J Grid Computing (2019) 17:813–829
https://doi.org/10.1007/s10723-019-09488-w

Supporting Programmable Autoscaling Rules for Containers and Virtual Machines on Clouds

József Kovács

Received: 19 November 2018 / Accepted: 14 August 2019 / Published online: 30 August 2019

Abstract With the increasing utilization of cloud computing and container technologies, orchestration is becoming an important area on both cloud and container levels. Beyond resource allocation, deployment and configuration, scaling is a key functionality in orchestration in terms of policy, description and flexibility. This paper presents an approach where the aim is to provide a high degree of flexibility in terms of available monitoring metrics and in terms of the definition of elasticity rules to implement practically any possible business logic for a given application. The aim is to provide a general interface for supporting programmable scaling policies utilizing monitoring metrics originating from infrastructure, application or any external components. The paper introduces a component, called Policy Keeper performing the auto-scaling based on user-defined rules, details how this component is operating in the auto-scaling framework, called MiCADO and demonstrates a deadline-based scaling use case.

Keywords Cloud · Virtual machine · Container · Docker · Autoscaling · Distributed monitoring

J. Kovács (✉)
Institute for Computer Science and Control, Hungarian Academy of Sciences, Kende u. 13-17, Budapest 1111, Hungary
e-mail: jozsef.kovacs@sztaki.mta.hu

1 Introduction

Nowadays, cloud [1] computing tends to be the de-facto standard for building flexible, easily maintainable, scalable infrastructure. The usage of commercial and private clouds [2] however requires more and more intelligent orchestration technologies to utilize the elasticity [3] of the clouds and to support the requirements of the applications. As the number of ported applications is growing, orchestration technologies are facing with new challenges. One of the key challenges of the orchestration technologies and tools is how to minimize the usage of resources while satisfying the capacity requirements of the application. On commercial clouds this feature may save costs for the user while on private clouds this feature may save costs for the operators of the cloud.

The most important functionalities supported by the orchestration technologies are resource allocation/ deallocation, application deployment/ undeployment, configuration/ reconfiguration, monitoring, failure detection/ handling/ healing and resizing/ scaling. In order to save costs, orchestration must focus on efficient resource allocation and scaling. Since for each application efficiency can be reached in a different way, it is hard to implement them with the same scaling mechanism. Therefore, an approach is needed where the goal is to provide maximum flexibility in performing decision on scaling. The motivation behind flexible decision making is coming from the users and operators of private and commercial clouds our lab is in connection with.

A nation level community cloud in Hungary is the MTA Cloud [4] that is financed by the Hungarian Academy of Sciences in order to serve all the scientists of the academy. This cloud is built as an open alliance of cloud sites. Currently two sites are set up by two academic institutes (MTA SZTAKI and MTA Wigner Data Center) but the open framework enables other institutes to join with their own clouds. The only expectation is that the joining partner must use Openstack as the IaaS [2] cloud software stack. The motivation behind the selection was to use an on-premise cloud software with strong ecosystem. Having the same cloud software on each site increases compatibility, integrity and makes migration much easier for the users. Since the number of scientific projects, users of MTA Cloud [5] and the quantity of resource demand are increasing much faster than the capacity of the cloud, a kind of overcommitment is applied on resource allocation i.e. the aggregated quota of the users is greater than the capacity of the cloud. Since most of the users are not familiar with orchestration, it is more and more important for the operations team to increase the efficiency of resource utilization by providing and disseminating automatic scaling solutions for the cloud users.

Scalability is the central issue to explore in the EU H2020 COLA [6] project where more than 20 different industrial applications are targeted to adapt for various cloud systems in a highly scalable way. The clouds and interfaces selected to be supported in the project include EC2 (for AWS and Opennebula), Nova (for Openstack), Cloudsigma, Azure and CloudBroker. The industrial applications ported to these clouds are significantly different in terms of their nature (web services, job execution), their requirements (memory, cpu or network load), and their technologies (container and/or virtual machine). The aim in this project is to design an orchestration tool to provide scaling solution for a wide variety of requirements.

To summarize the motivation described in this paper, we need a scaling solution which is able to orchestrate both virtual machines and containers while the two level scaling can be independent or cooperative. Even more virtual machine level scaling or container scaling may be utilized alone since virtual machines represent resources while containers represent applications in this approach. The motivation behind scaling at both levels is to perform application-level (container) scaling together with automating resource (virtual machine) allocation which perfectly fits to the requirements of the EU H2020 COLA project.

Due to the wide variety of aspects there should be no limitation regarding the metrics forming the base of a scaling decision. Practically, we need the possibility to form a scaling logic based on the value of any monitoring metrics regardless its origin. The monitoring metrics forming the inputs of the scaling decision must be able to arrive from any location e.g. from external source not belonging to the resources and infrastructure executing the application. Please, note that most of the scaling solutions have this limitation. Finally, the scaling mechanism should support more general expression based scaling than the currently wide-spreaded trigger and threshold based scaling mechanisms.

This paper is organized as follows. The next section overviews several related works which show similarities to the developments described in this paper. Section 3 presents the concept and design principles aims to introduce a scaling solution that is general in terms of scaling logic specification and in terms of scaling metrics. The specification of scaling logic is described in Section 4 while section 5 details how the scaling policies are realized in the component called Policy Keeper. Then the integration of the Policy Keeper into MiCADO (cloud and container orchestrator in COLA) is shown in Section 6 while its supported scaling scenarios are summarized in Section 7. Finally, an example scaling policy dealing with job deadline is presented in Section 8 before the conclusion outlined in Section 9.

2 Related Work

There are numerous solutions for scaling either on cloud or on container levels, however it is hard to mention tools which provides scaling functionality on both levels in a combined way.

One of the most used de-facto standard scaling service is the one provided by AWS Auto Scaling [7]. In this environment triggers determine how to act over an application for which CPU utilization, network usage or disk operation related metrics are beyond a predefined threshold. There are a few predefined scaling policies (e.g. “target tracking”, “step”, “simple”,” scheduled”,…) to ease the utilization of the scaling functionality, however new application business strategies beyond the provided policies which do not fit into the trigger based scaling is not supported.

Another work worth mentioning here is RightScale [8] which is realized as a broker between cloud providers and users by providing unified interfaces. Its autoscaling solution is based on triggers and thresholds. They support many different and popular scaling related metrics like for example Mysql active connections and http server requests. However, these scaling indicators may not be able to support all types of application business strategy.

The work described in [9] realizes a programmable framework, where the scaling logic can be implemented inside the application. The predefined primitives for scaling are implemented as a programming library. The elasticity controller becomes part of the application which provides high-flexibility for implementing business logic for scaling for the sake of (re)programming the application code.

One of the most widespread environment providing scaling at virtual machine and at container level is Kubernetes [10] which was originally a Google product and now is hosted by Cloud Native Computing Foundation [11]. The Kubernetes horizontal pod autoscaler supports reactive threshold-based rules for CPU utilization metric and with its plugins supported metrics can be extended. The user specifies thresholds for the values of metrics and actions to be taken in order to change the replica number. For virtual machine level scaling Kubernetes provides Cluster scaling. Fixed algorithm is provided where parameters may slightly influence the basic operation. Whenever pods are suffering from insufficient resources, further worker nodes are instantiated. Supported clouds are GCE, AWS and Azure.

