A Model-Driven Engineering Approach for Flexible and Distributed Monitoring of Cross-Cloud Applications

Daniel Baur, Frank Griesinger, Yiannis Verginadis, Vasilis Stefanidis and Ioannis Patiniotakis

10.1109/UCC.2018.00012

A Model-Driven Engineering Approach for Flexible and Distributed Monitoring of Cross-Cloud Applications

Daniel Baur, Frank Griesinger
Institute for Information Resource Management
Ulm University, Germany
{firstname.lastname}@uni-ulm.de

Yiannis Verginadis, Vasilis Stefanidis, Ioannis Patiniotakis
Institute of Communications and Computer Systems
National Technical University of Athens
Greece
jverg@mail.ntua.gr, stefanidis@central.ntua.gr, ipatini@mail.ntua.gr

Abstract—**Cloud computing and its computing as a utility paradigm offers on-demand resources, enabling its users to seamlessly adapt applications to the current demand. With its (virtually) unlimited elasticity, managing deployed applications becomes more and more complex raising the need for automation. Such autonomous systems leverage the importance to constantly monitor and analyse the deployed workload and the underlying infrastructure serving as knowledge-base for deriving corrective actions like scaling. Existing monitoring solutions, however are not designed to cope with a frequently changing topology. We propose a monitoring and event processing framework following a model-driven approach, that allows users to express *i)* the monitoring demand by directly referencing entities of the deployment context, *ii)* aggregate the monitoring data using mathematical expressions, *iii)* trigger and process events based on the monitoring data and finally *iv)* attach scalability rules to those events. We accompany the modelling language with a monitoring orchestration and distributed complex event processing framework, capable of enacting the model in a frequently changing multi-cloud infrastructure, considering cloud-specific aspects like communication costs.**

Index Terms—**cloud computing, monitoring, complex event processing**

I. Introduction

Cloud computing and its paradigm of *computing as a utility* [1] provides on-demand compute, storage and network resources to its users. This enables users to seamlessly adapt the allocated resources to the current demand of the application by applying horizontal (or vertical) scale operations in times of higher respectively lower demand. However, to make full use of this elasticity, the adaptation process requires automation and hence an adaptive system.

This need lead to the development of cloud orchestration (management) tools [2] that automate the deployment and the management of an application across multiple cloud providers. For this purpose, they typically follow a model-driven engineering (MDE) approach, where the user first describes his application and the required resources using a modelling language, e.g. the Topology and Orchestration Specification for Cloud Applications (TOSCA [3]) or the Cloud Application Modelling and Execution Language (CAMEL [4]). Following the MDE paradigm implies the use of tools that typically apply the MAPE [5] loop by monitoring the current context of the application and apply scalability rules to either scale-out or scale-in individual components of the application. Thus, it is very important to constantly monitor and analyse infrastructures health status to detect situations that may reveal adaptation needs or optimisation opportunities. Monitoring delivers the knowledge that is required to make appropriate mitigation decisions with respect to the adopted infrastructure and deployed applications.

Yet, most traditional monitoring tools are badly suited for monitoring cloud deployments [6] especially in multi-cloud scenarios for two main reasons: *i)* they are designed for static or slowly changing infrastructures, *ii)* they are not aware of cloud specific attributes like monetary costs for data transmission across region or cloud boundaries. Applying such tools in an elastic, thus quickly changing environment may lead to the following problems: *i)* in dynamic placement scenarios, where the underlying infrastructure is derived or even may change during runtime (e.g. by matchmaking or placement algorithms) a infrastructure centric approach is not possible, *ii)* typically the superset of monitoring information is gathered on all allocated resources wasting storage and network resources, *iii)* all data is stored in a central database where it is processed to derive e.g. aggregated values and trigger events again consuming network resources and *iv)* the tools need to be manually adapted to follow the changing infrastructure or handle the increased load.

To overcome these shortcomings we propose an adaptive and distributed monitoring and event processing framework accompanying our existing cloud orchestration tool. We adopt a model-driven engineering approach for depicting the monitoring demand and the processing of monitoring data. We further propose an architecture capable of automatically adapting the monitoring demand to either changes *i)* in the monitoring demand expressed by the user or *ii)* in the underlying topology that is composed of allocated resources and deployed applications. In addition, we adopt the architecture to the cloud environment reducing costly cross-region/-cloud traffic and achieve an architecture that automatically scales with the monitored system.

The remainder of this paper is structured as follows: First, Section II gives background information establishing a common understanding of the vocabulary and previous work used throughout the paper. Second, Section III gives an overview of our approach depicting a high-level architecture. Afterwards, Sections IV and V depict the architecture and implementation of the monitoring and the distributed complex event processing in detail. Subsequently, Section VI discusses related work in this area before Section VII concludes the paper.

II. Background

As our work is partly build upon existing work this section will briefly explain the concepts used within this paper. In addition, this section establishes a running example, that is referenced in later sections of the paper.

A. Cloudiator

Cloudiator [7]–[9] is a cross-cloud orchestration engine, capable of acquiring resources from a multi-cloud environment, and deploying (possibly) distributed applications across these resources.

The application model of Cloudiator is runtime-driven, consisting of three main entities: *i)* a *Job* as a logical group of multiple *ii) Tasks* that depict the workload that needs to be executed on an allocated resource (e.g. running a MySQL server or executing a batch file) by exporting *Interfaces* and *iii) Processes* that depict the instantiation of a *Task* currently running on an allocated resource. For resources, Cloudiator uses the abstract concept of a *Node*, that can represent a virtual machine started at an IaaS provider, but also existing servers or even PaaS components.

Cloudiator relies on a provider-agnostic application model, meaning that it does not reference any provider specific details like identifiers. Instead it is based upon an application focused requirement specification relying on a constraint language [10] that recently was extended to also support constraints expressed in the Object Constraint Language (OCL) [11]. To enact the requirements expressed by the user, Cloudiator automatically discovers offers of previously registered cloud providers. Cloudiator translates the discovered offers into constraints that are combined with the constraints provided by the user to form a constraint satisfaction problem (CSP). This CSP is then solved to derive the infrastructure resources (e.g. virtual machines) required for hosting the workload.

