Journal of Systems Architecture 95 (2019) 36–46

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Infracomposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows

Emad Heydari Beni[a,*], Bert Lagaisse[a], Wouter Joosen[a]

imec-DistriNet, KU Leuven, Leuven, 3001 Belgium

A R T I C L E I N F O

Keywords:
Workflow
Cloud computing
Middleware
Adaptability
Reflection
Parallel and distributed architectures

A B S T R A C T

The simulation and optimization of complex engineering designs in automotive or aerospace involves multiple mathematical tools, long-running workflows and resource-intensive computations on distributed infrastructures.

Finding the optimal deployment in terms of task distribution, parallelization, collocation and resource assignment for each execution is a step-wise process involving both human input with domain-specific knowledge about the tools as well as the acquisition of new knowledge based on the actual execution history.

In this paper, we present a policy-driven adaptive and reflective middleware that supports smart cloud-based deployment and execution of engineering workflows. This middleware supports deep inspection of the workflow task structure and execution, as well as of the very specific mathematical tools, their executions and used parameters. The reflective capabilities are based on multiple meta-models to reflect workflow structure, deployment, execution and resources. Adaptive deployment is driven by both human input as meta-data annotations as well as adaptation policies that reason over the actual execution history of the workflows. We validate and evaluate this middleware in real-life application cases and scenarios in the domain of aeronautics.

1. Introduction

Engineers in major industries, such as aerospace and automotive, use simulation and optimization workflows to create, simulate and optimize complex designs. Such workflows are complex and long running processes, which are typically composed of various software tools and services, to simulate and optimize physical properties such as strength, vibrations, geometrical decomposition or material selection. Engineers use different hardware to execute these workflows, e.g., their desktop computers or High Performance Computing (HPC) clusters.

Current situation. Desktop computers have limited capacity in terms of processors, memory and storage. In addition, the parallel execution of the experiments is tied to the number of available computers. HPC clusters, unlike desktop computers, are very efficient and powerful, but they are constructed with dedicated expensive hardware and their capacity is not always directly available. Besides, time slot reservation and complex queuing API are yet another hassle for those with long-running or recurring experiments.

The promise of the cloud. Engineers can nowadays benefit from cloud computing to gain on-demand access to the required resources for their workflows, often based on cheap commodity hardware. Cloud computing is a model for enabling *on-demand* network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) [1]. Cloud orchestration tools enable automated provisioning of the required cloud-related resources such as virtual machines, virtual networks, and required infrastructure software and middleware platforms. As such, the infrastructure and deployment process become completely *composable and programmable*.

Challenges. There are still key problems and challenges when deploying and executing engineering workflows in the cloud. One needs to automate the deployment, as well as support smart scaling and execution of simulation and optimization workflows in the cloud. For each deployment and execution of the workflow, this process includes adaptive deployment to collocate, separate and parallelize the different tasks and their specific tools on the right amount and the right type of nodes. Both can also vary depending on the specific parameters for a certain execution. Automating this process includes automatic determination of the required resource types (virtual machines, storage volumes, etc), automatic estimation of the amount of cloud resources (number of virtual machines, amount of memory and cores, etc), as well as the automatic bootstrapping and destruction of the required infrastructure.

InfraComposer: towards smart deployment in the cloud. To address these challenges, we present a **reflective and adaptive middleware** that enables and manages smart, adaptive workflow deployment, scaling and execution in the cloud. We leverage both the domain-specific knowledge about the concrete tools that are used, and deep inspection of these tools when deployed and executing on the cloud platform.

* Corresponding author.
E-mail addresses: Emad.HeydariBeni@cs.kuleuven.be (E. Heydari Beni), Bert.Lagaisse@cs.kuleuven.be (B. Lagaisse), Wouter.Joosen@cs.kuleuven.be (W. Joosen).

https://doi.org/10.1016/j.sysarc.2019.03.001
Received 7 March 2018; Received in revised form 3 January 2019; Accepted 6 March 2019
Available online 9 March 2019
1383-7621/© 2019 Elsevier B.V. All rights reserved.