Cloudify [12] is an open-source model-driven cloud native orchestration platform. It operates based on an application description following the TOSCA [13] de-facto modeling language. The user describes the resources, applications, services and their deployment together with their linked scaling rules. Scaling is realized by a built-in workflow (like all other application-related operations install/start/stop/heal etc. are implemented). For scaling, upper and lower thresholds are introduced which can be parameterized (metrics, plugins for monitoring) and actions (to increase/decrease the number of instances) can be associated to be taken upon reached thresholds. The most outstanding features of Cloudify are the high-level description language and cloud independence. Scaling is definitely supported, however with a simple trigger-threshold mechanism.

The approach which is hard to find among the related works is to provide a general interface for implementing flexible scaling decisions where scaling can be realized in a combined way for virtual machines and containers.

3 Concept and Design Principles

Policy Keeper is intended to perform a decision on scaling by calculating the optimal number of instances needed to be created. The scaling decision must be made on two levels:

1. **Virtual machine level**: Scaling up or down the number of virtual machines (on which Docker Swarm [14] cluster is realized and the container application is running) realizes adding or removing resources from the cluster. Whenever virtual machine level upscaling happens, a new Docker node is attached and the Docker cluster grows. Downscaling at virtual machine level means removing Docker nodes from the Docker Swarm cluster i.e. the Docker cluster loses resource.
2. **Container level**: In Docker Swarm a (micro)service is realized by containers running on the nodes of Docker Swarm in a distributed way. In order to add more resources to a particular microservice, the number of instances of the containers (realizing the Docker Service) must be increased. Docker Swarm makes sure the containers are executed in parallel on the nodes of the cluster and the user requests arriving to the service is distributed among the containers for processing. Scaling up and down the number of containers of a given service increases the parallelism of the request handling at containers level i.e. increases the resources associated to the given service.

To implement scaling, there are control loops realized on virtual machine and container levels. Control loops are depicted in Fig. 1.

The virtual machines (i.e. nodes) are represented by boxes entitled Node1 and Node2. First, information is collected on the nodes by the monitoring

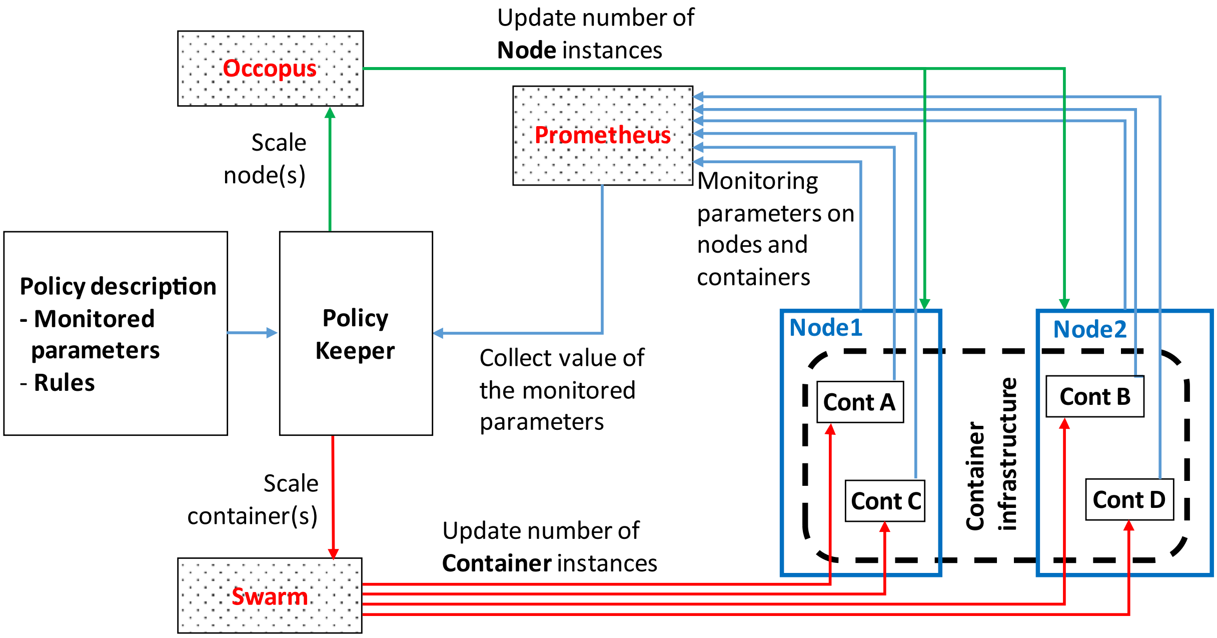


Fig. 1 Control loops to scale virtual machines and containers

system realized by Prometheus [15]. As a decision maker service, the Policy Keeper component holds the list of monitored parameters (extracted from Prometheus) and the scaling rules describing the decision on scaling. Once a decision is made, a lightweight cloud orchestration tool realized by Occopus [16] performs the scaling of the nodes, i.e. launches or destroys virtual machines and attaches them as workers to the Docker Swarm cluster. This mechanism realizes the virtual machine level control loop.

The containers labelled by Cont A, B, C, D are forming a container infrastructure and realizing services for the users. Various types of parameters are monitored and collected by Prometheus which can be used for decision making. Policy Keeper holds the scaling rules for each service and performs the decision in function of the value of the incoming parameters. The decision of container scaling is finally realized by Docker Swarm.

The Policy Keeper decisions are based on the inputs provided by the monitoring system realized by Prometheus, while scaling is implemented by Occopus (on cloud level) and Swarm (on container level). To complete the control loop either at the level of nodes or at the level of containers, design decision must be made on the following:

- what are the parameters to be monitored on the observed object (node or service);
- what is the scaling rule which provides a decision based on the actual value of the monitoring parameters.

3.1 Major Design Principles

To avoid limiting the monitoring parameters, the first design principle is to let the monitoring parameters be defined dynamically for each submitted application. This must be true for parameters not supported by the current monitoring setup. *The key principle is to make the monitoring system dynamically extendable in terms of data sources and monitoring parameters*.

To make the scaling rule (which defines the decision making algorithm in Policy Keeper in terms of scaling) as flexible as possible, the scaling rule is considered as an input. It is a more flexible solution than predefining a fixed algorithm or providing selection possibility from a predefined list of scaling rules. *The key design principle is to make the scaling rule specifiable by the user in a flexible way.*

3.2 Monitoring Parameters

Most typical scaling rules require performance-related parameters of the executing nodes at virtual machine level. Monitoring the cpu-, memory- and network load of the nodes provides most of the parameters for a typical scaling rule. However, there are situations where an application (realized by a container infrastructure) requires scaling based on parameters that are not on the list of predefined monitored parameters. For example, the application may require significant disk capacity on the node to cache some data, or may require other types of resources inside the virtual machine that are not among the list of predefined monitored parameters to ensure proper scaling.

The main goal of Policy Keeper is to provide maximum flexibility in terms of monitored parameters. To do that, Policy Keeper aims to support dynamically configured list of monitored parameters instead of a selection from already configured parameters. To do that, user is allowed to dynamically specify new parameters that will be monitored, even if the current monitoring setup is not able to gather the value of these new parameters. This dynamic extension of the monitoring system is supported by Prometheus through its query language, query API and dynamically configurable exporters [17] realizing the data extraction.

Scaling rules on the level of application may require the handling of more complex scenarios. These scenarios may rely on monitoring parameters which are not predefined and provided by the default built-in monitoring system. Moreover, if an application scaling rule requires some information which exists inside the application's internal state, a special data collection component is required to be attached to the monitoring system as data source.