An (abbreviated) example for a job description is given in Listing 1. It defines a three-tier installation of MediaWiki[1], that consists of three individual tasks: *i)* a load balancer splitting load across (possibly) multiple instances of the wiki, *ii)* the wiki application itself and *iii)* a database storing the content.

The deployment process of Cloudiator is split into two phases: *i)* the allocation phase where Cloudiator retrieves the requirements attached to each *Task* defined in the job model, then solves the resulting CSP and contacts the cloud provider APIs to allocate the resources for each *Task* and the *ii)* deployment phase were Cloudiator connects to the allocated resources and creates *Processes* by executing the workload attached to each *Task*.

[1]https://www.mediawiki.org/

Listing 1: Job Description Example

```
name: mediawiki
tasks:
- name: wiki
  ports:
  - type: PortRequired
    name: WIKIREQMARIADB
  - type: PortProvided
    name: WIKIPROV
    port: 80
  interfaces:
  - type: LifecycleInterface
    preInstall: ./preInstall.sh
    install: ./install.sh
    postInstall: ./postInstall.sh
    start: ./start.sh
  requirements:
  - constraint: nodes->forAll(hardware.cores
   ↪  >= 2)
    type: OclRequirement
  - constraint: nodes->forAll(hardware.ram >=
   ↪  2048)
    type: OclRequirement
  - constraint:
   ↪  nodes->forAll(image.operatingSystem =
   ↪  'UBUNTU')
    type: OclRequirement
- name: loadbalancer
  ports: ...
  interfaces: ...
  requirements: ...
- name: database
  ports: ...
  interfaces: ...
  requirements: ...
communications:
- portRequired: WIKIREQMARIADB
  portProvided: MARIADBPROV
- portRequired: LOADBALANCERREQWIKI
  portProvided: WIKIPROV
```

B. Scalability Rule Language

To express the scalability requirements we rely on the Scalability Rule Language (SRL) [12] that allows to *i)* define (raw) metrics representing the current state of the system (e.g. CPUUsage representing the CPU usage of one node), *ii)* derive composite metrics by expressing mathematical operations on raw metrics (e.g. calculate the average) *iii)* trigger (simple) events based on threshold violations (e.g. if average CPU usage of the last five minutes is above 90% trigger event), *iv)* concatenate events by logical operations (e.g. if event OR event) and finally *v)* attach scaling actions to events (e.g. if event then scale component horizontally).

Listing 2: Scalability Rule example

```
# Provides the formula to calculate the
# CPU Average (MEAN)
composite metric CPUAverage {
    metric formula Formula_Average {
      function arity: UNARY
      function pattern: REDUCE
      MEAN( CPUUsage )
    }
}
# Context that uses the CPUAverage formula
# to calculate the average of all nodes
composite metric context CPUAvgMetricContextAll {
   metric: CPUAverage
   window: 5 min
   schedule: 1 min
   quantifier: ALL
}
# Context that uses the CPUAverage formulate
# to calculate the average of each node
composite metric context CPUAvgMetricContextAny {
   metric: CPUAverage
   window: 1 min
   schedule: 1 min
   quantifier: ANY
}
# condition > 50.0
metric condition CPUAvgMetricConditionAll {
   context: CPUAvgMetricContextAll
   threshold: 50.0
   comparison operator: >
}
# condition > 80.0
metric condition CPUAvgMetricConditionAny {
   context: CPUAvgMetricContextAny
   threshold: 80.0
   comparison operator: >
}
# defines a scale out of the wiki task
horizontal scaling action HorizontalScalingWiki {
   type: SCALE_OUT
   task: wiki
}
# event is fired if CPUAvgMetricConditionAll
# condition is violated
non-functional event CPUAvgMetricNFEAll {
   metric condition: CPUAvgMetricConditionAll
}
# event is fired if CPUAvgMetricConditionAny
# condition is violated
non-functional event CPUAvgMetricNFEAny {
   metric condition: CPUAvgMetricConditionAny
}
# combines both non-functional events
# using AND operator
binary event pattern CPUAvgMetricBEPAnd {
   left event: CPUAvgMetricNFEAll
   right event: CPUAvgMetricNFEAny
   operator: AND
}
# defines that horizontal scaling action
# is execute if event is raised
scalability rule CPUScalabilityRule {
   event: CPUAvgMetricBEPAnd
   actions [HorizontalScaling]
}
```

An (abbreviated) example for a scalability rule is given in Listing 2, defining the rule: *if the average CPU load on all nodes hosting the wiki application is above 50% over the last five minutes and the average CPU load on one node hosting the wiki is above 80% over the last minute then horizontally scale the wiki task.*

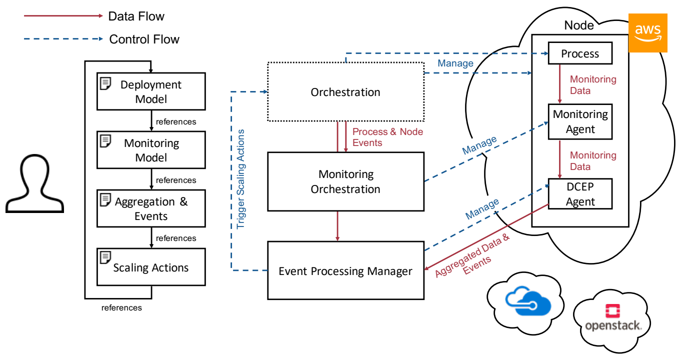

Fig. 1: High-level architectural overview

III. Overview

Figure 1 gives an overview to our model-driven monitoring approach. We ask the user to provide a model, consisting of four sub-models referencing each other: *i)* the deployment model depicting the (application) workload that needs execution and its infrastructure requirements (cf. Section II-A), *ii)* an application-centric monitoring model depicting the modelling demand by referencing entities of the deployment model (cf. Section IV-A), *iii)* an aggregation and event model allowing to derive higher level metrics (e.g. by aggregation) and trigger events (e.g. on thresholds violations) and *iv)* a model depicting scaling actions that are attached to previously defined events (cf. Section II-B).

