

- Verify that the container is up and running with the 'ps' command:

```
shell> docker ps
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
adapt                         m12                 a247026d9236        2 minutes ago       811MB
```

3.3 User Manual

Once started, the ADAPT Deployment Orchestrator provides the REST endpoints documented in section 2.2.2.1 for interactions with the other DECIDE components.

It is possible to test the component independent of the rest of the system, for evaluation purposes, by feeding the endpoints with proper input parameters. We provide some sample templates for input data in the folder 'WP4/Adapt_deployment/demo'.

Please consider that ADAPT Deployment Orchestrator is a component which is meant to be used by automatic tools; reproducing the behaviour via human interactions may result difficult due to the amount of manual interactions and configuration steps.

The pre-requisites for successful testing are:

- Having credentials to a git repository where an Application Descriptor document will be fetched by ADAPT.
- Having credentials and permissions to access and start resources on the cloud broker environment configured. Such credentials comprise:
    - Username and password, to interact to the broker API and start resources.
    - SSH keypairs, to connect to the resources created. These are usually created and set up in the user profile creation steps, when registering to a cloud provider service. *In addition, it is also necessary to get the internal ID of the keypairs, which can be extracted automatically by automated tools but requires advanced steps for manual tests*. We are planning to set up a test user with well-known ids and credentials in order to facilitate the verification from human end users.
- Having credentials to access the DECIDE private Docker image registry.

The steps for the test are:

1. Provisioning of an ADAPT Deployment Orchestrator instance specific to the test application.
2. Creation of Terraform configuration files for the infrastructure and for the services environments.
3. Initialization and planning of the infrastructure.
4. Provisioning of the infrastructure.
5. Initialization and planning of the services.
6. Provisioning of the services.
7. Verification of the application.

3.3.1 Provisioning of an ADAPT Deployment Orchestrator instance

- In the 'WP4/Adapt_deployment/demo' folder, open the file 'adapt-descriptor.json' (which contains the description of the cloud resources we want to create: a virtual machine and a container for running an ADAPT Deployment Orchestrator instance) and replace all the fields marked with 'TO_BE_FILLED' with proper credentials.
- Using the 'curl' command line tool, POST the JSON data to the REST endpoint for the deployment of an ADAPT Deployment Orchestrator instance:

```
shell> curl -H "Content-Type: application/json" -X POST --data @adapt-descriptor.json
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/adapt/prepare
```

- Verify that a folder named 'My-Example-App-adapt' is created on the container:
  sudo docker exec -it adapt ls

```
shell> sudo docker exec -it adapt ls
```

- Verify the following subfolders structure is there:

```
shell> sudo docker exec -it adapt ls -R My-Example-App-adapt

My-Example-App-adapt:

infrastructure  services

My-Example-App-adapt/infrastructure:

My-Example-App-vm-node-adapt.tf

My-Example-App-adapt/services:

My-Example-App-container-node-adapt.tf
```

The files with '.tf' extension contains the configurations for Terraform.

- Initialize the infrastructure environment and the services environment:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/init/My-
Example-App-adapt/infrastructure

...

shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/init/My-Example-App-
adapt/services
```

You can poll the urls returned by the commands to verify the status and logs of the request.

- Temporary step for phase 1: to allow ADAPT to create and configure resources via the cloud broker API, it is necessary that it can access the resources via ssh authentication. To enable this, ADAPT must access the private ssh key related to the profile of the user for the cloud broker. Since in the current phase of DECIDE a centralized user profile management component is not in place yet, we must perform a manual step to copy in the ADAPT container such private ssh key according to the following steps:
  - Get the ssh private key from your user profile for the cloud broker. (If you haven't generated it yet, do it now from the user profile management sections in the user interface of the cloud broker).
  - Copy the private key locally on the Linux machine that runs the ADAPT deployer container.

- Copy the private key from the Linux machine to the container, in the folder '/home/ubuntu/terraform/keypairs/My-Example-App/' and name it 'private-key-openssh':

```
shell> sudo docker cp /[key-location-folder]/private-key-openssh
adapt://home/ubuntu/terraform/keypairs/My-Example-App/
```

This process will be automated in the final distribution of DECIDE.

- Create a deployment plan for the infrastructure:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/plan/My-Example-App-
adapt/infrastructure
```

You can poll the urls returned by the commands to verify the status and logs of the request.

- Deploy the infrastructure:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/apply/My-Example-App-
adapt/infrastructure
```

You can poll the urls returned by the commands to verify the status and logs of the request.

Get from the logs the public IP assigned to the virtual machine and mark it down, as it will be needed later and referred as **ADAPT_IP**.

- Create a deployment plan for the service:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/plan/My-Example-App-
adapt/services
```

You can poll the urls returned by the commands to verify the status and logs of the request.

