Chapter 5
The SODALITE Runtime Environment

Indika Kumara, Giovanni Quattrocchi, Dragan Radolović, Kamil Tokmakov, Jesús Ramos Rivas, and Willem-Jan Van Den Heuvel

Abstract Modern applications need to be dynamically orchestrated on heterogeneous infrastructures for reasons such as performance, regulation compliance, or cost. This chapter presents the SODALITE runtime environment that can deploy, monitor, and manage applications on heterogeneous infrastructures consisting of Cloud, HPC, and Edge resources. The SODALITE runtime deploys the applications in the target infrastructures based on the deployment artifacts generated by the SODALITE model-driven approach presented in Chap. 3. It can also monitor the deployed applications and their infrastructure resources, generate alerts, and adapt application deployments.

5.1 Introduction

Modern computing infrastructures consist of heterogeneous, software-defined, high-performance computing environments and resources, including Cloud servers, Edge

I. Kumara (✉) · W.-J. Van Den Heuvel
Jheronimus Academy of Data Science, Tilburg University, Tilburg, The Netherlands
e-mail: i.p.k.weerasinghadewage@tilburguniversity.edu

W.-J. Van Den Heuvel
e-mail: w.j.a.m.v.d.heuvel@jads.nl

K. Tokmakov
High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany
e-mail: kamil.tokmakov@hlrs.de

G. Quattrocchi
Politecnico di Milano, Milan, Italy
e-mail: giovanni.quattrocchi@polimi.it

D. Radolović
XLAB, Ljubljana, Slovenia
e-mail: dragan.radolovic@xlab.si

J. R. Rivas
ATOS, Barcelona, Spain
e-mail: jesus.ramos.external@atos.net

© The Author(s) 2022

E. Di Nitto et al. (eds.), *Deployment and Operation of Complex Software in Heterogeneous Execution Environments*, PoliMI SpringerBriefs,
https://doi.org/10.1007/978-3-031-04961-3_5

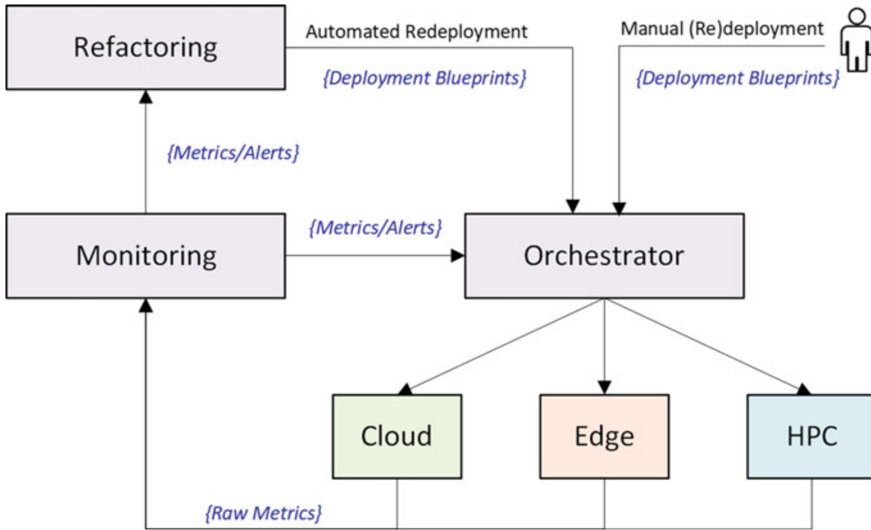

Fig. 5.1 The architecture of the SODALITE runtime environment

accelerators, HPC clusters, and Serverless platforms. Advanced applications require complex and heterogeneous deployments that match their components with the infrastructure that offers the best performance fulfilling their requirements. In this context, SODALITE aims to address this heterogeneity by providing a toolset that enables developers and infrastructure operators to achieve faster development, deployment, and execution of applications on different heterogeneous infrastructures In particular, the runtime layer of SODALITE is responsible for the orchestration, monitoring, and adaptation of applications on these infrastructures.

Figure 5.1 shows the high-level architecture of the *SODALITE* runtime environment, which consists of the components *Orchestrator*, *Monitoring System*, and *Refactoring System*. *Orchestrator* is responsible for (re)deploying a given application on the Cloud-Edge-HPC hybrid infrastructures by executing IaC scripts as necessary. It receives the initial deployment model (from a developer) or a new alternative deployment model (from *Refactoring* System) as a TOSCA model instance. The developers can use the *SODALITE IDE* to create deployment models for applications and trigger their (re)deployment. *Monitoring System* collects different metrics and events from both the application and Cloud-Edge-HPC infrastructure. It can also emit alerts, which are complex events over metrics or simple events. In response to the events from *Monitoring System*, *Refactoring* may decide to modify and reconfigure the current deployment model instance of the application.

In the rest of this chapter, we discuss the *SODALITE* runtime environment in detail. We first present the design and capabilities of the *Orchestrator* (Sect. 5.2), highlighting its deployment and redeployment operations. Next, we focus on the approach to support data management and data transfer between the cloud and HPC clusters (Sect. 5.3) and on monitoring applications and infrastructure with the *Mon-*

itoring System (Sect. 5.4). Finally, our support for the adaptation of the deployment models (Sect. 5.5) and managing resources at runtime (Sect. 5.6) is discussed.

5.2 Orchestrating Applications

The *SODALITE Orchestrator* is capable of deploying, undeploying, and redeploying applications over heterogeneous infrastructures. The applications to be deployed are packaged as CSAR (TOSCA Cloud Service Archive) files. Our *Orchestrator*, namely xOpera,[1] is a meta-orchestrator that coordinates multiple low-level resource orchestrators. xOpera is compliant with TOSCA YAML v1.3 standard.

5.2.1 Architecture of Orchestrator

Figure 5.2 shows the high-level architecture of the *Orchestrator*, which mainly consists of *Meta-Orchestrator*, *IaC-based Orchestration Layer*, *Image Registry*, *Authentication and Authorization Manager*, and *Application Data Manager*.

- **Meta-Orchestrator** coordinates the low-level resource orchestrators of the execution platforms through the *IaC-based Orchestration Layer* to deploy and manage applications. SODALITE runtime currently supports five key types of execution platforms: Edge (Kubernetes), private Cloud (OpenStack and Kubernetes), pub-

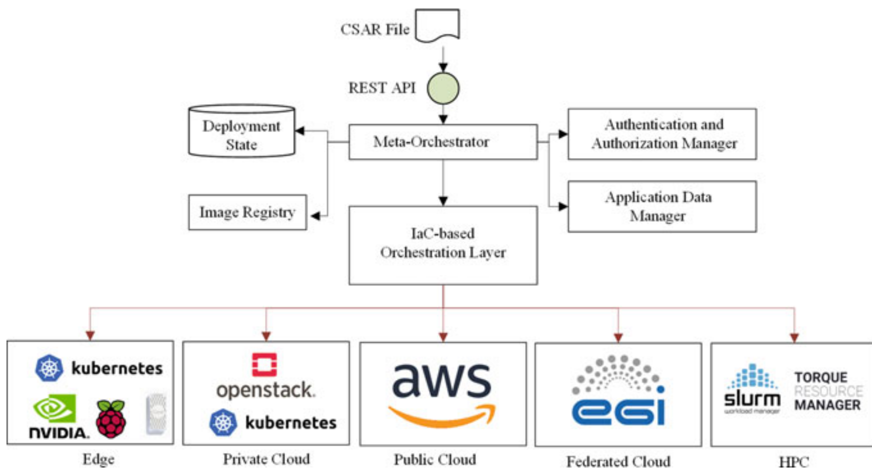

Fig. 5.2 Architecture of the *Orchestrator*

[1] https://github.com/xlab-si/xopera-opera.

lic Cloud (AWS), federated Cloud (EGI OpenStack), and HPC (TORQUEand SLURM).