Dynamic extension of exporters provides further advantages in this autoscaling mechanism. The most widespread autoscaling systems do not focus on collecting monitoring information from external systems which are not part of the application or of the resources and of the infrastructure associated to it. For example, when autoscaling depends on a parameter that is held by a remote server, designing a scaling logic is not so trivial without the support for monitoring external components. Just think of processing emails coming from an external email server where the speed of processing may depend on the number of emails waiting in the queue at the moment. In this situation the email server is not part of the infrastructure executing the application, but we need to monitor it and feed the value to the scaling logic.

3.3 Scaling Rules

Scaling rules are intended to calculate the required number of replicas of containers for a certain service and/or the required number of instances of virtual machines. A scaling rule should express the direction (up/down) and quantity (instance number) of scaling. A scaling rule may be reutilized by different applications provided that the application characteristics are similar and the business policy needed by the operator/user of Policy Keeper is similar. A complex scaling rule has the task of coordinating the resource capacity available for an application (virtual machine level scaling) and the resource usage by the application (container level scaling). For both, the aim of the Policy Keeper is to provide maximum flexibility, configurability. The complexity of the scaling rules, and the variety of user requirements may easily result in insufficient support from scaling rules in case Policy Keeper tries to provide a predefined set of scaling rules.

Using predefined scaling rules may perfectly support some groups of applications. However, the variety of requirements will always result in more complex rules to be implemented. To support scaling rules and policies for diversity of applications and requirements, Policy Keeper supports scaling rules to be defined as user inputs. Handling the scaling rules as inputs provides maximum flexibility for the user and removes limitations in relation to the supported types of applications and scaling logic.

The scaling rule for the Policy Keeper must be an expression that can be automatically evaluated with the monitoring parameters as input, and the output of the evaluation is the decision on scaling i.e. the number of instances. To give the user as much freedom as possible, the scaling rule should be able to formalize arithmetic, logic and control expressions.

In Policy Keeper, the scaling policy contains the list of monitoring parameters together with their definition and the scaling rules referring to the parameters. The policy is described in YAML [18] and the selected language for expressing the scaling rule is Python. The simplicity of the language and the easy evaluation resulted in introducing the support of Python language in the scaling rule definition.

3.4 Modularity of the Environment

Docker Swarm, Occopus and Prometheus have been mentioned during the introduction of the concept as selected tools for realizing the container execution framework, the cloud orchestrator and monitoring services. One of the advantages of this concept is the support for modularity approach, since any of the mentioned tools can be replaced. For example, Occopus can be replaced by any other cloud orchestrator tools like Terraform [19], while Kubernetes can be an alternative for Docker Swarm. The necessary features for cloud and container orchestrators are deployment and scaling. In case of monitoring systems Policy Keeper requires an API towards which expressions can be sent for evaluating monitoring metrics and the possibility to dynamically add new data sources to the monitoring network.

4 Policy Definition

The Policy Keeper component takes a policy description as input for handling the monitoring sources (Prometheus exporters), the monitoring queries (Prometheus expressions), the monitoring alerts (Prometheus alerts) and the scaling rules (decision making in Python). The policy description is structured to address sections for each of these topics. Policy description uses YAML syntax and has the following structure:

```
stack: <name of docker stack>
data:
  sources:
    - '<ip>:<port>'
  constants:
    <name of constant>: '<value of the constant>'
  queries:
    <name of parameter>: '<prometheus query expression>'
  alerts:
    - alert: <name of alert>
      expr: '<prometheus logical expression>'
      for: <time period: 1s, 1m, 1h, etc.>
scaling:
  nodes:
    min: <minimum number of nodes>
    max: <maximum number of nodes>
    target: |
      <Python code to realize scaling rule>
  services:
    - name: "<name of docker service to scale"
      min: <minimum number of containers>
      max: <maximum number of containers>
      target: |
      <Python code to realize scaling rule>
```

The variable called ‘stack’ is required to identify the Docker stack to be manipulated through Docker Swarm. Under the section named‘data’, all Prometheus query and alert related settings can be specified. The section called ‘scaling’ contains the scaling related specification, both for ‘nodes’ i.e. to scale at virtual machine level and for Docker ‘services’ i.e. to scale at container level. A more detailed description of the policy, will be provided in the next sections.

4.1 Data Sources

Dynamic attachment of an external exporter can be requested under the ‘source’ subsection by adding a list item with the ip address and port number of the exporter. The following YAML structure shows an example:

```
data:
  sources:
    - '192.168.154.116:8090'
      - 'rabbitmq_exporter:8090'
      - 'myexporter.mydomain.com:6000'
```

Each item found under the ‘data’/‘sources’ subsection is configured under Prometheus which starts collecting information provided/exported by the exporters. Once done, the values of the parameters provided by the exporters become available as input for a query expression.

4.2 Metrics

To utilize one of the exporters i.e. to query the value of a metric collected by the newly configured exporter, a Prometheus query expression must be defined. Prometheus queries must be listed under the ‘queries’ subsection under the ‘data’ section of scalability policy. An example is shown below:

```
data:
 queries:
   REMAININGTIME: '{{DEADLINE}}-time()'
   ITEMS: 'rabbitmq_queue_messages_persistent
                              {queue="machinery_tasks"}'
```

In this example, two variables called ‘REMAININGTIME’ and ‘ITEMS’ have been defined with their corresponding Prometheus query expressions. Each time the Policy Keeper instructs Prometheus to evaluate the queries, the returned value is associated to the variable name and can be referred in the scaling rule.

4.3 Constants

As it can be seen in the previous example, for ‘REMAININGTIME’ variable, a predefined constant has been referred. Each referred constant, specified under the‘constants’, subsection is replaced by its associated value. The following YAML structure shows an example:

```
data:
  constants:
    DEADLINE: 1529499571
```

Referring a constant, Jinja2 [20] type syntax (i.e. using double brackets around the name of the constant) must be used. Here is an example to refer to the value of a constant:

```
{{DEADLINE}}
```

4.4 Alerts

Prometheus supports alerting mechanism. Alerts can be considered as notifications over events which are important in relation to scaling. For example, an event which describes that a certain service became overloaded can be configured in order to trigger an up-scaling procedure reducing the load on the actual containers.

To utilize the alerting system of Prometheus, alerts can be defined in the Policy Keeper scaling policy description under the ‘alerts’ subsection of ‘data’ with a list of dictionary of three pieces of key-value (‘alert’, ‘expr’, ‘for’) pairs. The following YAML structure shows an example (together with constants to make it clear) where an alert is configured to fire whenever the average cpu usage for all the containers belonging to the given service is above a certain threshold for at least 30 s.

```
data:
  constants:
    SERVICE_NAME: 'stressng'
    SERVICE_FULL_NAME: '{{stack}}_stressng'
    SERVICE_TH_MAX: '60'
    SERVICE_TH_MIN: '20'
  alerts:
  - alert: service_overloaded
    expr: 'avg(rate(container_cpu_usage_seconds_total
                {container_label_com_docker_swarm_service_name="{{SER
                VICE_FULL_NAME}}"}[30s]))*100 > {{SERVICE_TH_MAX}}'
    for: 30s
```

A named alert (‘alert’) is a logical expression (‘expr’) which is evaluated by Prometheus and the alert is fired when the expression remains “True” for a period of time defined by the third key (‘for’).