- Deploy the service:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[your_adapt_deployer_ip]:[your_adapt_deployer_port]/terraform/apply/My-Example-App-
adapt/services
```

You can poll the urls returned by the commands to verify the status and logs of the request.

3.3.2 Creation of Terraform configuration files for the infrastructure and for the services environments

- In the DECIDE workflow, the information contained in the Application Descriptor is passed to ADAPT via a Git repository, specific to the application that must be deployed. The file is stored and updated on that repository, and the Application Controller invokes ADAPT by POSTing a JSON data structure containing the needed information to access the repository and the specific revision of the file (please refer to the specification in section 2.2.2.1). To reproduce this mechanism manually, you have to specify your repository data and push a configuration file there, according to the following directions. In the 'WP4/Adapt_deployment/demo' folder, open the file 'app-descriptor.json' (which contains the description of the cloud resources we want to create: two virtual machines and a set of containers for running the Socks Shop application) and replace:
  - all the fields marked with 'TO_BE_FILLED' with proper credentials;

- The fields marked with 'TO_BE_FILLED_WITH_ADAPT_IP' with the IP address of the ADAPT instance generated in the previous step.

Now, you have to push this file into your Git repository, at a well-specified path, and get the revision number of the file. This information is required in the next step. Here follows some directions on how to do it. Let's assume that your Git repository is named "my-app-repo", and that you have cloned it locally via the Git "clone" command as in the following:

```
shell> git clone git@git.code.myrepositories.com:decide/my-app-repo.git
```

You have now to copy the modified 'app-descriptor.json' in your project folder, then commit and push it to the remote repository:

```
shell> git commit -m "updated app descriptor" app-descriptor.json

shell> git push
```

Now, you have to get the revision of the file:

```
shell> git rev-parse HEAD

06f926e7bc8a6bd40703874dd9c05ed71e6ca49a
```

The returned hexadecimal code is the revision number, needed in the next step together with the information related to your Git repository.

- In the 'WP4/Adapt_deployment/demo' folder, open the file 'preparation-post-data.json' and replace:
  - all the fields marked with 'TO_BE_FILLED_XXX' with proper data.
- Using the 'curl' command line tool, POST the json data to the REST endpoint for the creation of configuration files for the infrastructure and for the services:

```
shell> curl -H "Content-Type: application/json" -X POST --data @preparation-post-data.json
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/all
```

- Verify that a folder named 'My-Example-App' is created on the container:

```
shell> sudo docker exec -it adapt ls
```

- Verify the following subfolders structure is there:

```
shell> sudo docker exec -it adapt ls -R My-Example-App

My-Example-App:

infrastructure  services

My-Example-App/infrastructure:

My-Example-App-vm-node-1.tf

My-Example-App-vm-node-2.tf

My-Example-App-adapt/services:

My-Example-App-container-carts-db.tf

My-Example-App-container-carts.tf

My-Example-App-container-catalogue-db.tf

My-Example-App-container-catalogue.tf

My-Example-App-container-front-end.tf
```

```
My-Example-App-container-orders-db.tf
My-Example-App-container-orders.tf
My-Example-App-container-payment.tf
My-Example-App-container-queue-master.tf
My-Example-App-container-rabbitmq.tf
My-Example-App-container-shipping.tf
My-Example-App-container-traefik-private.tf
My-Example-App-container-user-db.tf
My-Example-App-container-user.tf
My-Example-App-container-zipkin.tf
My-Example-App-network-node-1-carts-network.tf
My-Example-App-network-node-1-user-network.tf
My-Example-App-network-node-2-catalogue-network.tf
My-Example-App-network-node-2-orders-network.tf
My-Example-App-network-node-2-shipping-network.tf
My-Example-App-services-common.tf
```

The files with '.tf' extension contains the configurations for Terraform.

3.3.3 Initialization and planning of the infrastructure

- Initialize the infrastructure environment:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/init/My-Example-App/infrastructure
```

You can poll the urls returned by the command to verify the status and logs of the request.

- Create a deployment plan for the infrastructure:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/plan/My-Example-App/infrastructure
```

You can poll the urls returned by the command to verify the status and logs of the request.

3.3.4 Provisioning of the infrastructure

- Deploy the infrastructure:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/apply/My-Example-App/infrastructure
```

You can poll the urls returned by the command to verify the status and logs of the request.

3.3.5 Initialization and planning of the services

- Initialize the services environment:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/init/My-Example-App/services
```

You can poll the urls returned by the command to verify the status and logs of the request.

- Create a deployment plan for the services:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/plan/My-Example-App/services
```

You can poll the urls returned by the command to verify the status and logs of the request.