- **IaC-based Orchestration Layer** is responsible for acquiring, allocating, and configuring resources from the execution platforms, and deploying and configuring application components using those resources. SODALITE uses Ansible as the IaC tool. Ansible playbooks realize the lifecycle operations for nodes/relationships in a deployment model in TOSCA.
- **Image Registry** stores container images. It can be a private or public repository, for example, Docker Hub or Google Container Registry. The private repositories should provide REST APIs to pull the images through IaC.
- **Authentication and Authorization Manager** handles the user and secrets management across the whole SODALITE stack. It applies role-based access control and token-based authentication. Each TOSCA blueprint and deployment is associated with a project domain with specific roles, an access type to which requires a token with specific JWT (JSON Web Token) claims. Each critical orchestration operation such as deployment, undeployment, deployment updates can only be performed by providing a valid access token. The implementation uses the Keycloak[2] identity and access management solution.
- **Application Data Manager** incorporates various transfer protocols and endpoints to achieve transparent data management across multiple infrastructure providers. Section 5.3 discusses data management capabilities in detail.

5.2.2 *Orchestration APIs*

The *Orchestrator* exposes its capabilities as RESTFul APIs.

- Blueprint Management. The blueprints (CASR files) can be added, removed, updated, and queried. The blueprints can also be accessed and managed through Git user accounts. Blueprint metadata such as id, version, and name can be obtained.
- Deployment Management. The applications can be deployed, and undeployed based on their blueprints. The deployment status and history can be obtained, and the failed deployments can be resumed. Application redeployment is requested by submitting the new version of the application deployment model. The *Orchestrator* will calculate the difference between the deployed instance and the new blueprint and will (un)deploy it. The new blueprint can be another version of the previously used blueprint or some version of another blueprint.
- Blueprint Inspection. The *Orchestrator* can also validate the syntax of TOSCA blueprints based on the version v1.3 of TOSCA Simple YAML Profile specification. It also supports calculating the differences between the current deployment state and a new blueprint.

[2] https://www.keycloak.org/.

5.3 Managing Application Data

The components of heterogeneous applications are deployed across various execution platforms to utilize the capabilities of the different platforms. As such, one component can use HPC resources for better performance of batch computation, while another Cloud resources for better scalability and elasticity. Furthermore, this is also a possibility of processing on Edge devices. Using such a hybrid setup, where dependent components of the applications are deployed across various platforms, might require data transfers from one platform into another, and the orchestration system must support them. In this section, we explore the possibilities of data transfers between application components deployed across multiple infrastructure targets.

The current data management services found in scientific communities (e.g. FTS3,[3] Rucio,[4] DynaFed,[5] OneData[6]) mostly focus on HPC and Cloud storage platforms and do not cover Edge, IoT and serverless platforms. Inversely, Edge and serverless platforms (e.g. MQTT,[7] Apache Kafka,[8] Fledge,[9] Apache NiFi[10] and StreamSets[11]) target stream and Cloud storage platforms, but do not target HPC platforms. Therefore, in order to meet the SODALITE objectives for supporting heterogeneous infrastructures, data management for mentioned platforms shall be provided.

The RADON project[12] has developed a set of standard TOSCA libraries[13] for lifecycle management of data pipelines, which is inline with IaC-based orchestration in SODALITE. The concept of data pipeline allows composition of application components (e.g. microservices, serverless functions or self-contained components) as independently deployable and scalable pipeline tasks with the data movement and possible data transformation between the components ([4]). As an underlying technology for data pipelines, Apache NiFi service is used. It exposes a REST API for data flow management between pipeline elements (blocks) and also provides connectors to various platforms and storage systems, such as S3, GCS, Azure, Apache Kafka, HDFS, MQTT, HTTP, (S)FTP, etc. This enables fetching data from one storage provider and pushing data to another provider as a pipeline task. As part of the collaboration with RADON project, we studied the feasibility of using data pipelines as components to unify data management between various heterogeneous platforms. SODALITE extended RADON's TOSCA and IaC libraries for data pipeline man-

[3] https://github.com/cern-fts/fts3.

[4] https://github.com/rucio/rucio.

[5] https://lcgdm.web.cern.ch/dynafed-dynamic-federation-project.

[6] https://github.com/onedata/onedata.

[7] https://mqtt.org/.

[8] https://github.com/apache/kafka.

[9] https://github.com/fledge-iot/fledge.

[10] https://github.com/apache/nifi.

[11] https://github.com/streamsets/datacollector.

[12] https://radon-h2020.eu/.

[13] https://github.com/radon-h2020/radon-particles.

agement, which currently target multi-cloud storage and serverless platforms, with GridFTP support—a common file transfer protocol used in HPC. This enables an interoperability between HPC, Cloud storage types and data streams. Extended TOSCA libraries can be found in the joint RADON-SODALITE organization.[14]

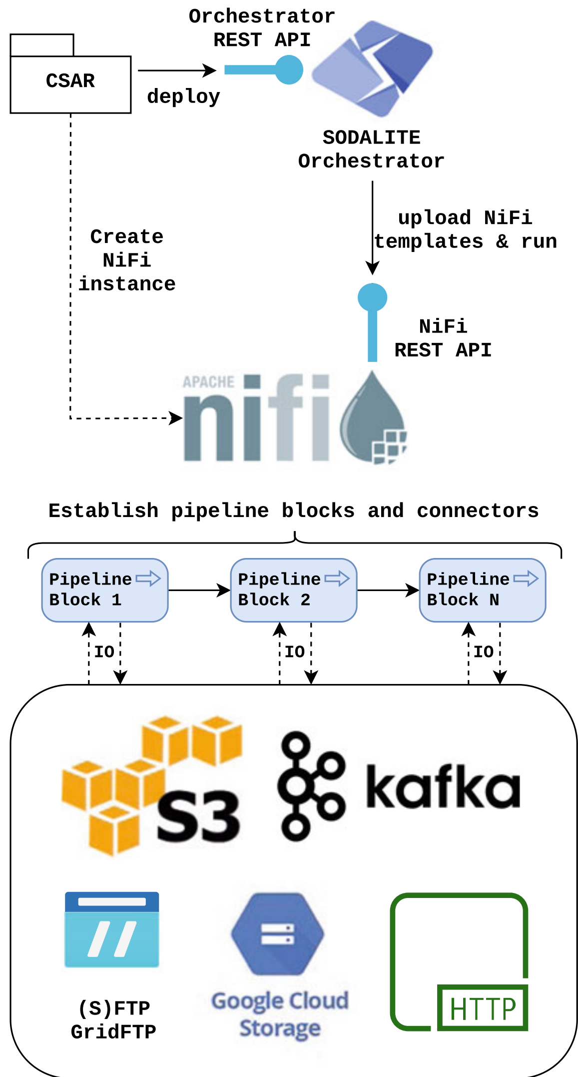

Fig. 5.3 IaC data pipeline management architecture

[14] https://github.com/RADON-SODALITE.