In case of alert triggering event the value “true” will be associated to the variable defined in section “alert”. Similarly to the queries expression, the alert definition may also refer to constants included in {{}} brackets. To check if an alert is firing, the scaling rule simply refers to the name of the alert as a Boolean variable. The following YAML code shows an example:

```
scaling:
  ...
  services:
   - name: ‘stressng’
     ...
       target: |
       if service_overloaded:
            m_container_count+=1
```

4.5 Scaling Rules

A scaling rule in the policy description expresses the decision on scaling i.e. it is realized by a code snippet. A scaling rule must be defined for nodes (i.e. to scale at virtual machine level) and for services (i.e. to scale at container level). The following YAML code shows the structure of the scaling section inside the policy description:

```
scaling:
  nodes:
    min: <minimum number of nodes>
    max: <maximum number of nodes>
    target: |
      <Python code to realize scaling rule>
  services:
    - name: "<name of docker service to scale"
    min: <minimum number of containers>
    max: <maximum number of containers>
    target: |
          <Python code to realize scaling by updating the number of
          instances of the target nodes>
```

Policy Keeper supports the specification of the scaling rule by a Python expression under the ‘target’ keyword. The Python expression must be formalized with the following conditions:

- Each constant defined under the ‘constants’ section can be referred; its value is the one defined by the user
- Each variable defined under the ‘queries’ section can be referred; its value is the result returned by Prometheus in response to the query string
- Each alert name defined under the ‘alerts’ section can be referred, its value is a logical ‘True’ in case the alert is firing, ‘False’ otherwise
- Expression must follow the syntax of the Python language
- Expression can be multiline
- The following predefined variables can be referred; their values are defined and updated by Policy Keeper:

  … m_nodes: Python list of nodes belonging to the Docker Swarm cluster.

… m_node_count: the target number of nodes.

… m_container_count: the target number of containers for the service the evaluation belongs to.

…m_time_since_node_count_changed: time in seconds elapsed since the number of nodes has changed

- In node level scaling rule, the name of the variable to be set is 'm_node_count'; as an effect the number stored in this variable will be set as target instance number for the virtual machines.
- In container level scaling rule, the name of the variable to be set is 'm_container_count'; as an effect the number stored in this variable will be set as target instance number for the given container service.

The next example shows a YAML embedded Python code to scale up and down based on the events 'service_overloaded' and 'service_underloaded' can be done simply as shown in the next YAML code:

```
scaling:
  services:
 - name: myservice
   min: 1
   max: 5
   target: |
     if service_overloaded:
        m_container_count+=1
     if service_underloaded:
        m_container_count-=1
```

The scaling rule (specified under the 'target' keyword) is evaluated periodically. Before each evaluation, the values of the variables and alerts are updated based on Prometheus queries. The Python expression is expected to update the necessary variables ('m_container_count' in this case) to express the need for scaling.

When the expression is evaluated and the target number of containers or nodes are calculated, each calculated values are limited between 'min' and 'max' values defined for the particular node or container. As a consequence, Policy Keeper always keeps the target number between the minimum and maximum regardless of the value returned by the scaling expression.

5 Internal Operation

Policy Keeper implements the scalability decision service by monitoring, evaluating and instructing. The policy keeping functionality has been developed from scratch in Python using Flask [21] for implementing the service endpoint.

The Flask based Python code is running in a container and communicates to Prometheus (for evaluating queries and alerts), to Occopus to realize node scaling and to Docker to realize Docker service scaling.

The user-defined input for Policy Keeper is a YAML-based description specified in Section 4. The internal operation of Policy Keeper is illustrated in Fig. 2. The following paragraphs details the internal operation.

The operation of Policy Keeper starts with the invocation of the start method of its REST API (Step1 in Fig. 2). The parameter is a scaling policy description in which Policy Keeper first resolves the text where references are used. At this step Jinja2 is used to resolve variable references.

The next phase (Step2 in Fig. 2) configures Prometheus. Configuration involves the registration of the user-defined exporters through the configuration file of Prometheus. The Prometheus configuration file written in YAML contains a section called "scrape_config" specifying a set of monitoring data exporters (called targets) from which data should be scraped from. For each scrape target, a job name ("job_name" attribute) must be specified. Policy Keeper registers with its own name:

```
scrape_configs:
- job_name: 'policykeeper'
  scrape_interval: …
  static_configs:
    - targets: ['<user defined exporter ip:port>']
```

There are several parameters (e.g. "scrape_interval") which are optional and can be fine-tuned in each scrape job definition. The subsection named "static_configs" contains the "targets" keyword, which is a list of endpoints for the Prometheus exporters. To configure Prometheus to scrape i.e. to collect metric information from the exporters, endpoint values must be inserted into the list specified by the "targets" keyword.

In case the source is an external exporter (not running under Docker Swarm autoscaled by Policy Keeper) the configuration finishes with the insertion of the endpoints. However, for internal