To be able to constantly adapt the monitoring demand and to achieve a distributed processing of the gathered data, the architecture consists of three high-level components: *i)* an orchestration component handling allocation of resources and the deployment, the *ii)* monitoring orchestration handling the monitoring and *iii)* the DCEP component handling aggregation of monitoring data and events. The **Orchestration** component is external to the monitoring system, but is responsible for announcing topology changes to the monitoring system, e.g. if new resources are allocated or new processes are deployed. As all state changes within our tool are announced using events, the monitoring system can easily subscribe to events it is interested in. The **Monitoring Orchestration** component takes the monitoring model as input and subscribes to aforementioned events of the orchestration layer. Whenever topology changes are announced it *i)* derives the new demand and *ii)* connects to the **Monitoring Agents** to enact it. The same applies to the **Event Processing Manager** that reconfigures the DCEP agents. The **DCEP Agents** receive (raw) monitoring data collected from the **Monitoring Agents** and aggregate those metrics and evaluate threshold conditions to trigger events in a distributed manner. If events are triggered that are connected to scalability rules, the event processing manager calls the orchestration layer that enacts the change.

IV. Monitoring Framework

The task of the monitoring framework is to collect the raw metrics (cf. Section II-B) and pass them to the DCEP framework described in Section V that aggregates and processes those metrics and triggers scaling events. For this purpose it consists of three components: *i)* a modelling Language allowing the user to define the monitoring demand *ii)* the monitoring orchestration whose task is to instruct the monitoring agents to load the correct sensors based on which workload is running on each node *iii)* and the monitoring agents that run on every managed node and are responsible for collecting the monitoring information.

A. Monitoring Model

To capture the monitoring demand, we rely on the user providing a model as depicted in Figure 2. The user can define multiple *Monitors* each gathering a defined metric using a *Sensor*. A *Sensor* may be represented by a *PushSensor* that starts a telnet server on the optionally defined port or a random port that will be advertised to the running process by the orchestration layer. This telnet server can then be used by the running process to propagate (application specific) monitoring data (cf. Section IV-C). Additionally a *PullSensor* represents a (traditional) *Sensor* were data is gathered by executing logic collecting data from the underlying node, operating system or the running process. For this purpose the *PullSensor* refers to a class name loading the logic and optionally an URL where the logic can be loaded from. A *PullSensor* runs at a defined *Schedule* depicting the frequency at which the data is gathered. A *DataSink* defines the endpoint where the collected monitoring data will be reported. The default *DataSink* is the DCEP framework (cf. Section V) but the user may add multiple other *DataSinks* like time series databases e.g. for historical storage or visualisation. Several *Tags* can be attached to a monitor. A *Tag* provides context information in the form of key-value pairs, that is later attached to the gathered sensor data. By default, the monitoring framework attaches information about the environment the sensor is running on (e.g. the id of the node) and entities it relates to (e.g. task or process). This information is crucial for the aggregation process as it allows the DCEP framework (cf. Section V) to correlate the gathered monitoring data. Finally, the *Target* links the monitoring model to the application model described in Section II-A. A monitor can be either linked to a *Job*, a *Task*, a *Process* or a *Node*. The linking follows the depicted hierarchy, meaning that if the user e.g. links the monitor to a specific *Job*, all *Tasks* of this *Job* and subsequently all *Processes* and *Nodes* corresponding to the *Job* will be monitored. Additionally, the user may express a query constraint in OCL, to further limit the target entities by targeting their underlying attributes. The query `job.tasks.processes.node->select(n | n.cloud.type = CloudType::Private)` will e.g. monitor all nodes that belong to the target job and are running on a private cloud.

Listing 3 gives an example of a monitor, monitoring the CPU usage on all nodes where a process of the *Wiki* task of the *MediaWiki* job is installed.

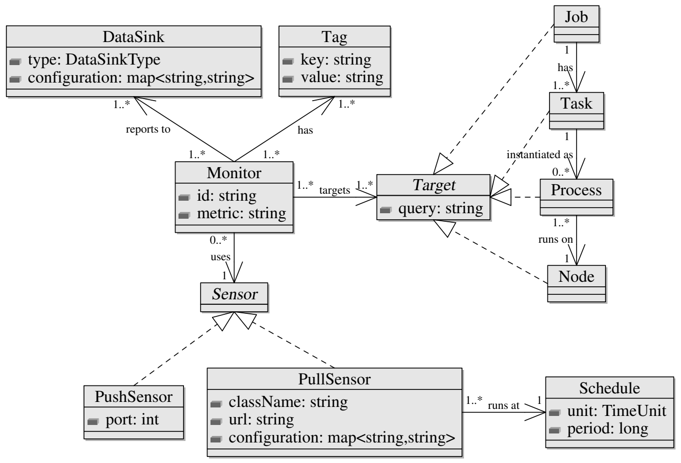

Fig. 2: Monitoring Demand

Listing 3: CPU Usage example

```
metric: CPUUsage
targets:
- type: Task
  identifier: wiki
sensor:
  type: PullSensor
  className: org.cloudiator.CPUSensor
  schedule:
    unit: SECONDS
    period: 5
sinks:
- type: INFLUX
  configuration:
    INFLUX_URL: http://example.org:8086
```

B. Monitoring Orchestration

The monitoring orchestration component is responsible for *i)* capturing the monitoring demand by the user (cf. Section 2) and *ii)* connecting the deployment layer (cf. Section II-A) with the monitoring layer by instructing the monitoring agents to load the correct sensors based on the monitoring demand given by the user, and the running entities started by the deployment layer.

The basic workflow of the monitoring orchestration is shown in Figure 3. It depicts the main components of the monitoring framework: *i)* the monitoring agent running on nodes and *ii)* the monitoring orchestration component. In addition, it depicts two other components the monitoring orchestration interacts with: *i)* the node agent responsible for allocating new resources and *ii)* the process agent, responsible for executing workload on the allocated nodes.