Figure 5.3 depicts an overview on data pipeline management. The Orchestrator exposes a REST API for deployment of a CSAR, which contains a TOSCA application topology description along with TOSCA node types, IaC and other dependencies. The application topology must specify pipeline blocks and connections between them, as well as specify which NiFi instance to use and whether the orchestrator must create a new instance or use the existing instance of NiFi. On the lower level, to instantiate a pipeline block, the orchestrator uses NiFi REST API to upload a NiFi XML template that describes the pipeline block. NiFi then registers the template and returns the ID of the pipeline, which in turn is used by the Orchestrator to request NiFi for the pipeline execution. Same happens for every pipeline block in the application topology. At this point, the registered pipeline blocks are established and functional, and the Orchestrator relies on NiFi instances to perform data movements between the pipeline blocks or move data to a certain storage system as a pipeline task.

A PipelineBlock [3], depicted in Fig. 5.4, is an entity that executes pipeline tasks, such as data processing, API calls invocation, fetching data from or pushing data to remote storage systems or stream platforms, etc. The PipelineBlock may contain input (DataIngestionQueue) and output (DataEmissionQueue) queues for buffering input and resultant data. Using these queues, multiple PipelineBlocks can be connected sequentially, forming a group of PipelineBlocks. Similarly, multiple groups can also be connected using InputPipes—gateways for receiving input data from the previous group or external data source, and OutputPipes—for forwarding resultant data to the next group or external data sink.

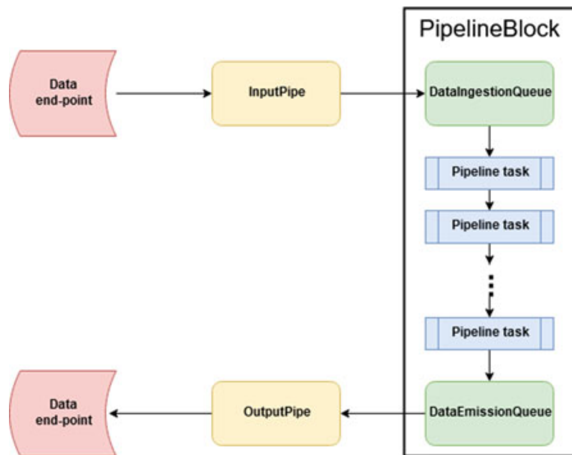

Fig. 5.4 Architecture of data pipeline block [3]

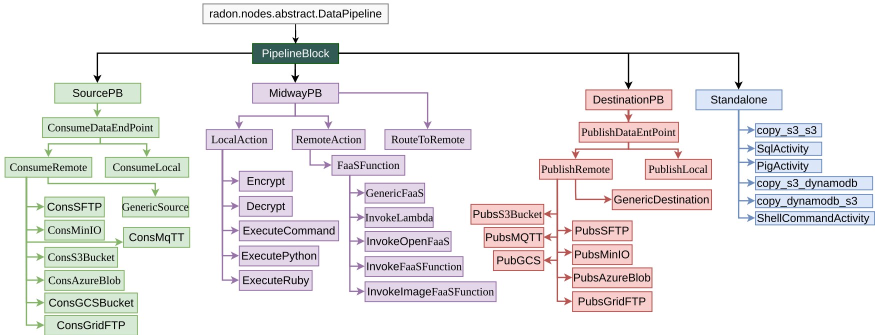

Fig. 5.5 A hierarchy of featured TOSCA node types

The IaC data management features are limited to the capabilities and functionalities offered by Apache NiFi. We mainly focus on utilising NiFi for multi-protocol and multi-platform data movement, abstracted in TOSCA and IaC. Current structure of featured TOSCA node types for data pipeline blocks is presented in Fig. 5.5, and they can be categorised into four classes of pipeline blocks and can be extended:

1. Source pipeline blocks—for consuming data from a data endpoint (e.g. HTTP, FTP, S3, GCS, Kafka, MQTT, GridFTP).
2. Destination pipeline blocks—for publishing data to a data endpoint (e.g. HTTP, FTP, S3, GCS, Kafka, MQTT, GridFTP).
3. Midway pipeline blocks—for executing data processing tasks (e.g. local data processing, encryption, invoke serverless FaaS).
4. Standalone pipeline blocks—for performing independent activities (e.g. copy from S3 to S3).

Listing 5.1 depicts an example of two data pipeline blocks that allow the data transfer between GridFTP server and an S3 bucket. *PubsS3Bucket* is a data pipeline block that publishes data to an S3 bucket, whereas *ConsumeGridFtp* consumes data from a GridFTP server. The *ConsumeGridFtp* pipeline block has a *connectToPipeline* requirement, which points to the *PubsS3Bucket* pipeline block, therefore connecting these data pipeline blocks and performing data transfers between GridFTP and S3.

```
PubsS3Bucket:
    type: radon.nodes.datapipeline.destination.PubsS3Bucket
    properties:
        BucketName: "gridftp-result-bucket"
        cred_file_path: "/home/user/.aws/nifi_credentials"
        schedulingStrategy: "EVENT_DRIVEN"
        schedulingPeriodCRON: "* * * * * ?"
        name: "sendToS3"
        Region: "eu-central-1"
    requirements:
        - host: NiFi

ConsumeGridFtp:
    type: radon.nodes.datapipeline.source.ConsumeGridFtp
    properties:
        gridftp_port: 2811
        intermediate_folder: "/tmp/nifi_gridftp_subscribe/"
        schedulingStrategy: "EVENT_DRIVEN"
        schedulingPeriodCRON: "* * * * * ?"
        name: "receieveFromGFTP"
        gridftp_user: "user"
        gridftp_host: "gridftp.server.example.com"
        gridftp_cert_path: "/home/user/.globus"
        gridftp_directory: "~/target_dir/"
    requirements:
        - host: NiFi
        - connectToPipeline:
            node: PubsS3Bucket
            relationship: con_ConnectNifiLocal
            capability: ConnectToPipeline
```

List. 5.1: Snippet of TOSCA node template with S3 publisher and GridFTP consumer that allow data transfer from GridFTP server to an S3 bucket

5.4 Monitoring Applications and Infrastructures

The deployed application is continuously monitored, allowing the user to consult the state of the deployment as well as making the data available to other components such as *Deployment Refactorer*. The main requirements the monitoring system must meet are:

1. Dynamic addition and deletion of monitored components
2. Monitoring of different levels (infrastructure, runtime environment and application)
3. Transparency to the user
4. Possibility to add alerting rules for specific components
5. Access to metrics filtered by deployment, so that they are only available to the deployment owner.

We designed the monitoring system to meet these requirements (see Fig. 5.6). The APIs that appear on the figure are the ones used by components outside of the monitoring system. The system is composed of the following elements:

- Monitoring server: It collects metrics from the exporters and saves them, exposing a service to other components to get monitoring data. It also detects alerting rule violations and triggers alerts.

- Alert manager: When an alert is triggered by the monitoring server, it sends the alert to the subscribed services, like the *Deployment Refactorer*.
- Rule server: Allows the dynamic creation of alerting rules, registering them in the monitoring server.
- Exporter registry: Allows the dynamic registration and deregistration of exporters in the monitoring server.
- Dashboard server: Stores dashboards and makes them available to users. The dashboards aggregate the metrics stored in the monitoring server and present them in a meaningful way. There are different types of dashboards depending on the type of exporter they aggregate the metrics from.
- Dashboard registry: It allows the dynamic creation of dashboards in the dashboard server, also manages the user permissions to access each dashboard so that each user can only view the metrics belonging to their deployments.
- Exporters: They collect the metrics from the monitored resources and expose them to the monitoring server. There are different types depending on which resource they monitor.