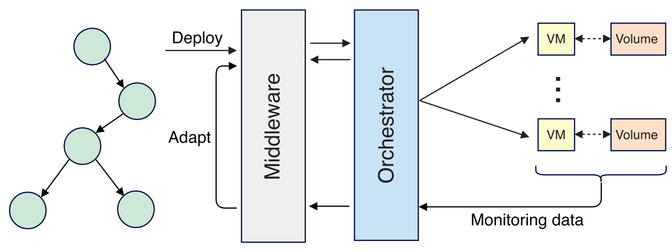


Fig. 1. Middleware for cloudifying simulation workflows.

The **adaptive middleware** is driven by both input from the engineers about the properties of the tools they use, as well as execution history of these tools. The input from the engineers is specified as annotations on the workflows, and is based on human knowledge and assumption about the tools with regards to CPU usage, memory usage and network usage. The execution history over time will be used to optimize the original deployment and scaling plan, and thus to further adapt to actual real execution knowledge (see Fig. 1).

As such, our middleware defines two key contributions to existing orchestration and deployment middleware:

1. *Annotation-driven resource reservation and deployment planning.* Based on the annotations in the workflow, an initial deployment plan will be generated. A deployment plan is a topology and orchestration specification for a cloud application which can be executed by a cloud orchestrator (see Section 3.2, 3.2).
2. *History-driven adaptive scaling and reconfiguration.* In addition, the middleware adapts the configurations of the deployment plans based on the execution history of the workflows using previous results. This is driven by policies that can reason about, and perform statistical analysis on the execution history (see Section 3.3).

To achieve this, the **reflective meta-models in the middleware** enable reification of

1. key architectural concepts such as workflows, tasks, specific tools and their deployment,
2. key execution concepts such as specific tool executions with specific parameters, and
3. key resource utilization concepts such as nodes, cores, cpu time, memory, storage and network metrics.

Moreover, we validate and evaluate the concepts with real world, industrial use cases and scenarios reflecting actual production settings (see Section 2). The main focus of this paper is to present a holistic cloudification system for engineering workflows with a special attention to the reification of various concepts. We validate the system's applicability and relevance by trying and analysing these engineering use cases in general, and in particular aeronautics.

Compared to our ARM 2017 paper [2], this paper includes about 45% new material. First we extend the middleware concepts with policy-based adaptation. Second, we describe the fine-grained architecture of the adaptive and reflective middleware. Third, we provide an extensive validation and evaluation in multiple real life application cases and adaptive scenarios.

The rest of this paper is structured as follows. Section 2 presents two motivating scenarios of engineering workflows and their common patterns. Section 3 describes the architecture and the concepts of the middleware. Section 4 validates multiple adaptive (re)deployment scenarios. Section 5 presents the state-of-the-art in cloudification of workflows, adaptive middleware, and auto-scaling techniques. Section 6 outlines the limitations and the possible opportunities to extend the current work to become smarter and more comprehensive. Section 7 concludes this paper and outlines our research outcomes.

2. Motivation and use cases

In this section, we describe two examples of industrial workflows from the aeronautics domain as motivation and validation in order to discuss adaptive scenarios.

2.1. Electrical Wiring Interconnection System (EWIS)

EWIS is one of the important steps of the multidisciplinary design optimization (MDO) in the design process of aircrafts. Aerospace engineers design and execute complex simulation and optimization experiments to find optimal solutions for a cockpit's wire harness routing.

Motivating Scenario. When considering the tasks in the workflow, it is unclear whether the main optimization tool used by one of the tasks is memory intensive or not. The most optimal deployment is achieved in a step-wise adaptive process, first driven by the engineer's annotations, then by the execution history of the tool.

(i) First, the engineer of the workflow specifies that the tool (the wire harness tool) is not CPU intensive but memory intensive because it loads a very large amount of data (i.e., physical features of an aircraft) to the memory. At least, this is the engineer's assumption. He defines this assumed knowledge as annotations.
(ii) The middleware will allocate a large amount of memory with few number of cores for each virtual node in the cloud and enables the engineer to execute the experiment as specified in the workflow.
(iii) However, execution history shows that the tool is mostly CPU intensive because the executing nodes reached the system load saturation limits, and the memory was overallocated.
(iv) The deployment plan should be updated to employ a higher number cores and less amount of memory for the virtual nodes.