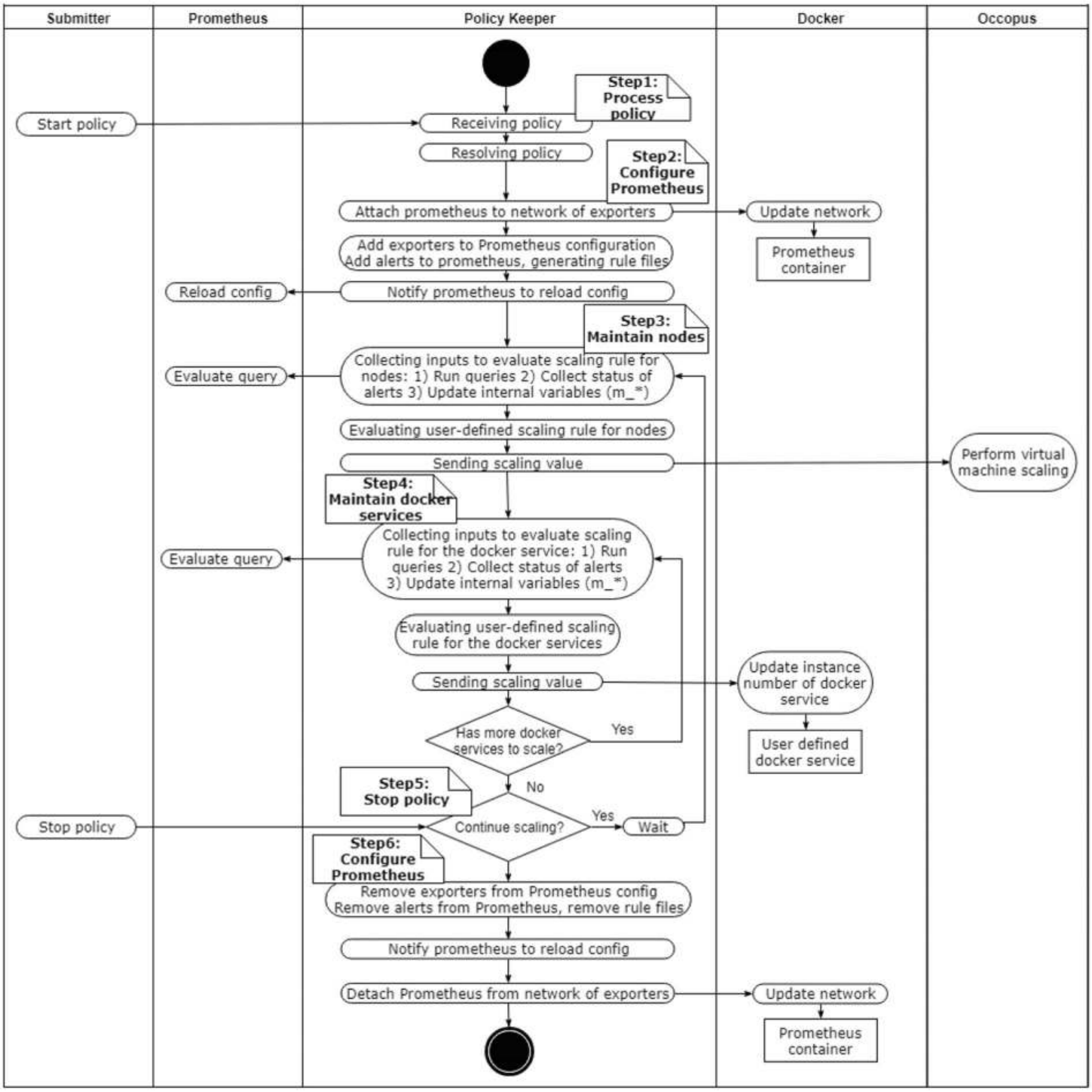

Submitter
Prometheus
Policy Keeper
Docker
Occopus
Step1: Process policy
Start policy
Receiving policy
Resolving policy
Step2: Configure Prometheus
Attach prometheus to network of exporters
Update network
Add exporters to Prometheus configuration
Add alerts to prometheus, generating rule files
Prometheus container
Reload config
Notify prometheus to reload config
Step3: Maintain nodes
Evaluate query
Collecting inputs to evaluate scaling rule for nodes: 1) Run queries 2) Collect status of alerts 3) Update internal variables (m_*)
Evaluating user-defined scaling rule for nodes
Sending scaling value
Perform virtual machine scaling
Step4: Maintain docker services
Evaluate query
Collecting inputs to evaluate scaling rule for the docker service: 1) Run queries 2) Collect status of alerts 3) Update internal variables (m_*)
Evaluating user-defined scaling rule for the docker services
Sending scaling value
Update instance number of docker service
User defined docker service
Has more docker services to scale?
Yes
No
Step5: Stop policy
Stop policy
Continue scaling?
Yes
Wait
Step6: Configure Prometheus
Remove exporters from Prometheus config
Remove alerts from Prometheus, remove rule files
Notify prometheus to reload config
Detach Prometheus from network of exporters
Update network
Prometheus container

Fig. 2 Internal high-level operation of Policy Keeper

exporters, Policy Keeper instructs Docker to let the Prometheus service attach to the (Docker) network of the exporter service, otherwise Prometheus would not reach the internal exporter. The next step, is the generation of the rule files based on the alert definition specified in the policy file. At the end of this phase, Prometheus is notified to reload its configuration i.e. to activate the changes.

At this point, all preparation has been done, the periodic maintenance (evaluation and scaling) cycle can start. Each cycle starts with the node maintenance followed by the container maintenance. Predefined time separates consecutive cycles.

Node maintenance (Step3 in Fig. 2) starts with collecting all the inputs necessary to evaluate a scaling rule (specified by the policy). The first step is to evaluate the variables defined in the “queries” section of the policy. For each item a Prometheus query expression is defined which is sent to Prometheus for evaluation. When all variables are evaluated the calculation continues with collecting the state of alerts if any has been specified.

When an alert is fired the Policy Keeper is notified by Prometheus through a special extension of Prometheus called Alert Manager. These notifications are registered inside the Policy Keeper and evaluated later when the status of alerts are referred by the scaling rules. For this purpose, a Boolean variable will be generated and associated for each alert (see Section 4.4).

The final step in collecting inputs for the evaluation is the update of values of the built-in variables (specified in Section 4.5).

Evaluation of the scaling rule for the node means the execution of the Python code specified in the scaling policy specified in Section 4.5. The evaluation is done by a separate module in Policy Keeper. The result of the evaluation is the required number of instances to scale the nodes to. The final step is to notify Occopus about the scaling decision and set the number of Docker worker nodes accordingly.

The next stage of the operation is the maintenance of the Docker Services (Step4 in Fig. 2). The procedure described in details through the following paragraphs is performed for each individual Docker Service specified in the scaling policy. Practically, it has the same pattern as a for the node maintenance, since the same steps are performed for the Docker Service.

Scaling a Docker Service starts with the collection of respecting inputs to evaluate the scaling rule including the evaluation of the queries by Prometheus, reading the status of the alerts and updating the internal variables. Following the pattern drawn by the node maintenance the scaling rule for the Docker Service will be evaluated and the outcome of the evaluation may instruct the Docker Swarm to scale the Docker Service to the calculated number of replicas.

When the Policy Keeper is instructed to stop the maintenance (Step5 in Fig. 2) maintenance loop (Step3 and Step4 in Fig. 2) terminates. As a consequence, Policy Keeper rolls back all the changes made in Step2, i.e. removes changes from the configuration file of Prometheus, detaches Prometheus from any network it has been attached to, removes rule files containing the alerts and notifies Prometheus to reinitialize its (original) configuration. Finally, Policy Keeper becomes inactive and waits for further instructions through its REST API.

6 Integration with MiCADO

The overall architecture of MiCADO [22, 23] has been initially designed by the COLA EU project [6]. The main components are Prometheus for monitoring, Docker Swarm for container orchestration, Occopus for virtual machine orchestration, Submitter to handle TOSCA-based descriptions and finally the Policy Keeper to perform decision on scaling. This section focuses on the implementation of Policy Keeper and the surrounding components connected to it. A detailed architecture of the Policy Keeper and its environment can be seen in Fig. 3.

MiCADO integrates Prometheus as a monitoring tool on the master node and has two exporters running on each worker node to collect information on the node and on the containers running on a given node. With the support of these two built-in exporters (node exporter [24] and cadvisor [25]) a long list of parameters (metrics) can be monitored and queried from Prometheus.

To monitor a parameter that is not supported either by the node exporter or by the cAdvisor, Policy Keeper provides a mechanism to attach (i.e. register) new user-defined exporters dynamically. By defining the location of a new exporter, Policy Keeper can configure Prometheus to collect metrics from a user-defined exporter. The new exporter can be either executed by MiCADO (internal) or can be executed outside and independently from MiCADO (external). Deployment of an exporter is not performed by the Policy Keeper. Once an exporter is attached, its metrics become available in Prometheus. Policy Keeper uses the query interface of Prometheus to collect the values of the parameters originating from the exporters.

Policy Keeper also supports alerting with the help of Prometheus. When the scaling policy contains definition of alerts, they are registered in Prometheus to be maintained. Alert manager is a component part of the Prometheus software package and handles alerts sent by Prometheus. Alert manager organizes the alerts and notifies Policy Keeper through its REST interface when an alert fires. Upon scaling decision Docker Swarm and Occopus realizes creation or removal of instances if necessary.