The rest of this section will discuss these components and their implementations in detail. carries out, as well as some details on their implementation.

5.4.1 Exporters

The main task of exporters is probing a resource to extract data, treat this data if necessary to convert it to meaningful metrics, and expose these for the monitoring service. There are different types of exporters depending on the resource they can extract metrics from.

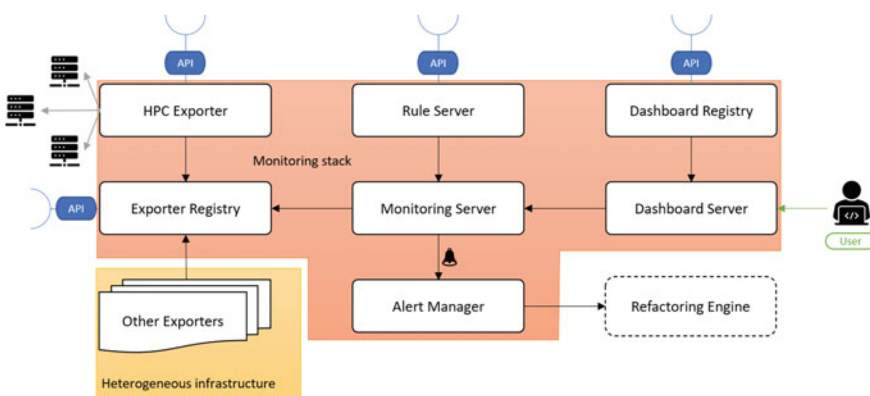

Fig. 5.6 The architecture of the *SODALITE@RT* monitoring system

Exporters are made up of different collectors, each of them collecting a different type of metrics. For example, to expose a VM's OS metrics the node exporter is used, which, among others, has the CPU, netstat, and file-system collectors to monitor CPU usage, file-system status, and operating system's network statistics respectively.

In SODALITE there are 5 exporter types in use:

- Node exporter: Extracts a machine's OS metrics, such as CPU and RAM usage, context swaps, and file-system stats. One of them is deployed on each virtual machine and edge node.
- Skydive exporter: Exposes the infrastructure's network statistics such as network flow and traffic metrics.[15]
- HPC Exporter: To monitor the jobs submitted to an HPC as well as the HPC's infrastructure status (available nodes, queue status, etc.).
- Edge exporter: It contains accelerator-specific collectors for any attached heterogeneous accelerators (e.g., Edge TPU and GPU). They provide specific insight into the attached accelerators. This may include aspects such as the number of devices available, the load average, or thermal properties.

The Ansible playbooks that are responsible for setting up nodes also deploy the exporters associated with each node. The configuration parameters for exporters can be provided using TOSCA node properties. Listing 5.2 shows a snippet of an Ansible playbook that installs the EdgeTPU exporters into the edge nodes in a Kubernetes cluster. It uses the Ansible modules for executing the relevant Helm charts. By having the exporter creation and registration in the exporter registry in the standard Ansible playbooks, these actions are transparent to the user, who only needs to provide high-level settings for the exporters.

```
1  tasks:
2  -name: Add Prometheus Community repository
3      community.kubernetes.helm_repository:
4          name: prometheus-community
5          repo_url: https://prometheus-community.github.io/helm-charts
6  -name: Add Adaptant repository
7      community.kubernetes.helm_repository:
8          name: adaptant
9          repo_url: https://adaptant-labs.github.io/charts/adaptant
10 -name: Install EdgeTPU exporter
11     community.kubernetes.helm:
12         name: edgetpu-exporter
13         chart_ref: adaptant/edgetpu-exporter
14         release_namespace: "{{namespace}}"
```

List. 5.2: Snippet of an Ansible playbook for installing the EdgeTPU exporter.

5.4.1.1 HPC Exporter

The HPC exporter is a special case. It connects to an HPC front-end through SSH and runs commands to gather information about the queues, node status, and job statistics.

[15] http://skydive.network/.

In order to carry out this task it needs to have the user's SSH credentials, it also must be deployed on a different machine than the HPC front-end itself, since it needs to use port 9110 to expose the metrics to the monitoring server, and most HPC's would not allow arbitrary ports to be open for security reasons. In order to solve this issue, there is only a single HPC exporter deployed alongside the rest of the monitoring system's core components, as part of SODALITE's back-end. The exporter, aside from exposing an endpoint for the monitoring server to collect metrics, also exposes 3 more endpoints, which form an API:

- /create: It accepts a JSON object containing the necessary configuration to monitor a given HPC front-end. It creates a collector in the HPC exporter and registers its association with the deployment ID and user that created it. All the metrics exposed by this collector will include labels to identify which deployment it is a part of.
- /delete: Removes the selected collector
- /addJob: It accepts a JSON object which contains a deployment ID and a Job ID, and adds the Job ID to the collector's list of jobs to monitor.

All the calls to this API are secured with the same JWT system used across SODALITE to ensure that only authorized users can create, delete and modify collectors and that users can only delete and modify their own collectors. This system is also used to retrieve the SSH credentials needed to connect to the HPC's front-end from the secrets vault. This way no SSH credentials need to travel unsecured as part of the configuration settings.

5.4.2 Monitoring Server

The monitoring server is the central piece of the monitoring system. Its job is to collect the metrics from all the registered exporters and store them in a database, in a time-series format. This component offers an API so that other components can query data. These queries are done in the server's own query language, which allows for data aggregation and filtering. This API is mainly used by the Dashboard server since in SODALITE, users do not have direct access to this monitoring server. This is due to security considerations since users should only have access to metrics coming from their own deployments' components.

The monitoring server also allows the definition of alerting rules, which consist of:

- An condition written in the monitoring server's query language that, when met for a certain amount of time, triggers an alert.
- For how long the condition must be met so that the alert is triggered.
- The severity of the alert
- The contents of the alert that will be sent to the Alert Manager when the alert is triggered. This may include information about the instance that generated this alert, or other context information.

An example of an alert that triggers when a VM's CPU usage has exceeded 75% (within the deployment with monitoring ID `7acf2a5-da51s4da-as44d1c1a8ftr`) is shown in Listing 5.3. When an alert is triggered, the monitoring server sends it to the Alert Manager, which is the component tasked with distributing the alerts to the subscribed services. The main service that consumes the alerts is the Refactoring Engine, which uses the alerts to trigger refactoring actions based on the content of such alerts.

```
1 alert: HighCPULoad
2 expr: 1- (avg by(instance,os_id) (irate(node_cpu_seconds_total{mode="idle",
  ↪  monitoring_id="7acf2a5-da51s4da-as44d1c1a8ftr"}[5m])))> 0.75
3 for: 5m
4 labels:
5   severity: warning
6 annotations:
7   summary: 'CPU load above 75% load (instance {{ $labels.instance }})'
8   description: 'CPU load is > 75%\n  VALUE = {{ $value }}\n  Monitoring ID: {{
  ↪  $labels.monitoring_id }}\n INSTANCE: {{ $labels.instance }}'
```

List. 5.3: An alerting rule for indicating high CPU usage in a node.

Rules cannot be dynamically added to the monitoring server by default. That is why the rule server exists. It consists of an API that allows the registration and removal of alerting rules on the monitoring server. The IDE has a module that allows users to create, edit, and add alerting rules to deployments they own through this component (see Sect. 3.2.5).