2.2. Design of the hinge system of an aircraft rudder

Another multidisciplinary design optimization (MDO) example can be found in the design process of aircrafts. Optimization of a hinge design is a crucial part of an aircraft rudder design as a whole, and it is automated as a workflow. The workflow consists of a set of engineering tools, which are responsible for meshing and stress analysis of hinge components, as well as performing a quasi-exhaustive search for all different possibilities in order to minimise objectives (e.g., total weight) [3,4].

Motivating scenario. These engineering tools are interdependent, and their execution flow within the workflow is a complex design by an MDO expert. Therefore, the supported level of parallellization, as well as the number of nodes and tool instances needed are unclear to an airplane engineer.

(i) First, the engineer of the workflow specifies to use only one very large node, as well as an expected execution time. (ii) The middleware will instantiate the node and execute the experiment as specified in the workflow. (iii) However, the execution history shows that the execution took much longer than the initial anticipation because of the large number of requested experiments. This resulted in many parallel runs and scheduled jobs. (iv) The deployment plan should be updated to adjust the number of nodes (rescaling) with regards to the expected execution time.

Section 4 presents more scenarios. Based on each of these workflows, a similar pattern emerges:

- The deployment middleware needs initial domain knowledge about the tools to achieve a first deployment plan.
- Actual executions of the tool with specific parameters might require optimization of the deployment plan.
- These workflows need different discipline analysis tools for execution, and each tool could be installed on a specific operating system and specific host type (memory-focused host, CPU-focused host or high-performing storage hosts with SSDs).

- These workflows are often computationally intensive, and their execution sometimes takes hours, days or weeks to be completed. Engineers use parallel runs to speed up the execution. That may have impact on network topology.
- These workflows are used in a continuous improvement process by reconfiguration of the design parameters, and recurring re-executions to achieve the optimized objectives. As such, the actual parameters of the executions might require adaptation of the deployment as tools might become more dependent on CPU than disk for different parameters.

These common patterns introduce several key problems and challenges (refer to Section 1) leading to manual, duplicate, complex, time-consuming work for engineers.

3. The infracomposer middleware

This section presents the architecture of InfraComposer, focussing on the different features and subsystems of the policy-driven reflective and adaptive middleware for the cloudification of engineering workflows.

First, the middleware supports annotation-driven, cloud-based deployment of engineering workflows and their different subtasks. During execution, it collects runtime information about task executions and the underpinning infrastructure by reflective monitoring of resources, as well as deep inspection of software tools. Adaptation policies, which are based on the execution history, enable the middleware to reconfigure the deployment plans predictively to be adaptive for recurrent execution of the workflows.

The InfraComposer middleware architecture consists of four main components (see Fig. 2): (i) a *workflow manager* component to expose a workflow deployment API and to identify the annotated tasks and their annotations, (ii) a *configurator* component to generate configurations based on given annotations with respect to execution history data, (iii) a *deployment plan composer* component to produce deployment plans based on elementary deployment modules for the cloud orchestrator and to initiate the deployment, and (iv) a *monitoring* component to store live monitoring data of workflow execution.

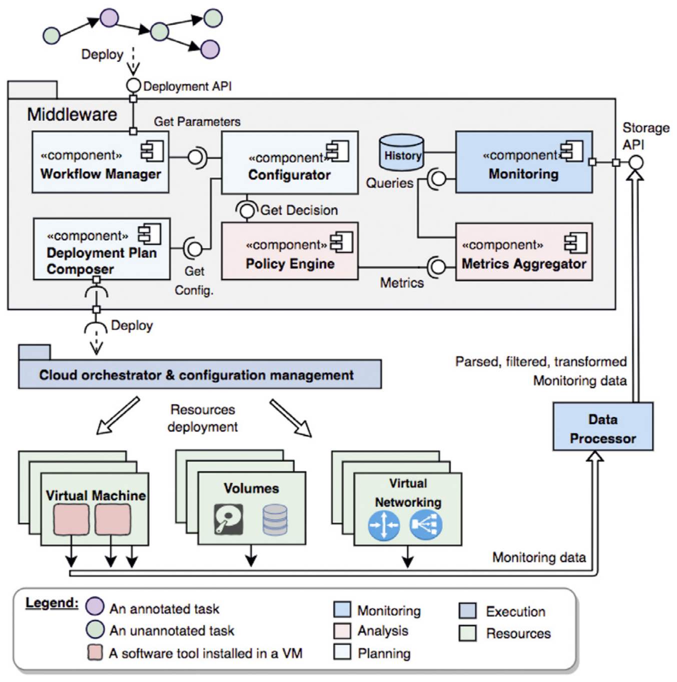


Fig. 2. Overview of the proposed middleware with the annotation processing and configuration components.