As first step (1), the user registers the monitoring demand either via a YAML syntax as depicted in Figure 2 that is parsed and translated into API calls by a client, or directly via the RESTful API exposed by the monitoring orchestration component. The expressed demand is stored in an internal database (2). Afterwards, the monitoring orchestration subscribes to events (3) submitted from the node agent and process agent, announcing changes in the topology i.e. whenever a new node

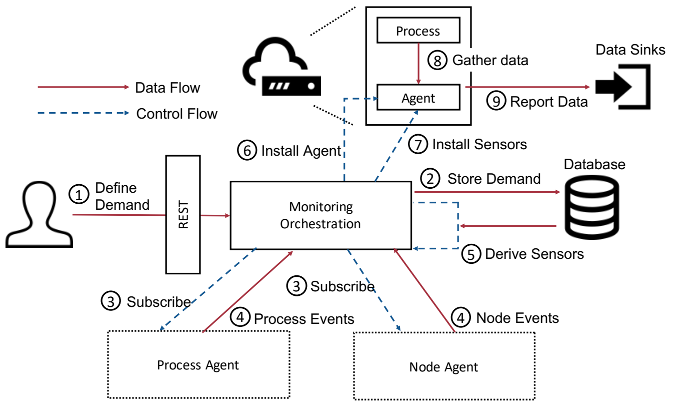

Fig. 3: Monitoring Orchestration Workflow

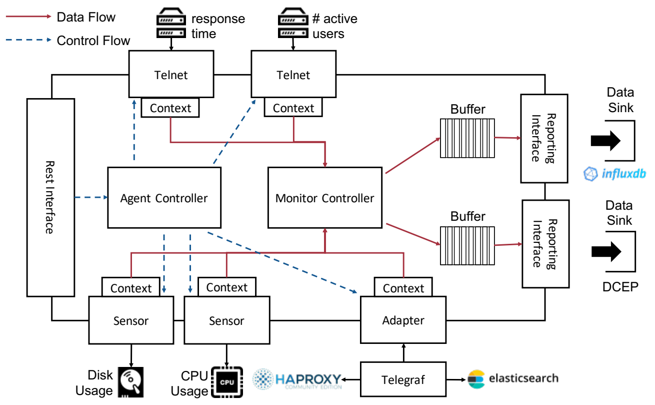

Fig. 4: Monitoring Agent Architecture

respectively a new process is started or terminated. Whenever a corresponding event is received (4) by the orchestration component, it retrieves the monitoring demand given by the user and matches it to the retrieved event based on the condition expressed by the target entity (and optionally by the filter expression formulated within) (5). Whenever the new process or node matches and thus is required to be monitored, the orchestration component will install the agent on the target machine (if not installed yet) (6). Afterwards it will contact the agent, and reconfigure it (7) to start the required sensors also providing context information with respect to the event triggering the reconfiguration (using the tags entity described in Section IV-A). Finally, the agent will gather the monitoring data (8) and report it to the configured data-sinks (9). This process is covered in more detail in Section IV-C.

Whenever the topology of the deployed job (processes or nodes) changes, or the user makes modifications to the monitoring model a reconfiguration process is triggered. In this case, the already enacted model (stored within the monitoring agents) is also considered when deriving the required sensors. The calculated difference is finally configured within the already running agents.

C. Monitoring Agent

The task of the monitoring agent is to monitor the underlying node or processes running on the same node the agent is deployed on. Figure 4 gives an detail overview of the agent's architecture. The agent exposes a **RESTful interface** that is used by the monitoring orchestration component (cf. Section IV-B) to configure the agent during runtime, e.g. if the user changes the monitoring model or a new process is spawned on the node the agent is running on. The request is forwarded to the **Agent Controller** which is responsible for managing all entities of the agent.

The monitoring agent employs three strategies to gather monitoring data: *i)* sensors actively gathering data on the underlying host by executing code gathering the metric *ii)* telnet sockets that allow the running process to push data to the monitoring agent and *iii)* adapters interfacing with third-party monitoring tools giving access to existing sensor implementations. **Sensors** are Java programs implementing the gathering logic. We provide by default multiple sensors gathering system information like CPU, memory, disk and network usage by either relying on management interfaces provided by the Java virtual machine (JVM) or the SIGAR API[2]. In addition, we provide an abstract implementation for sensors, that facilitates the development of custom sensors, that can be loaded during runtime. After loading the sensor, it is run at the user-defined schedule. The **Telnet** servers provide the possibility for running processes to actively push monitoring data to the agent following a simple line based protocol: <metric_name/_id, value, timestamp>. As multiple processes may run on the same node, the agent is capable of starting multiple servers on different ports, to be able to correctly correlated context information. To ensure that only applications installed on the same node can interact with the telnet server, it only listens on the local interface by default. Finally, the agent provides **Adapters** to existing sensor frameworks (currently Telegraf[3] is supported) giving access to a large variety of already implemented sensors.

Internally, the monitoring agent relies on two concepts of monitoring data *i)* (raw) measurements (value and timestamps) that are emitted by the sensors and *ii)* metrics that originate from measurements but contain additional context information and are bound to one data sink. This allows to e.g. duplicate measurements for multiple data sinks or annotate sensor data with different contexts. The **Monitor Controller** is responsible for converting the (raw) measurements to metrics taking the context of the sensors into account and route the metrics to each data sink.

Finally, **Reporting Interfaces** provide the implementation for publishing the metrics to data sinks. They are responsible for converting the data format of the monitoring agent to the target format of the data sink and finally transmit it. Currently, the agent supports multiple time series databases (InfluxDB[4],

[2]https://github.com/hyperic/sigar
[3]https://www.influxdata.com/time-series-platform/telegraf/
[4]https://www.influxdata.com/time-series-platform/influxdb/

KairosDB[5], Druid[6]) and message queues (Kafka[7], ActiveMQ[8]) but support for different databases and queues can be provided by implementing the required conversion and publishing logic. For each reporting interface, the agent uses **Buffers** to cater for the fact that it may be more efficient to report multiple metrics at once and to be able to overcome short term network outages. Since the buffers may delay the reporting of metrics, they can be disabled.