5.4.3 *Exporter Registry*

Much like with alerting rules, the monitoring server does not allow for the dynamic registration of exporters. All the endpoints it scrapes metrics from must be known at the time of deployment. In order to allow dynamic creation of resources during the lifetime of the platform, a key aspect of SODALITE, the exporter registry is used.

The targets for the monitoring server are actually endpoints offered by the exporter registry. There is an endpoint for each type of exporter. When a new exporter is deployed (for example, when a new VM is created as part of an application deployment), the address of the new exporter is registered with the exporter registry, which ensures that when the monitoring server scrapes metrics, it also scrapes from all its registered exporters.

5.4.4 *Dashboard Server*

The users see all the metrics collected by the monitoring system on a dashboard that has been created for each deployment and exporter type. Each dashboard can be

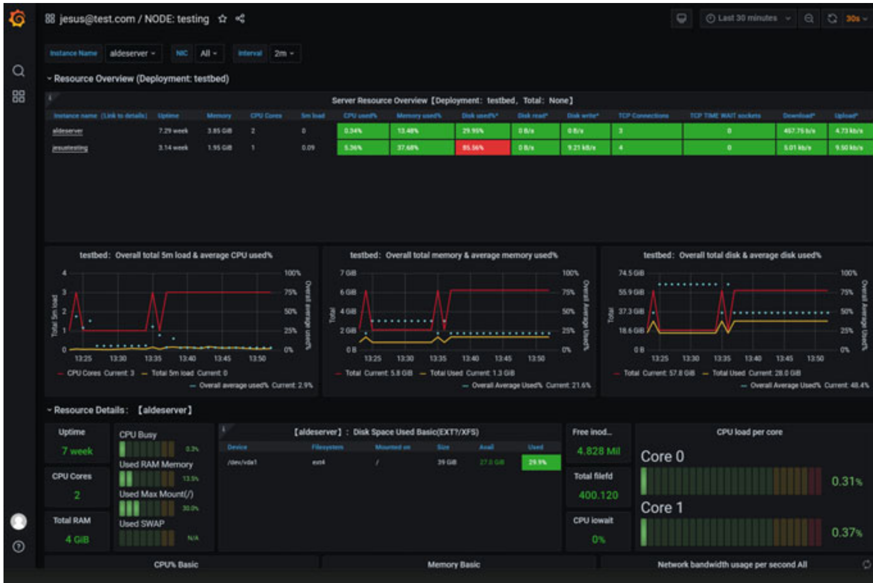

Fig. 5.7 Example of dashboard showing metrics collected by node exporters

accessed on a web browser and is a set of graphs and indicators that aggregate the metrics from the corresponding Monitoring ID and exporter type. The component that hosts the dashboards is the dashboard server, which also takes care of authenticating users to make sure only users that have access to a Monitoring ID can view its dashboards.

An example of a dashboard for node exporters is shown in Fig. 5.7, it includes a list of the VMs that are part of the dashboard's deployment and allows the user to select one of them to view detailed metrics such as CPU, memory, or disk usage graphs, as well as other indicators like the number of context swaps.

To ensure that a dashboard only contains information from a certain deployment, the Monitoring ID the dashboard belongs to is hard-coded when it is created, which means that there must be a way to create dashboards dynamically, a task fulfilled by the dashboard registry.

5.4.5 Dashboard Registry

The dashboard registry consists of an API that exposes a number of endpoints, all secured by the JWT generated by Keycloak when the user logs in the IDE:

- /dashboard (POST): The registry creates one dashboard per exporter type from a set of templates for the required monitoring ID on the dashboard server. It also

sets the dashboard permissions so that only the user that made this call is able to view these dashboards.
- /dashboard (DELETE): Deletes the dashboards that belong to the provided monitoring ID, if the user has the access to them.
- /dashboard/user (GET): Returns the URL of all the user's dashboards
- /dashboard/deployment/<monitoring_id> (GET): Returns the URL of the dashboards that belong to the given monitoring ID, only if the user has access to them.

By using this system we can ensure the security of the monitoring system and at the same time allow for great scalability and flexibility, key values of SODALITE.

5.5 Adapting Application Deployments

In response to the data collected and events received from *Monitoring System*, *Deployment Refactorer* decides and carries out the desired changes to the current deployment of a given application. In this section, we present the architecture and the key capabilities of *Deployment Refactorer*.

5.5.1 Architecture of Deployment Refactorer

Figure 5.8 shows the architecture of the *Deployment Refactorer*. The overall deployment adaptation logic can be codified as an ECA (Event-Condition-Action) policy. Policy Engine can enact and manage such policies. In order to build complex policies, *Deployment Refactorer* provides a set of utilities: *Workload Predictor*, *Performance Predictor*, *Deployment Configuration Selector*, and *Performance Anomaly Detector*. *Workload Predictor* uses linear and polynomial regression models to forecast the workload (the number of requests for the next period). Given the predicted workload and the deployment options used, *Performance Predictor* can predict the performance metrics. If the current deployment model variant cannot meet the performance goals, *Deployment Configuration Selector* can be used to find an alternative deployment model from the allowed set of deployment model variants (expressed in the deployment variability model). *Performance Anomaly Detector* can be used to continuously monitor the current deployment for anomaly behaviors, and generate alerts. The predictive ML models used by each of these components are stored in the *Predictive Model Repository*. The features used by such models are stored in the *Feature Store*. In the rest of this section, the support for the key capabilities of Deployment Refactorer is presented.

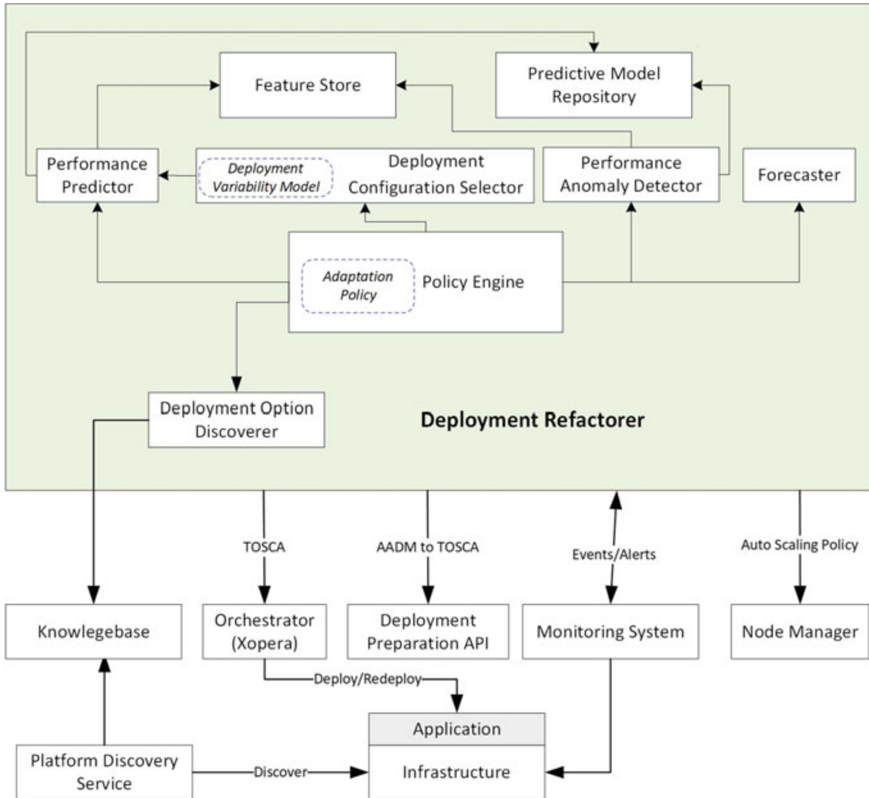

Fig. 5.8 Architecture of Deployment Refactorer

5.5.2 Policy-Based Deployment Adaptation

To allow a software engineer to define the deployment adaptation decisions, we provide an ECA (event-condition-action) based policy language. Figure 5.9 the key concepts of the policy language. A policy consists of a set of ECA rules.