3.3.6 Provisioning of the services

- Deploy the service:

```
shell> curl -H "Content-Type: application/json" -X POST
http://[ADAPT_IP]:[ADAPT_PORT]/terraform/apply/My-Example-App/services
```

You can poll the urls returned by the command to verify the status and logs of the request.

3.3.7 Verification of the application

If the above steps succeeded, you would be able to access the Socks Shop application at the url:

- http://[node-2-ip]:80

Where [node-2-ip] is the address of the virtual machine 2 started by the 'infrastructure apply' operation.

3.4 Licensing information

The plan is to release the ADAPT Deployment Orchestrator component, developed by HPE, as open source software. HPE has to follow an internal process with reviews and decisions at corporate level to decide and approve the license under which to release the developed software. Unfortunately, this process takes time and it is not yet completed at the time of writing, therefore the licensing information for the released software is not yet available. However, credentials can be provided under request, download urls and access to systems can be requested by filling the form at the following url: https://www.decide-h2020.eu/contact.

4 Conclusions

This deliverable has described the first ADAPT prototype that will be implemented during the first year of the project. The prototype is focused on the deployment and adaptation functionalities, which will be carried out by the Deployment Orchestrator component.

The main functionalities of this component have been listed, along with an explanation of how it fits within the whole DECIDE workflow. An introduction to the ShockShop application, the test application that will be used to validate DECIDE, can be found as well.

The architecture of the Deployment Orchestrator has also been analyzed. The component has been thoroughly described, including exposed APIs and tools used. Furthermore, the deliverable provides information about the packaging of the component, the requirements to install it and the steps a user would have to follow to test it.

Lastly, the license under which this piece of software will be released has been specified.

References

[1] Flask, [Online]. Available: http://flask.pocoo.org/. [Accessed 28 11 2017].

[2] Werkzeug, "The Python WSGI Utility Library," [Online]. Available: http://werkzeug.pocoo.org/. [Accessed 28 11 2017].

[3] Jinja, [Online]. Available: http://jinja.pocoo.org/. [Accessed 28 11 2017].

[4] Flask-RESTPlus, [En línea]. Available: https://flask-restplus.readthedocs.io/en/stable/. [Último acceso: 28 11 2017].

[5] HashiCorp, "Terraform," [Online]. Available: https://www.terraform.io/. [Accessed 28 11 2017].

[6] DECIDE Consortium, “D2.4 Detailed architecture v1,” 2017.

[7] DECIDE Consortium, “D4.1 Initial DECIDE ADAPT Architecture,” 2017.

[8] Weaveworks and Container solutions, "Sock Shop," [Online]. Available: https://microservices-demo.github.io/. [Accessed 28 11 2017].

[9] Weaverworks, “Sock shop application architecture,” [Online]. Available: https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md.

[10] Pivotal, "Cloud native applications," [Online]. Available: https://pivotal.io/cloud-native. [Accessed 28 11 2017].

[11] Traefik, [Online]. Available: https://traefik.io/. [Accessed 28 11 2017].

[12] HashiCorp, "Consul," [Online]. Available: https://www.consul.io/. [Accessed 28 11 2017].

[13] HashiCorp, "Terraform Interpolation," [Online]. Available: https://www.terraform.io/docs/configuration/interpolation.html. [Accessed 28 11 2017].

[14] Python Software Foundation, "Python," [Online]. Available: https://www.python.org/. [Accessed 28 11 2017].

[15] Wikipedia, “Web Server Gateway Interface,” [Online]. Available: https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface. [Accessed 28 11 2017].

[16] Wikipedia, "Infrastructure as Code," [Online]. Available: https://en.wikipedia.org/wiki/Infrastructure_as_Code. [Accessed 28 11 2017].

[17] DECIDE Consortium, “D4.10 Initial multi-cloud application helpers,” 2017.

[18] HashiCorp, «Github. HCL,» [En línea]. Available: https://github.com/hashicorp/hcl. [Último acceso: 28 11 2017].

[19] NGINX, "Wiki," [Online]. Available: https://www.nginx.com/resources/wiki/. [Accessed 28 11 2017].

[20] SmartBear Software, "Swagger," [Online]. Available: https://swagger.io/swagger-ui/. [Accessed 28 11 2017].

[21] C. C. Evans, "YAML," [Online]. Available: http://yaml.org/. [Accessed 28 11 2017].

[22] Go, "The Go Programming Language," [Online]. Available: https://golang.org/. [Accessed 28 11 2017].

[23] A. Sandor, "Github. Microservices demo," [Online]. Available: https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md. [Accessed 28 11 2017].