V. Distributed Complex Event Processing (DCEP)

Besides implementing a flexible and adequate way for deploying monitoring sensors in cross-clouds application scenarios (cf. Section IV), it is critical to accommodate the appropriate flexible framework which is able to cope with an unknown and unbounded number of monitoring events, aggregate, filter and correlate them in order to guide application adaptations according to the scalability requirements expressed (cf. Section II-B). In this section, we discuss the details of such a framework that we call Distributed Complex Event Processing (DCEP) framework.

A. DCEP Architecture

In previous work [13], we have introduced a distributed complex event processing architecture that follows automatically the infrastructural resources commissioned at any given time for efficiently monitoring their health status and detecting optimisation opportunities. This novel approach refers to an Event Processing Network (EPN) [13] as seen in Figure 5. The approach was introduced for efficiently distributing over several virtualised resources that may span multiple cloud providers to monitor the deployment of multi-cloud applications and reconfigure them based on the perceived workload fluctuations and health status of the underlying infrastructures. This mechanism is flexible enough since it uses a hierarchical event processing approach to avoid message flooding incidents that may stall the processing of monitoring events and may delay the detection of optimisation opportunities. In order to achieve this, on each private or public node, a Distributed Event Processing Agent (DCEP) is deployed (Figure 5) that encapsulates: *i)* publish/subscribe functionalities (based on ActiveMQ[9]) for efficiently propagating monitoring events and *ii)* rich event processing capabilities (based on the powerful ESPER[10] engine) for hierarchically processing all the intercepted monitoring data (i.e. bringing the processing closer to the infrastructure that produces the monitoring data). The communication among the layered DCEP Agents and Event Processing Manager, occurs using secure protocols, namely HTTPS and SSH. We note that this approach is based on the publish/subscribe paradigm which is superior to any other time-based polling approach, with respect to a

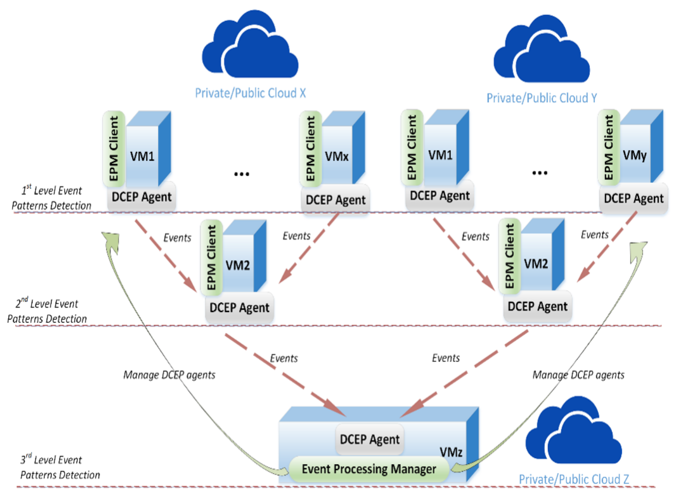

Fig. 5: Conceptual Architecture of a DCEP for Monitoring Cross-Cloud Applications

dynamic environment where the location of the event sources is not known beforehand and can change at any given time upon a reconfiguration decision. Moreover, the introduction of this layered approach allows for proper filtering of events that are allowed to reach the upper levels of this topology. In this way, we achieve the minimum network bandwidth consumption for monitoring purposes while at the same time we guarantee the immediate and efficient detection of situations that dictate mitigation actions. With respect to the event processing capabilities, we use efficient CEP engines able to perform fast and complex correlations and processing of events over certain time or event sliding windows. Of course, the size of these windows has a direct impact on the heap size that should be available to the DCEP Agent. Furthermore, in cases where there is a need for persisting all the events transmitted then our approach could be combined with the use of a time-series database per each cloud used. Obviously, the adoption of a time-series database just for its querying and alerting capabilities would have a significant impact to the event processing expressivity that can be supported.

B. Translating Scalability Requirements to Hierarchical Complex Event Patterns

Listing 4: SRL to EPL rules translation algorithm

```
algorithm srl2epl is
input: SRL_model srl
output:
  DAG ← {}, /*Decomposition graph*/
  S2T ← [], /*Sensor−to−target list*/
  E2A ← [], /*Event−to−action list*/
  R2L ← []  /*EPL Rule−to−Level list*/
/* Process SRL model */
sc_rules ← Get_Scalability_Rules(srl)
for each scalability_rule in sc_rules do
   event ← Get_Event(scalability_rule)
   action ← Get_Action(scalability_rule)
   E2A ← E2A ∪ [event, action]
   node ← Find_Node(DAG, event)
```

[5]https://kairosdb.github.io/
[6]http://druid.io/
[7]https://kafka.apache.org/
[8]http://activemq.apache.org/
[9]https://activemq.apache.org/
[10]http://www.espertech.com/esper/

```
15:     if node = ∅ then
16:       node ← Add_Root(DAG, event)
17:       node→Level ← Get_Level(srl, event)
18:       Decompose_Event(DAG, node)
19:     else
20:       if not Is_Root(DAG, node) then
21:         Make_Root(DAG, node)
22:       end if
23:     end if
24: end for
25: for each simple_event in Get_Leafs(DAG) do
26:     condition ←
27:       Get_Condition(srl, simple_event)
28:     operator ←
29:       Get_Condition_Operator(condition)
30:     metric ← Get_Condition_Metric(condition)
31:     Update_Simple_Event (DAG, simple_event,
32:       operator, metric)
33: end for
34: for each raw_metric in Get_Leafs(DAG) do
35:     sensor ← Get_Sensor(Camel, raw_metric )
36:     target ←
37:       Get_Target(srl, raw_metric)
38:     S2T ← S2T ∪ [sensor, target]
39:     Update_Raw_Metric(DAG, raw_metric, sensor)
40: end for
41: /* Generate EPL Rules and Event Topics */
42: for each node in Get_All_Nodes(DAG) do
43:     node→topic ← Create_Event_Topic(node)
44:     name ← Get_Name(node)
45:     operator ← Get_Operator(node)
46:     level ← Get_Level(node)
47:     children ← Get_Children(DAG, node)
48:     rule ← Generate_EPL_Rule(name, topic,
49:       operator, level, children)
50:     R2L ← R2L ∪ [rule, node->level]
51: end for
52: procedure Decompose_Event is
53:     input event, DAG
54:     output DAG
55: begin
56:     if Is_Composite_Event(event) then
57:       oper ← Get_Composition_Operator(event)
58:       children ← Get_Constituents(event)
59:       node ← Find_Node(DAG, event)
60:       node→operator ← oper
61:       for each child in children do
62:         node' ← Find_Node(DAG, child)
63:         if node' = ∅ then
64:           node' ← Add_Node(DAG,event ,child)
65:           node'→Level←Get_Level(Camel,child)
66:           Decompose_Event(DAG, child)
67:         end if
68:       end for
69:     end if
70: end
```