- **Events and Conditions**. A *condition* of a rule is a logical expression of events. We consider two common types of events pertaining to the deployment model instance of an application: deployment state changes and application and resource metrics. The application and resource metric events include (raw or aggregated) primitive metrics collected from the running deployment, for example, average CPU load, as well as alerts or complex events that represent predicates over primitive metrics, for example, the above-mentioned *HostHighCPULoad* alert.
- **Actions**. The actions primarily include the common change operations (*Add*, *Remove*, and *Update*) and the common search operations (*Find* and *EvalPredicate*) on nodes, relations, and their properties. Additionally, the custom actions

can be implemented and then used in the deployment adaptation rules, for example, actions for predicting the performance of a particular deployment model instance or predicting workload. To ensure the safe and consistent changes to the deployment model instance, *Deployment Refactorer* makes the change operations to a local representation (a Java Object model) of the deployment model (represented using the concept of models@runtime [2]). Once the adaptation rules in a rule session are executed, *Deployment Refactorer* translates the current local object model to a TOSCA file and calls the update API operation of the Orchestrator with the generated file.

The *Deployment Refactorer* cadaptation policies. It supports the addition, removal, and update of policies. It can parse given policies, process events, and execute the policies. The policy rules are triggered as their conditions are satisfied, and the desired changes are propagated to the deployment model instance.

Listing 5.4 shows an example of a deployment adaptation rule that reacts to the events *HostLowCpuLoad* and *HostHighCpuLoad* by moving the snow application between a medium VM and a large VM.

5.5.3 Data-Driven Deployment Switching

We use a machine learning-based approach to implement deployment switching. In particular, we use a performance model that can predict the performance of a given deployment alternative in terms of deployment options used by the variant. The deployment options represent architectural and resource selection decisions that

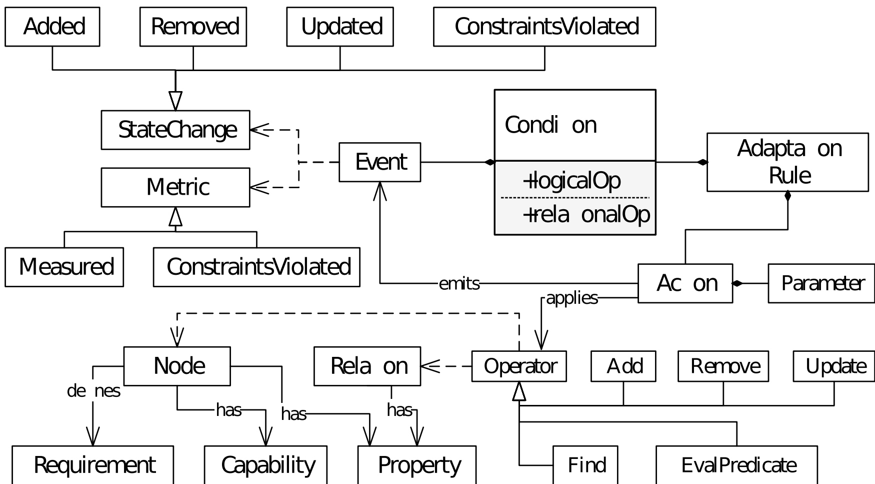

Fig. 5.9 Meta-model of the deployment adaptation policy language

```
1  rule "HostHighCpuLoad"
2    when
3      $f1 : Alert(name == "HostHighCpuLoad")
4    then
5      Node snowvm2node = refMgt.findMatchingNodeFromRM("( ?name = \"snow-vm_new_2\" )
           ");
6      AADMModel aadmModel = refMgt.getAadm();
7      aadmModel.addNode(snowvm2node);
8      List<Node> nodes = refMgt.getNodeMatchingReqFromRM("snow/snow-vm-2");
9        for (Node node : nodes) {
10             aadmModel.addNode(node);
11       }
12     aadmModel.updateProperty("snow-skyline-extractor", "ports", "8080:8080");
13     aadmModel.updateRequirement("snow-skyline-extractor", "host", "snow-docker-host
           -2");
14     ...
15     refMgt.saveAndUpdate();
16 end
17
18 rule "HostLowCpuLoad"
19   when
20     $f1 : Alert(name == "HostLowCpuLoad")
21   then
22     AADMModel aadmModel = refMgt.getAadm();
23     aadmModel.removeNode("snow-vm-2");
24     aadmModel.removeNode("snow-docker-host-2");
25     aadmModel.removeNode("snow-docker-registry-certificate-2");
26     aadmModel.updateProperty("snow-skyline-extractor", "ports", "8082:8080");
27     aadmModel.updateRequirement("snow-skyline-extractor", "host", "snow-docker-host
           ");
28     ...
29     refMgt.saveAndUpdate();
30 end
```

List. 5.4 A snippet of a deployment adaptation rule

are made by the experts when creating deployment models, for example, inclusion or exclusion of a web cache, use of a cluster mode, and use of a large VM or small VM. The initial performance models are built offline, and at runtime, based on the monitored data, the models are retrained as necessary, for example, if the model accuracy drops below a predefined threshold. Figure 5.10 shows the design time and runtime workflows of our deployment switching approach.

We first model the allowed set of deployment variants for a given application based on the deployment decisions, their instantiations, and their inter-dependencies. Based on this deployment variability model, we select an initial valid sample of deployment variants and measure the performance of each variant in the sample. We use the measured application performance dataset to train a predictive model and then evaluate its performance. If the model prediction accuracy is unacceptable, the performance of an additional sample of deployment variants is measured and used to retrain the model.

To model the allowed variations in the deployment topology of an application, we use the feature modeling technique, which is a widely-used variability modeling technique [1], and is also supported by open source and commercial tools. We used FeatureIDE,[16] which is an Eclipse plugin that can be installed into the SODALITE IDE. A feature model can represent the commonalities and variations in a family of artifact variants as configuration options and their inter-dependencies and other constraints. An artifact can be a software system, application, design model, and more. By respecting all the constraints defined by the feature model, we can select a subset of configuration options (called a feature configuration), which represents a valid artifact variant or a family member. The feature leaf nodes can represent the component deployment options, which are the unique assignments of application components to VMs. Similarly, the feature group nodes (non-leaf nodes) and their hierarchical organization capture the logical decomposition of the deployment decisions. For example, the web server and the database cache can be deployed together (co-deployment) or separately (separate-deployment), which can be modeled using an XOR feature group.

To sample a variability model (i.e., to select a subset of deployment model variants), there exist many sampling strategies proposed by the research literature in the performance modeling of configurable systems. In our current implementation, we experimented with three sampling techniques: random sampling, T-wise sampling, and dissimilarity sampling.

To collect data for offline training of models, we used the benchmarking approach due to our preference for the accuracy of the performance data. For each deployment variant in the sample, we select the component deployment options, create them in the target environment, subject the application to a range of workloads using a load testing tool, and collect the performance metrics (response time) per workload.