The overall flow of operation focusing on a performed scaling event is as follows (see Fig. 3):

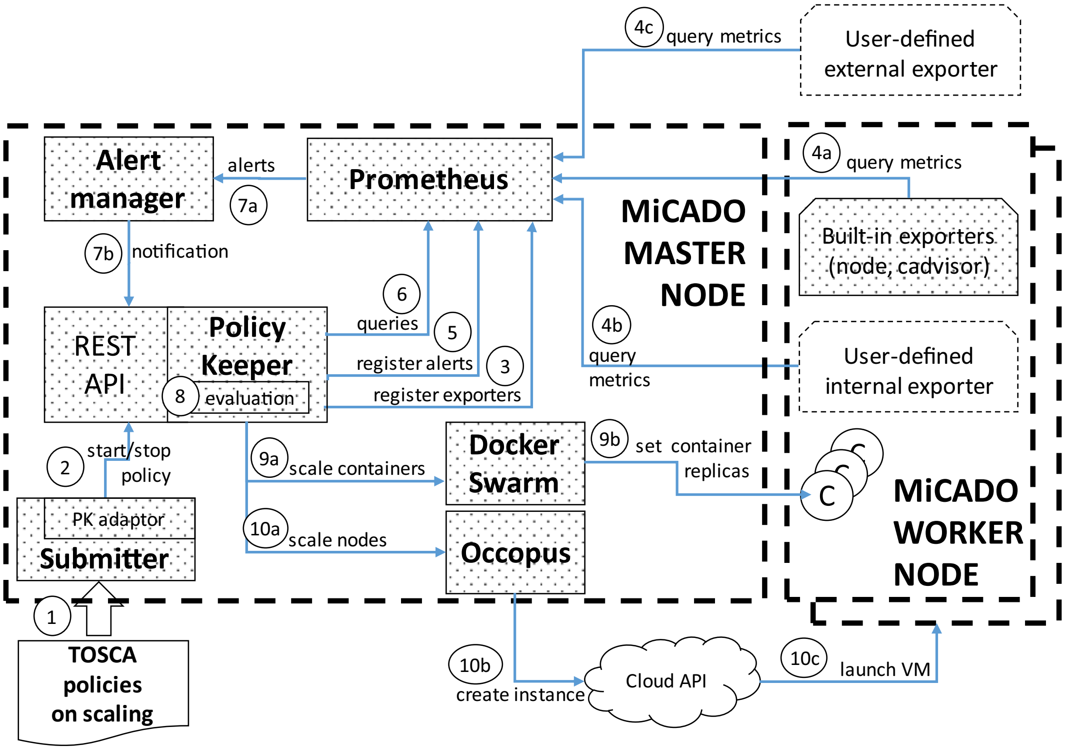


Fig. 3 MiCADO architecture with the integrated Policy Keeper

- The submitter receives a (TOSCA-based) description of the scaling policy as part of the overall (TOSCA) description of the container infrastructure (Step 1). Please, note that TOSCA policies and the submitter component is out of scope of this paper.
- The submitter uses its Policy Keeper adaptor to convert the TOSCA based scaling policy format to the native policy format (see Section 4) of the Policy Keeper. After the conversion, the policy is sent to the Policy Keeper through its REST interface for elaboration (Step 2).
- In the next step, the Policy Keeper registers the exporters – specified in the policy – with Prometheus (Step 3).
- Prometheus immediately starts pulling the metrics data from the exporters regardless they are built-in (Step 4a), user-defined internal (Step 4b) or user-defined external (Step 4c).
- In case the policy contains definition of alerts, Policy Keeper registers them with Prometheus as well (Step 5).
- At this point, Prometheus is ready to deliver metric values from its exporters. Policy Keeper periodically issues queries towards Prometheus to update the value of the variables (referred by the scaling rule) (Step 6).
- Whenever an alert is firing, Prometheus notifies Policy Keeper through Alert manager (Step 7a) which registers the event (Step 7b).
- Policy Keeper periodically reevaluates the scaling rules (Step 8) which contain references to query or alert based variables.
- As a result of the reevaluation of the scaling rules, Policy Keeper may instruct Docker Swarm (Step 9a) to scale up/down a given container (Step 9b).

- The evaluation of the scaling rule may also result in scaling at virtual machine level. In this case, Policy Keeper instructs Occopus (Step 10a) for scaling which in turn asks the target cloud API to create/destroy instances (Step 10b). Finally – in case of "create" – a new VM is launched (Step 10c) on which a new worker node is built up and attached to the master.

This step-by-step operation of Policy Keeper and its environment ensures the realization of two control loops, one on virtual machine and the other container levels.

7 Supported Scenarios

The solution detailed in the previous sections provides a kind of framework where various scenarios can be supported based on the scaling rules and monitoring extensions. The following list gives some hints on a selection of interesting scenarios which can be implemented in the framework.

- **Threshold based scaling.** This scenario is very easy to be implemented, since CPU / Memory/ Disk/ Network consumption of the containers and capacity of the virtual machines are available based on the current built-in Prometheus exporters. The decision algorithm can be designed based on periodic checking of the actual values or on specifying alerts which fire when threshold is reached.
- **Scaling driven by an external component.** In case the scaling decision is made by an external component, the actual values can easily be propagated to Policy Keeper through the Prometheus monitoring system using proper exporter components. To avoid writing of a new exporter for this purpose, the easiest way is to store (and update) the scaling values in a database and make it visible for the Policy Keeper using for example an sql exporter attached to Prometheus. The algorithm submitted can refer to these values when deciding on scaling.
- **Cloud-only (container-free) scaling.** Policy Keeper is designed for scaling virtual machines and containers simultaneously. However, the architecture has been designed in a way that the two scaling level may operate independently. No operational problem occurs in case the container scaling algorithm is missing. To use Policy Keeper for a node-only scaling scenario, container related settings must be simply omitted.
- **User driven manual-scaling.** This scenario can be useful when system admins want to decide on scaling by themselves. The easiest way of implementing this scenario in Policy Keeper is to store the decision (of system admins) in a database similarly to the 'Scaling driven by an external component' scenario.
- **Scaling based on internal metrics of the application.** There are several options to gather information from the application internal state space. The key factor is to export the application state variables in a way that is compliant with Prometheus monitoring system. There are several options. An exporter can be written from scratch to provide the necessary monitoring info. In case the application is a web-service, its interface could provide exporter functionality or if the application has a database component, scaling related values can be stored and exported from the database.
- **Scheduled (time-based) scaling.** When the application requires up and downscaling at predefined times the Policy Keeper algorithm can be written in conform to that requirement. Handling the actual time and date is possible within the scaling algorithm of Policy Keeper.
- **Deadline based scaling.** In case the application performs for example job execution where the job is taken as an item from a queuing system scaling algorithm can be designed based on three parameters: deadline, average execution time and the actual length of the queue. The length of the queue can be continuously monitored, there are exporters for that purpose (e.g. rabbitmq exporter [26]). Average execution time can also be either a dynamically monitored parameter or a static input one. Finally, deadline must be provided by the user.

8 Use-Case Demonstration of Deadline-Based Policy

In order to demonstrate the flexibility of the Policy Keeper, a deadline-based policy is presented in this section. For the demonstration, a simple queuing tool called CQueue [27] is used. This tool is a container execution service consists of a master and any number

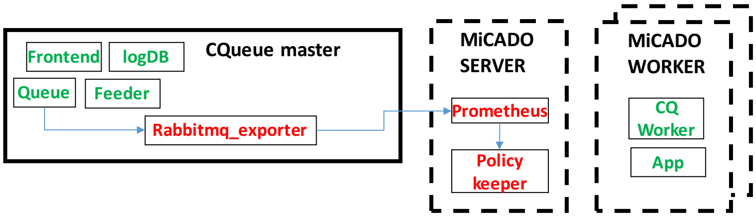
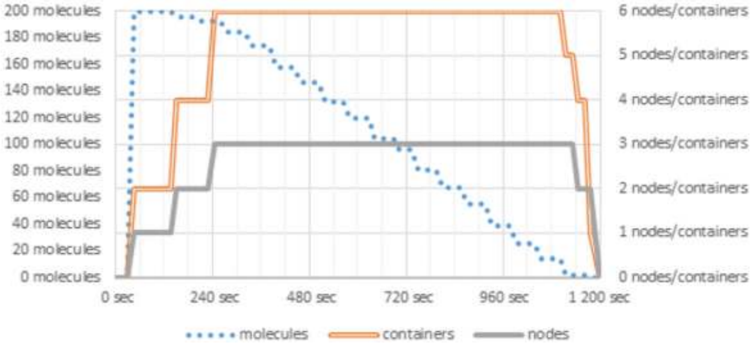


Fig. 4 Architecture for deadline-based policy with CQueue in MiCADO

Fig. 5 Number of jobs, nodes and containers in time during deadline-based execution



of worker components. Master implements a queue, where each item (called task in CQueue) represents the specification of a container execution (image, command, arguments, etc.). Each Worker component fetches the tasks one after the other and executes the container as specified by the task.