An important step in the overall process of deploying a DCEP for monitoring multi-cloud applications is the translation of the expressed rules in the SRL, to complex event processing patterns, which will be used in the various DCEP agents. Our implementation of the translation algorithm generates rules in the Event Processing Language (EPL) used by the ESPER CEP engine.

The translation algorithm involves the analysis of the scalability rules in the SRL model and the extraction of variables corresponding to monitored values (called metrics). Scalability rules correlate specific trigger events with scaling actions (e.g. scale in/out) that reconfigure a current placement topology (steps: 9-13 of Listing 4). Events and metrics can either be composite (i.e. calculated using simpler events or metrics), or simple. For this reason the translation algorithm decomposes events/metrics into a multi-root, directed acyclic graph (DAG) called decomposition graph. Each trigger event and each metric variable directly used in an expression becomes a root in the decomposition graph (steps: 14-23 of Listing 4). The roots are then recursively decomposed into their simplest components. Subsequently, simple events are converted into equivalent metric expressions (SRL defines simple events as metric expressions violating a threshold - steps: 25-33 of Listing 4). Again metric variables are extracted and decomposed into simple metrics, which are measured using specific sensors (steps: 34-40 of Listing 4). Care is taken to avoid inserting the same event/metric in the decomposition graph more than once. Eventually, the DAG is traversed and EPL rules are generated for every node (steps: 41-51 of Listing 4). Each node represents a specific application event type or metric variable, associated to a monitoring level. Nodes representing composite structures include information about the composition operation, while the node's children are the components being composed (steps: 52-69 of Listing 4). Based on the structure type (event or metric variable), the monitoring level as well as the composition operator, a suitable EPL rule template is used to produce the corresponding EPL rule. The produced EPL rules are grouped per monitoring level. These rulesets are then used to initialize the CEP engines of DCEP Agents, based on their position in the EPN (i.e. 1st level DCEP Agents receive Level 1 EPL ruleset etc.). The Event Processing Manager is responsible to send the rulesets to the appropriate DCEP Agents through secure communication channels (i.e. SSH).

Next, the main logic and the decomposition procedure of the translation algorithm are given in pseudo code (Listing 4). For the sake of brevity the implementation of various trivial, auxiliary subroutines used in pseudocode are omitted. Their functioning is easily deduced from their name (e.g. Get Level for acquiring the monitoring level of a node). The algorithm takes the target SRL as input. The outputs of the algorithm are the decomposition graph as well as various associative arrays correlating trigger events to actions, sensors to their targets and EPL rules to monitoring levels. Obviously the size and complexity of the SRL model influence the performance of the translation algorithm. However, since SRL translation to EPL rules occurs at multi-cloud application bootstrap and only when the application is remodelled, the performance of the algorithm is considered a minor issue.

C. An illustrative example

To clarify the algorithm, we apply it to the SRL example given in Listing 2. The algorithm generates the decomposition graph as depicted in Figure 6, which is used to create the

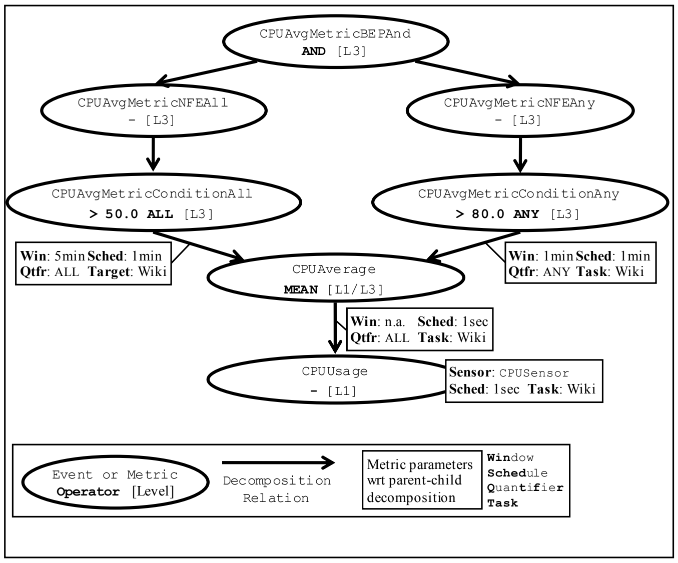

Fig. 6: Decomposition graph of the example

event streams and the EPL rules that will form the monitoring topology of the multi-cloud application. The decomposition process is as following.