To build the predictive models, the current literature in configurable systems has used many different learning algorithms, including traditional machine learning algorithms as well as deep learning models. In our current implementation, we used the following three models: Decision Tree Regression (DTR), Random Forest Regression (RFR), Multilayer Perceptron Neural Network (MLP). The performance

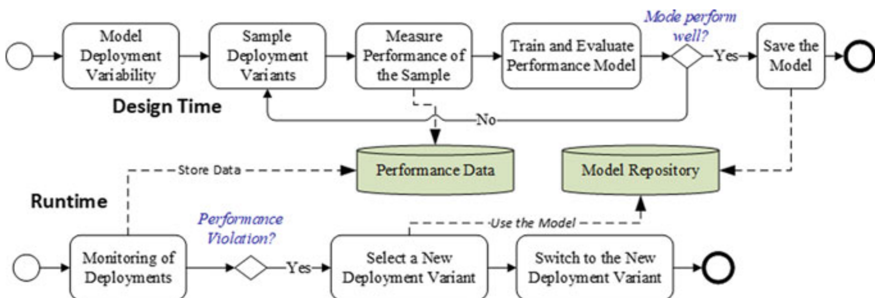

Fig. 5.10 Workflows of building and using predictive models for making deployment switching decisions

[16] https://featureide.github.io/.

prediction model is used at runtime to predict the performance of a given deployment variant for a given workload. If the current deployment model cannot satisfy the performance goals, then, a deployment model variant that can meet the performance goals is selected. For more information, we refer the readers to the relevant publication at [5].

5.5.4 Data-Driven Anomaly Detection

An anomaly can be defined as a rare event where the system behavior deviates from what is standard, normal, or expected. For example, a service could use an anomalous amount of resources, or the service network exhibits traffic anomalies. The ability to detect anomalies and trigger corrective actions is critical to maintain the quality of service and to prevent runtime service failures and undue usage of resources. In the SODALITE project, we consider the anomalies as Chaos that can occur in a containerized microservice system, for example, *CPU Hog*, *Pod Delete*, and *Pod Network Corruption.*

We aim to detect whether a compute node or a cluster is anomalous and classify the type of the anomaly at runtime, independent of the microservice application that is running on the compute node or the cluster. To detect and classify anomalies, we use a machine learning-based approach (see Fig. 5.11). We first build the machine learning models at the design time by utilizing historical resource usage and performance data that are collected from healthy and anomalous situations. Then, at runtime, we apply these models to the monitoring data from the application deployment to detect anomalies. The monitoring data is also used to update and adapt the models. By utilizing various capabilities of SODALITE (e.g., monitoring and alerting, platform and resource discovery, policy-based adaptation, and redeployment), the runtime detection and correction of anomaly behaviors are supported. We used three types of machine learning algorithms to build anomaly predictors: Decision Tree, Random Forest, and AdaBoost. All three models were able to predict anomalies with at least 97% accuracy.

5.5.5 TOSCA Compliant Refactoring Option Discovery

The *Deployment Refactorer* uses refactoring options for adapting a given deployment model. A refactoring option represents one or more nodes in a deployment model. *Deployment Option Discoverer* uses semantic web technologies for discovering TOSCA-compliant resources and deployment model fragments or refactoring options. It considers constraints on node attributes, node requirements, node capabilities, and node policies. The semantic annotation of resource models including

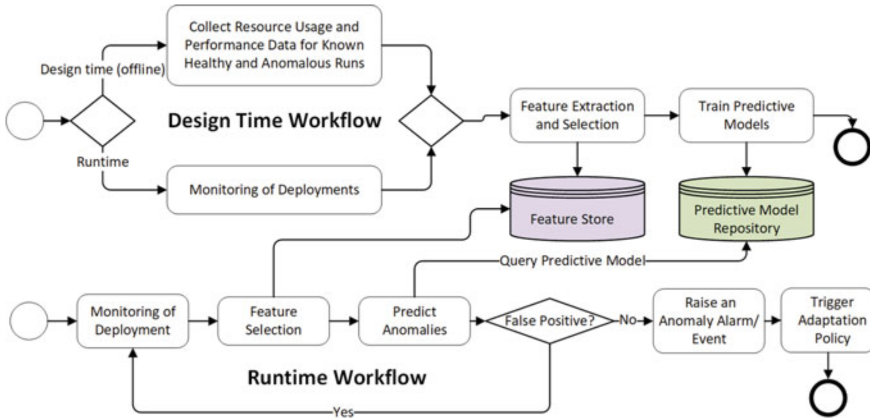

Fig. 5.11 An Overview of our Anomaly Detection Approach : **a** Training Workflow (Offline and Runtime), **b** Prediction Workflow (Runtime)

the attached policies enables machine reasoning which is then used for both the discovery and the composition of resources.

```
select DISTINCT ?node ?description ?nodetype
where {
        ?nodetype rdfs:subClassOf tosca:tosca.nodes.Compute .
        ?node rdf:type ?nodetype .
        OPTIONAL {?node dcterms:description ?description .}
        FILTER (?nodetype != tosca:tosca.nodes.Compute ) .
        FILTER (?node != owl:Nothing) .
        ?node soda:hasContext ?context .
    {
    ?context tosca:properties ?concept .
    OPTIONAL {?concept DUL:classifies snow:flavor .}
    OPTIONAL {?concept tosca:hasDataValue ?flavor .}
    }
    {?context tosca:properties ?concept1 .
        OPTIONAL {?concept1 DUL:classifies snow:image .}
        OPTIONAL {?concept1 tosca:hasDataValue ?image .}
    }
        FILTER ( ( ?flavor = "m1.small" ) && ( ?image = "centos7" ))
}
```

List. 5.5: Snippet of the SPARQL Query Generated for Retrieving Nodes Matching the Constraint flavor = "m1.small" && image = "centos7"

The *Deployment Option Discoverer* performs matchmaking by executing the SPARQL queries over the ontologies in the knowledgebase. It provides high-level system support to the *Deployment Refactorer* to allow searching for resources, for example, find (a logical expression over node properties). It has the SPARQL query templates for different types of resource matchmaking. The query templates are instantiated with the input data received through the high-level API operations. Listing 5.5 shows a snippet of the SPARQL Query generated for retrieving nodes matching the constraint flavor = “m1.small” && image = “centos7”.