We have implemented a deadline based policy with this lightweight container queueing system. Please, note that any other container queuing tool would also fit in this demonstration. In this example, the container to be executed stores a popular molecular docking simulation application called Autodock Vina [28]. One container execution item (specified in the queue) represents one job execution.

Figure 4 shows the high-level architecture of the demonstrated scenario. On the left hand side, a separate VM executes the Master component of CQueue, while the Worker component of CQueue (realized in a container) represents the Docker service to be executed and scaled up/down by Policy Keeper in order to reach a predefined deadline. Scaling up and down the CQueue worker component increases/decreases the processing speed of tasks.

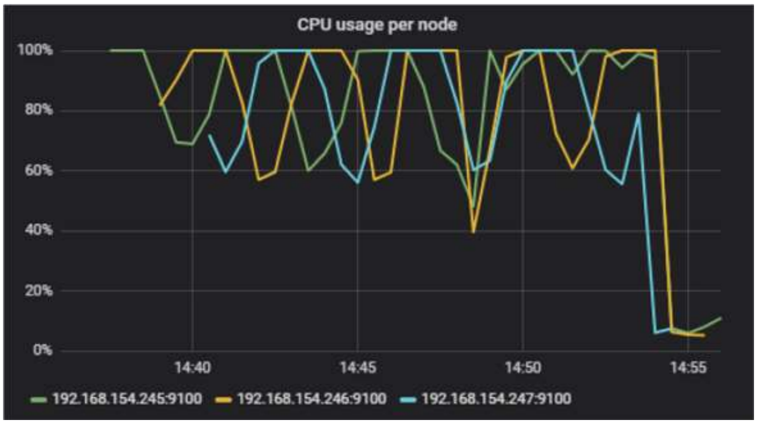
The processing speed to be set through scaling of CQueue workers relies on three main parameters: deadline (DL), actual number of items (ITEMS), and average execution time (AET) of the jobs running in the containers. The number of replicas to be executed can easily be calculated by the following expression:

`ceil(AET/((DL-ACTUALTIME)/ITEMS))`

Average execution time and deadline parameters are considered fixed in this demonstration. However, the number of items in the queue and the actual time are continuously monitored.

In order to monitor the number of items in the queue, a RabbitMQ exporter [26] has been deployed near to the Queue component of CQueue master. This exporter has the task of querying the number of remaining tasks in a queue. This exporter is part of the Prometheus exporter repository [17]. This repository contains many exporters covering thousands of monitorable parameters for many areas. In case when none of these exporters cover the parameter we need, it is possible to write our own exporter. Supporting dynamically attachable Prometheus exporters is an important feature of Policy Keeper utilized in this use case.

Fig. 6 CPU usage for each node during the experiment



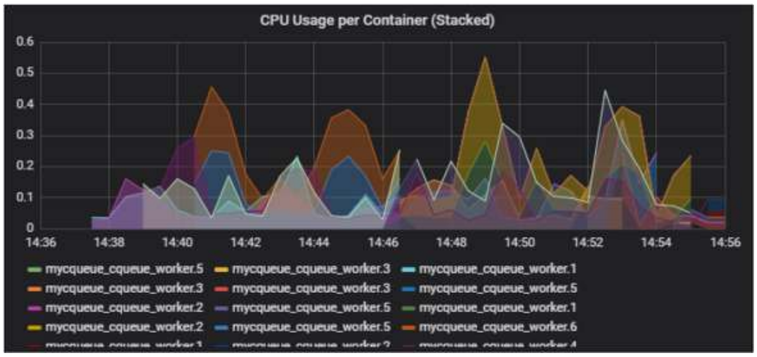


Fig. 7 CPU usage for each container during the experiment

The demonstration is executed as follows: 200 molecule docking simulation jobs have been submitted, with an average of 25 s execution time and with a 20 min (1200 s) maximum execution time for all the jobs from the time of submission. The policy has been implemented in a way that each node executes maximum two simulations (containers) in parallel.

During the experiment, following the initial calculation, MiCADO started to scale the worker nodes up to three. After approximately half a minute, the first VM appears and the job execution starts in two containers (see Fig. 5). The calculation predicted that six containers (running on three VMs) were necessary to meet the deadline so after about four minutes all the nodes and containers were exploited. The number of nodes and containers started to scale down after about 18 min, and after 20 min all simulation jobs have been finished.

For inspecting the resource usage Grafana (part of MiCADO Dashboard) was used. Grafana under MiCADO was configured to show the CPU, memory and network usage both for the virtual machines (left side) and for the containers (right side). Figure 6 shows the CPU load for the three nodes during the experiment, while Fig. 7 shows the same for the containers. In Figs. 8 and 9 we can inspect the memory usage for the virtual machines and for the containers in the same timeframe.

During the experiment, the MiCADO worker nodes were launched on the SZTAKI Opennebula cloud. The CQueue worker container was defined under the cqueue_worker section with all the environment variables necessary for CQueue worker to build up connection to the CQueue master.

The scaling policy (see Code 1) shows the main sections (sources, constants, queries) for both nodes and containers. Under sources, the RabbitMQ exporter location is defined. The constants section contains average execution time (AET) and deadline (DEADLINE) as most important parameters. Deadline is a unix timestamp in this example. The queries section specifies the time remaining (REMAININGTIME) and number of simulation job remaining (ITEMS) to be monitored. Finally, the scaling rules for nodes and containers specify how many instances need to be launched based on

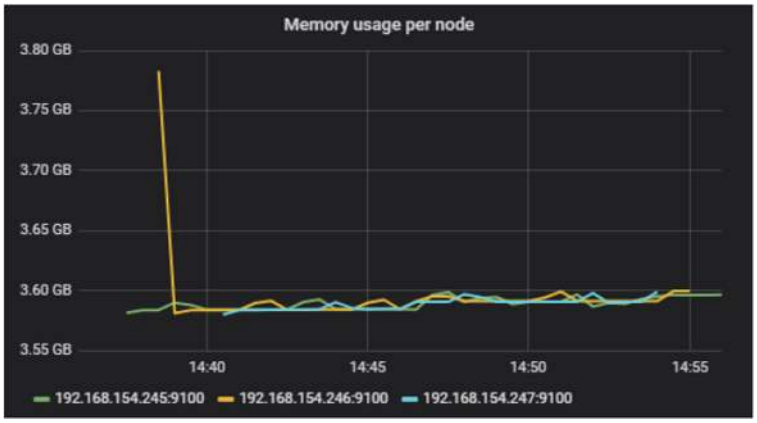


Fig. 8 Memory usage for each node during the experiment

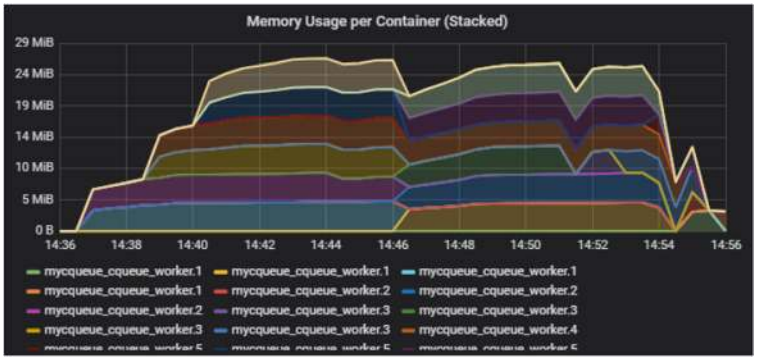


Fig. 9 Memory usage for each container during the experiment

the values of parameters specified under the constant and queries sections.