- Retrieve scalability rules from SRL model: *CPUScalabilityRule*
- Extract trigger events: *CPUAvgMetricBEPAnd*
- Extract corresponding scaling actions: *HorizontalScalingWiki*
- Save into Event-to-Action map, including affected task: [*CPUAvgMetricBEPAnd*, *HorizontalScalingWiki*, *Wiki*]
- Add trigger event as decomposition graph root: *CPUAvgMetricBEPAnd*
- Acquire roots grouping and update node info: *3rd_level*
- Decompose root event into simple events, if composite: *CPUAvgMetricBEPAnd* is a binary event pattern, i.e. it is composite, therefore decomposition is needed
  - Get composition operator: *AND*
  - Get constituent events (left, right): *CPUAvgMetricNFEAll*, *CPUAvgMetricNFEAny*
  - Add constituent events in decomposition graph as *CPUAvgMetricBEPAnds* children
  - Acquire children groupings and update nodes info: again *3rd_level*
- Recursively decompose children nodes, if composite: *CPUAvgMetricNFEAll* and *CPUAvgMetricNFEAny* events are both non-functional event, i.e. simple events
- Get the metric condition of each simple event and add them as (sole) children under the corresponding events: *CPUAvgMetricConditionAll* and *CPUAvgMetricConditionAny*
- For metric conditions:
  - Get metric conditions' operators and thresholds and update nodes info: > 50.0, > 80.0
  - Get metric conditions' metrics: *CPUAverage* for both metric conditions
  - Add constituent metric into decomposition graph as metric condition nodes child. Both metric condition nodes have the same child: *CPUAverage*
  - Acquire child grouping and update child node info: *3rd_level*
  - Get metric parameters (window, schedule, related task and quantifier) for metric conditions and update decomposition relations (i.e. vertices from parent to children) info: *Win5Min Schedule1Min ALL Wiki* and *Win1Min Schedule1Min ANY Wiki*
  - Recursive decomposition of child metric, if composite: *CPUAverage* is composite, therefore decomposition is required:
    * Get composition operator / formula: *MEAN*
    * Get constituent metrics: *CPUUsage*
    * Add constituent metric in decomposition graph as *CPUAverage's* child
    * Acquire child grouping and update node info: *1st_level*
    * Get metric parameters (schedule, quantifier and related application component) for metric, from corresponding metric context and update decomposition relation info: *Schedule1Sec ALL Wiki*
  - Recursive decomposition of new child metric: *CPUUsage* is a raw metric, therefore no further decomposition is possible
  - Get the corresponding monitor, sensor, interval and target and update the node info: *CPUSensor*, *Interval1Seconds*, *Wiki*
  - Save sensor and target pair in Sensor-to-Target map, including interval parameters: [*CPUSensor*, *Wiki*, *Interval1Seconds*]. At this point decomposition graph is as depicted in Figure 6.
  - Generate Event Topics by traversing decomposition graph and update graph nodes with the event topic names. For example, for the *CPUAvgMetricBEPAnd* node an event topic named *L3_CPUAvgMetricBEPAnd* will be created in third level event broker, where the corresponding events will be published.
  - Generate EPL rules by traversing decomposition graph. For each graph node an EPL rule must be generated (except for simple events or metric conditions bound to simple events). The generated rules will be stored in EPL Rule-to-Level map. For example the *CPUAverage* and its context *CPUAvgMetricContextAll* are represented in EPL as Listing 5 shows.

Listing 5: CPU Average in EPL

```
insert into CPUAverageAll (metricValue,
    target, level, timestamp)
select
  avg(metricValue) as metricValue,
  target as target,
  3 as level,
  current_timestamp() as timestamp
from CPUUsage#time(5 min)
```

```
output last every 1 min
```

VI. Related Work

This section will discuss related work in *i)* the area of monitoring systems and *ii)* the area of distributed complex event processing.

A. Monitoring

Traditional monitoring tools like Ganglia[11] or Nagios[12] are typically used to monitor static or slowly changing infrastructures warranting manual configuration. However, in a rapidly changing environment as given by the elasticity and dynamic of Cloud Computing and adaptive systems, the manual configuration is infeasible.

Most cloud providers offer there own proprietary monitoring solutions. Amazon e.g. offers CloudWatch[13] to monitor infrastructure and applications deployed on their premises. Similarly, Ceilometer[14] provides a monitoring solution for Openstack[15] based clouds. However, relying on cloud provider specific monitoring solutions leads to vendor lock-in and is impossible in cross-cloud scenarios, where a user wants to allocate resources across multiple cloud providers to e.g. increase fault tolerance.

The works by [6], [14], [15] provide a taxonomy and overview across existing monitoring solutions.

[16] proposes a monitoring architecture able to combine multiple (external) monitoring sources using adapters and a query language able to access those sources at the same time, however requiring the user to handle the monitoring itself. JCatascopia [17] proposes a distributed architecture using multiple monitoring servers and agents that use a variation of the publish-subscribe messaging pattern to configure the monitoring demand using a central database for storage and evaluation. In contrast, we use a hierarchical, distributed event processing scheme not requiring central storage. Varanus [18] achieves in-situ monitoring by relying on a peer-to-peer network of monitoring agents and an event based programming model to evaluate the data stored in-memory across multiple selected nodes. There approach still warrants manual configuration of the agents and programming of the aggregation logic which we overcome by using a model-driven approach.

Cloudify[16], a cloud orchestration tool based on the TOSCA modelling language also allows the user to express monitoring demand and scalability actions. In contrast to our approach however, the monitoring is statically defined on infrastructure level during the design phase and can not be changed during runtime.

[11]http://ganglia.sourceforge.net
[12]https://www.nagios.org
[13]https://aws.amazon.com/cloudwatch/
[14]https://docs.openstack.org/ceilometer
[15]https://www.openstack.org/
[16]https://cloudify.co/

B. Distributed Complex Event Processing

Monitoring and efficiently analysing data for recognising the need for application placement reconfigurations correspond to challenging tasks that require sophisticated tools and methods. Event processing is a method of tracking and analysing streams of data about application-related or infrastructural-related health status occurrences that happen (i.e. events), and issuing some alerts based on them. Complex event processing (CEP), corresponds to event processing that combines data for inferring patterns of events that may suggest more complicated circumstances [19]. CEP systems [20] are valuable in digesting and processing a multitude of event streams. Their big advantage is the ability to collect information from various heterogeneous data sources and filter, aggregate or combine them over defined periods of time (i.e. time windows). The idea of using CEP for monitoring infrastructures and applications has been applied with respect to two types of architectural approaches: centralised and distributed. The centralised CEP architecture is based on a single CEP engine which processes all monitored data and detects patterns by using rules. On the other hand, the distributed CEP architecture consists of a set of cooperating CEP engines that exchange messages and are able to more efficiently detect event patterns by considering rules that differ according to the proximity of the processing engine to the event source. In existing centralised CEP approaches [21], [22], [23] huge bandwidth and computational capabilities are required and usually they lack robustness and scalability because of the single point of failure when processing vast amounts of health status data. On the other hand, the distributed CEP architectures such as the parallel CEP processing architecture of Hirzels [24] and the work of Ku et al. [25], present better performance in terms of data processing throughput, due to workload sharing across multiple CEP engines, and establish better scalability results without any risk of single point of failure. Nevertheless, all these cases are bound to the use of simple infrastructures (i.e. only private resources or only cloud resources used from a single provider), a fact that limits, by default, the capability to detect reconfiguration opportunities in real, complex infrastructures that the modern organisations nowadays adopt.