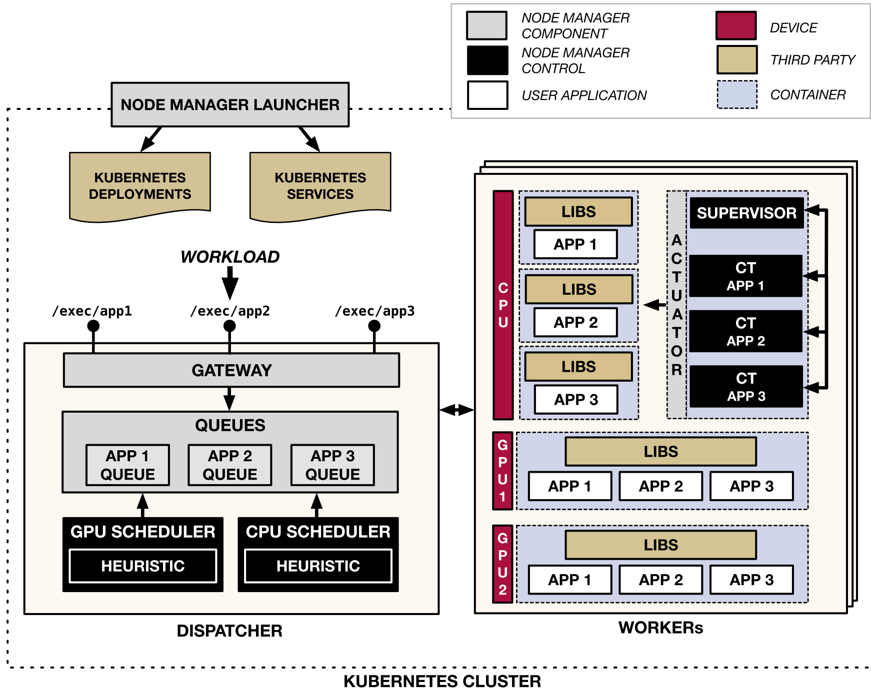

Fig. 5.12 Architecture of the *SODALITE* Node Manager

5.6 Vertical AutoScaling and Smart Scheduling

Component *Node Manager* aims to deploy and manage applications on existing resources deployed by the SODALITE users or by component *Deployment Refactorer*. In particular, *Node Manager* provides two main features that are complementary to the ones of the Deployment Refactorer: (i) it provides fast vertical scaling of computing and memory resources, and (ii) it provides scheduling to select the proper executor (i.e., a CPU or a GPU) for a given request.

While the model of *Node Manager* is general enough to support different platforms, its current implementation is based on TensorFlow.[17] TensorFlow is one of the most used frameworks for developing and executing Machine Learning applications that can be run on both CPUs and GPUs. Specifically, *Node Manager* uses TensorFlow Serving, an image of the container provided by TensorFlow that allows ML applications to be run as Docker[18] containers.

[17] https://www.tensorflow.org.

[18] https://www.docker.com.

The architecture of *Node Manager* is shown in Fig. 5.12. *Node Manager* deploys containerized applications on a Kubernetes[19] cluster and provides two main types of node: a dispatcher and a set of worker nodes (Kubernetes nodes). The dispatcher receives the incoming requests from the users of the different apps and acts as a smart load balancer.

It first stores the requests in a dedicated queue and, according to the requirements (Service Level Agreement) of the applications and their performance, it schedules them for execution on a fast GPU or a CPU. In the workers, dedicated control theoretical planners (CT in the figure) vertically scale the CPU cores allocated to each containerized application given the monitored performance of the system and the work done by the GPU. On each worker node, a *Supervisor* is also deployed to manage resource contention among the different running applications.

As soon as a user submits a trained model, along with its SLA, *Node Manager Launcher* generates or updates required Kubernetes *deployments* and *services* to let the system deploy and manage the application containers.

To exploit the CPUs and GPUs of a node, each application is bound to a specific *device*. In particular, given m applications selected to be deployed onto a worker node, *Node Manager* provisions:

- m containers containing one model each, and binds them to the node's CPU(s)
- one container, containing all the apps, for each GPU
- one container that includes the control theoretical planners for all the models, the *Supervisor*, and one actuator implemented as a Kubernetes volume.

Since we assume that the worker depicted in Fig. 5.12 comes with two GPUs, and it manages three applications, Node Manager deploys six containers in total.

The deployment and configuration of the *Node Manager* and of users' applications can be done using TOSCA blueprints.

In order to deploy applications, users of *Node Manager* must create an AADM similar to the example provided in Listing 5.6.

[19] https://kubernetes.io.

```yaml
inputs:
  node-manager-ip:
    type: string

 node_templates:
  node-manager-deploy:
   type: sodalite.nodes.nodemanager.deploy
   properties:
    endpoint: get_input: node-manager-ip
    models:
     - name: app1
      version: 1
      sla: 0.4
      alpha: 0.5
      profiled_rt: 0.15
      tfs_model_url: https://app1.com/archive/v1.tar.gz
      initial_replicas: 1
     - name: app2
      version: 1
      sla: 0.3
      alpha: 0.5
      profiled_rt: 0.15
      tfs_model_url: https://app2.com/archive/v1.tar.gz
      initial_replicas: 1
    available_gpus: 1
    tfs_image: tensorflow/serving:latest
    k8s_api_configuration:
     apiVersion: v1
     clusters:
      - cluster:
        certificate-authority: "/home/root/.minikube/ca.crt"
        server: http://32.21.111.161:8080
       name: minikube
     contexts:
      - context:
        cluster: minikube
        namespace: default
        user: minikube
       name: minikube
     current-context: minikube
     kind: Config
     preferences: {}
     users:
      - name: minikube
       user:
        client-certificate: "/home/root/.minikube/profiles/minikube/client.crt"
        client-key: "/home/root/.minikube/profiles/minikube/client.key"
```

List. 5.6: Deploying applications through *Node Manager*

Users must use a node template of type `sodalite.nodes.nodemanager.deploy` in order to be able to correctly interface with the deployed *Node Manager*. The type requires the definition of the following properties:

- `endpoint`: the endpoint to contact to deploy the applications, by default it is equal to <Node Manager IP>:5000/deployment
- `models`: an array that contains essential metadata of each application. In particular each element must specify:
  - the `name` of the application and its `version`.
  - the `sla` (maximum allowed response time for the application) and its nominal response time (`profiled_rt`). This value must be obtained by measuring on average the end-to-end latency of a request with an empty queue.

  - a tuning parameter of control theoretical planner (`alpha`). The higher its value, the faster the controller's responses to sudden changes in the performance. A too high value, could lead to oscillating resource allocation (default value is set to be 0.5).
  - a URL pointing to the TensorFlow model of the application (`tfs_model_url`)
  - the initial number of container replicas for the app (`initial_replicas`).

- `available_gpus`: the amount of gpus available on each machine
- `tfs_image`: the Docker image of the TensorFlow serving container (default is set to `tensorflow/serving:latest`).
- `k8s_api_configuration`: metadata regarding the Kubernetes clusters (clusters, users and contexts) as specified at https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/.

5.7 Conclusion and Future Work

The SODALITE platform enables the deployment of complex applications on heterogeneous Cloud-Edge-HPC infrastructure. It supports the modeling of heterogeneous application deployments using the TOSCA open standard, deploying such applications based on created models, and monitoring and adapting application deployments. SODALITE runtime employs machine learning-based approaches to switching between different deployment variants and detecting performance anomalies. SODALITE runtime includes the distributed control-theoretical planners that can support vertical resource elasticity for containerized application components that use both CPU and GPU resources.

We will be conducting future work in two key directions. On the one hand, we will further develop the SODALITE *runtime* by incorporating new infrastructures such as Open FaaS and Google Cloud, and by completing the integration of the runtime layer within the overall SODALITE stack. On the other hand, the monitoring and deployment adaptation support will be extended for the Edge-to-Cloud continuum.

References

1. Berger T et al (2013) A survey of variability modeling in industrial practice. In: Proceedings of the seventh international workshop on variability modelling of software-intensive systems. VaMoS '13. Pisa, Italy: Association for Computing Machinery. ISBN: 9781450315418, https://doi.org/10.1145/2430502.2430513
2. Blair G, Bencomo N, France RB (2009) Models@ run. time. In: Computer 42.10, pp 22–27
3. Dehury Chinmaya (2019) Data pipeline orchestration I. RADON publications, Tech. rep. RADON consortium
4. Dehury C et al (2020) Data pipeline architecture for serverless platform. In: European conference on software architecture. Springer, pp 241–246

5. Kumara I et al (2022) FOCloud: feature model guided performance prediction and explanation for deployment configurable cloud applications. In: IEEE Transactions on Services Computing, pp 1–1. https://doi.org/10.1109/TSC.2022.3142853

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.