In this particular scaling policy, the main goal was to demonstrate that scaling logic can be defined as an incoming code snippet with the combination of monitoring a parameter that is stored in an external parameter outside of the scaling infrastructure.

9 Conclusion and Future Work

The work discussed in this paper and realized by the Policy Keeper component supports the definition of scaling rules by using program code to reach full flexibility. Beyond the programmable scaling rules, the dynamic attachment of monitoring sources (i.e. Prometheus exporters) are also

```
data:
  constants:
    AET: 25
    DEADLINE: 1530270216
    MAXNODES: 3
    MAXCONTAINERS: 6
    cqueue.server.ip.address: ...
  sources:
    - '{{cqueue.server.ip.address}}:8090'
  queries:
    REMAININGTIME: '{{DEADLINE}}-time()'
    ITEMS: 'rabbitmq_queue_messages_persistent{queue="machinery_tasks"}'
scaling:
  nodes:
    - min: 1
      max: '{{MAXNODES}}'
      target: |
        reqnodes=0
        if ITEMS>0:
          reqconts = ceil(AET/(REMAININGTIME/ITEMS)) if REMAININGTIME>0 else MAXCONTAINERS
          reqnodes = ceil(reqconts/2)
          if reqnodes<m_node_count-1:
            m_node_count-=1
          if reqnodes>m_node_count:
            m_node_count+=1
        else:
          m_node_count = 0
  services:
    - name: "cqueue_worker"
         min: 1
         max: '{{MAXCONTAINERS}}'
         target:
        required_count = 0
        if ITEMS>0:
          required_count = ceil(AET/(REMAININGTIME/ITEMS)) if REMAININGTIME>0 else MAXCONTAINERS
          m_container_count = min(required_count, len(m_nodes) * 2)
        else:
          m_container_count = 0
```

Code 1 Scaling policy for processing items from an external queue supported in order to provide flexibility on accessing and utilizing monitoring metrics in the scaling rules as well. This approach has been implemented in a standalone component called Policy Keeper. In the framework of the COLA EU project Policy Keeper component has been integrated into the MiCADO orchestration framework, where provision of cloud and container resources are done by Occopus and Docker Swarm, while monitoring is performed by the Prometheus monitoring system. The modularity of MiCADO is demonstrated by a new implementation [23] where Kubernetes replaces Docker Swarm. In the background Terraform is also integrated into MiCADO as an alternative for Occopus. Due to the flexibility of programmable scaling rules, there are a wide variety of scaling scenarios which are supported. In the COLA EU project, the goal is to support more than 20 applications with different technologies, requirements and rules.

To improve the support for the development of scaling policies, the next step aims the provision of an environment where the scaling policy can be tested against different circumstances. The testing environment should support the analysis of the scaling policy before moving it in production. Further plans are targeting the support for machine learning algorithms where reinforcement learning algorithms could be a good candidate to support the Policy Keeper component.

Acknowledgements This work was funded by the European COLA - Cloud Orchestration at the Level of Application project under grant No. 731574 (H2020-ICT-2016-1). We thank for the usage of MTA Cloud (https://cloud.mta.hu/) that significantly helped us achieving the results published in this paper.

Funding Information Open access funding provided by MTA Institute for Computer Science and Control (MTA SZTAKI).

References

1. Buyya R, Broberg J, Goscinski AM. Cloud Computing: Principles and Paradigms. Wiley: Hoboken, New Jersey, 2011
2. Mell P, Grance T. The NIST definition of Cloud computing. NIST special publication 800-145 (final). Technical Report, 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
3. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in Cloud Computing: state of the art and research challenges. IEEE Transactions on Services Computing (TSC). **11**(2), 430–447 (2018)
4. The MTA Cloud infrastructure, https://cloud.mta.hu [March 05, 2019]
5. Research projects supported by MTA Cloud, https://cloud.mta.hu/en/projektek [March 05, 2019]
6. COLA: Cloud Orchestration at the Level of Application, http://www.project-cola.eu [March 05, 2019]
7. AWS Auto Scaling, https://aws.amazon.com/autoscaling/ [March 05, 2019]
8. Rightscale, website http://www.rightscale.com [March 05, 2019]
9. Galante, G., Bona, L.C.E.D.: A programming-level approach for elasticizing parallel scientific applications. J. Syst. Softw. **110**, 239–252 (2015)
10. Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. Queue 14, 1, Pages 10 (2016), DOI: https://doi.org/10.1145/2898442.2898444
11. Cloud Native Computing Foundation, https://www.cncf.io [March 05, 2019]
12. Cloudify, http://getcloudify.org/ [March 05, 2019]
13. Topology and Orchestration Specification for Cloud Applications, TOSCA, http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html [March 05, 2019]
14. Docker, http://www.docker.com [March 05, 2019]
15. Prometheus monitoring system, https://prometheus.io [March 05, 2019]
16. Kovacs J., Kacsuk P., Occopus: a multi-Cloud orchestrator to deploy and manage complex scientific infrastructures. Journal of Grid Computing, vol 16, issue1, pp 19–37, 2018
17. Prometheus exporters, https://prometheus.io/docs/instrumenting/exporters/ [March 05, 2019]
18. Official YAML Web Site, http://yaml.org [March 05, 2019]
19. Terraform, https://www.terraform.io [March 05, 2019]
20. Jinja2, http://jinja.pocoo.org/docs/2.10 [March 05, 2019]
21. Flask, http://flask.pocoo.org/ [March 05, 2019]
22. Kiss, T., Kacsuk, P.: Jozsef Kovacs et all: MiCADO—microservice-based Cloud application-level dynamic orchestrator. Future Generation Computer Systems, Volume. **94**, 937–946, ISSN 0167-739X (2019). https://doi.org/10.1016/j.future.2017.09.050
23. Official documentation site of MiCADO, https://micado-scale.readthedocs.io/en/0.6.1 [March 05, 2019]
24. Prometheus Node Exporter, https://prometheus.io/docs/guides/node-exporter/ [March 05, 2019]
25. Prometheus CAdvisor, https://prometheus.io/docs/guides/cadvisor/ [March 05, 2019]
26. RabbitMQ exporter for Prometheus, https://github.com/kbudde/rabbitmq_exporter [March 05, 2019]
27. CQueue simple container queueing system, http://www.lpds.sztaki.hu/occo/user/html/tutorial-building-clusters.html#cqueue-cluster [March 05, 2019]
28. Chen, H.Y., Hsiung, M., Lee, H.C., Yen, E., Lin, S.C., Wu, Y.T.: GVSS: A High Throughput Drug Discovery Service of Avian Flu and Dengue Fever for EGEE and EUAsiaGrid. J Grid Computing. **8**, 529–541 (2010). https://doi.org/10.1007/s10723-010-9159-7