VII. Future Work and Conclusion

We have presented a MDE based approach that uses a model to allow a user to express *i)* the monitoring demand related to his application/workload, *ii)* derive aggregated metrics by expressing mathematical operations on the gathered monitoring data, *iii)* trigger and process events by placing conditions on the gathered or derived data and finally *iv)* react on events by executing scaling actions. We accompany the model with a monitoring orchestration framework capable of enacting the expressed monitoring demand in a cross-cloud environment and a DCEP framework able to process the gathered data in a distributed manner.

While we have yet to thoroughly evaluate our approach, we can show the viability of our approach taking the *CPUAverage* metric as an example. Transmitting the CPU usage from the resources to a central database would lead to the transmission of one message*instance/second as this represents the schedule at which it is measured. Calculating the average directly on each virtual machine and only transmitting the result reduces the amount of messages to message*instances/minute thus reducing the traffic significantly. While this comes at the cost of an increased overhead on each node, the scalability of our approach should outweigh this.

There are several points for improvement in future work. Currently, our monitoring approach is mainly focused on Infrastructure as a Service (IaaS) clouds, as we rely on access to the underlying host to gather data. While we are able to support other types of deployments (e.g. PaaS) by running a global monitoring agent where applications running in such an environment can report monitoring data to, this counteracts our distributed approach. Generally, it would be interesting to rely on the mechanisms offered by the providers in such cases. One approach could e.g. transform monitoring demands and scalability rules to the provider-specific languages, while more complex expressions are evaluated by our framework. With respect to the DCEP framework, we currently rely on the user to express the event level (cf. Figure 5) at which an expression is computed. This static assignment could be improved by relying on an algorithm automatically deriving the optimal placement of the computation based on e.g. the network costs/latency or the workload/utilisation on the allocated resources.

Acknowledgments

The research leading to these results has received funding from the EC's Framework Programme HORIZON 2020 under grant agreement number 731664 (MELODIC), 732667 (RECAP) and 732258 (CloudPerfect).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley, Tech. Rep., 2009.

[2] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and J. Domaschka, "Cloud orchestration features: Are tools fit for purpose?" in *UCC*, 2015.

[3] D. Palma and T. Spatzier, "Topology and orchestration specification for cloud applications (tosca)," *OASIS, Tech. Rep*, 2013.

[4] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, D. Romero, M. Orzechowski, G. Kapitsaki, and A. Achilleos, "The cloud application modelling and execution language (camel)," 2017.

[5] B. Jacob, R. Lanyon-Hogg, D. Nadgir, and A. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*, ser. IBM redbooks. IBM Corporation, International Technical Support Organization, 2004.

[6] J. S. Ward and A. Barker, "Observing the clouds: a survey and taxonomy of cloud monitoring," *Journal of Cloud Computing*, vol. 3, no. 1, p. 24, 2014.

[7] D. Baur, S. Wesner, and J. Domaschka, "Towards a model-based execution-ware for deploying multi-cloud applications," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2014, pp. 124–138.

[8] J. Domaschka, D. Baur, D. Seybold, and F. Griesinger, "Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine," in *9th Symposium and Summer School on Service-Oriented Computing*, 2015.

[9] D. Baur and J. Domaschka, "Experiences from building a cross-cloud orchestration tool," in *3rd Workshop on CrossCloud Infrastructures & Platforms*, 2016.

[10] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A provider-agnostic approach to multi-cloud orchestration using a constraint language," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2018, pp. 173–182.

[11] Object Management Group (OMG), "Object constraint language specification, version 2.4," 2014.

[12] K. Kritikos, J. Domaschka, and A. Rossini, "Srl: A scalability rule language for multi-cloud environments," in *6th International Conference on Cloud Computing Technology and Science*, 2014.

[13] V. Stefanidis, Y. Verginadis, I. Patiniotakis, and G. Mentzas, "Distributed complex event processing in multiclouds," in *Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing*, September 2018.

[14] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[15] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar, "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art," *Computing*, vol. 97, no. 4, pp. 357–377, Apr 2015. [Online]. Available: https://doi.org/10.1007/s00607-014-0398-5

[16] B. König, J. A. Calero, and J. Kirschnick, "Elastic monitoring framework for cloud infrastructures," *IET Communications*, vol. 6, no. 10, pp. 1306–1315, 2012.

[17] D. Trihinas, G. Pallis, and M. Dikaiakos, "Monitoring elastically adaptive multi-cloud services," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2015.

[18] J. S. Ward and A. Barker, "Cloud cover: monitoring large-scale clouds with varanus," *Journal of Cloud Computing*, vol. 4, no. 1, p. 16, 2015.

[19] O. Etzion, P. Niblett, and D. C. Luckham, *Event processing in action*. Manning Greenwich, 2011.

[20] W. A. Higashino, "Complex event processing as a service in multi-cloud environments," 2016.

[21] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[22] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo, "Approaching the internet of things through integrating soa and complex event processing," in *Handbook of research on demand-driven web services: Theory, technologies, and applications*. IGI Global, 2014, pp. 304–323.

[23] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-level performance monitoring of cloud services based on the complex event processing paradigm," in *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*. IEEE, 2012, pp. 1–8.

[24] M. Hirzel, "Partition and compose: parallel complex event processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, 2012, pp. 191–200.

[25] T. Ku, Y. Zhu, K. Hu, and L. Nan, "A novel distributed complex event processing for rfid application," in *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, vol. 1. IEEE, 2008, pp. 1113–1117.