The rest of this section is structured as follows. First, we describe the annotation-based deployment. We then elaborate on the reflective capabilities and the meta-models. Third, we describe the policy-driven adaptation architecture.

3.1. Annotation-based deployment

A simulation and optimization workflow is a group of tasks that, once completed, will accomplish some objectives. As explained in Section 1, these tasks employ different analysis tools, which are responsible for the execution. Workflows and tasks can be annotated to provide more information about the required resources. InfraComposer is capable of identifying these annotations in the *workflow manager* component to provide necessary information for the *configurator* component. There are two categories of annotations: direct-deployment and resource-consumption annotations.

Direct-deployment annotations. These annotations provide the middleware with direct information concerning deployment of workflows on the cloud. The crucial aspects described by annotations are the employed analysis tools, their deployment locations and number of instances. Tools can either be collocated or separately deployed on the nodes. For example, annotations could describe that some tools should be deployed on one node, others on individual nodes, and there should be five instances of each node. Furthermore, network-related annotations can propose a networking scheme for tools and nodes where necessary. For example, network-level segregation of nodes can be achieved for a network intensive workflow.

Another direct-deployment annotation is the identifier of the existing resources (e.g., instance images, volumes, networking components, etc.). For instance, some of the engineering tools need human involvement during the installation process, or install slowly due to the size of the packages. Therefore, these tools can be installed and prepared as virtual machine images to speed up the deployment process. The unique identifiers of the images help the *configurator* component provide configurations to the *deployment plan composer* component.

Resource-consumption annotations. These annotations provide general, approximate information about the infrastructural resource requirements of the tools with regard to disk, memory, processor and network. The four main categories of resource-consumption annotations are presented in Table 1.

Workflows can specify whether the experiment is disk intensive, as well as the required space. Cloud providers, either private or public, offer various types of storage systems with varying capabilities, speed, etc. In addition, some clients are concerned about data locality due to the enterprise policies or governmental law (e.g., GDPR[5]). Such annotations guide InfraComposer to select appropriate storage options with respect to the given constraints.

The computationally intensive workflows should employ proper virtual machines in order to execute efficiently and to minimise the interference with other co-existing cloud users. Some virtual machines share the physical processors with other tenants, and some have

Table 1
Four main categories of resource-consumption annotations.

Category	Annotation	
Disk	Disk intensive	percentage
	Required disk space	GB
	Data locality	location
Memory	Memory intensive	percentage
	Required RAM	GiB
Processor	CPU intensive	percentage
	GPU intensive	percentage
	Required cores	number of cores
Network	Network intensive	percentage
	Required bandwidth	Mbps

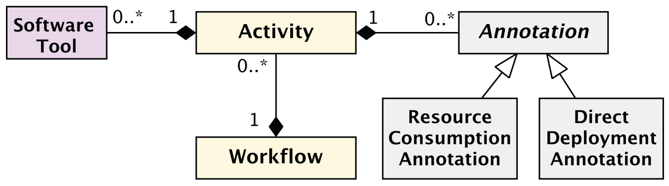


Fig. 3. The structural meta-model.

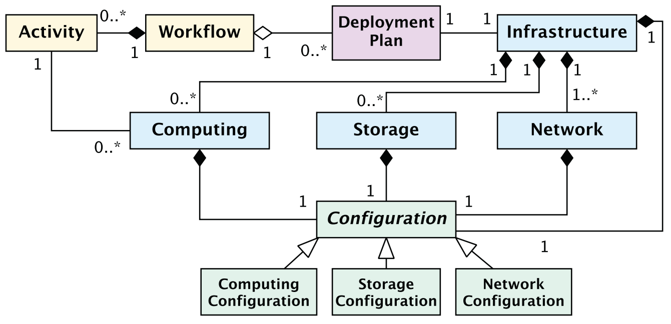


Fig. 4. The deployment meta-model.

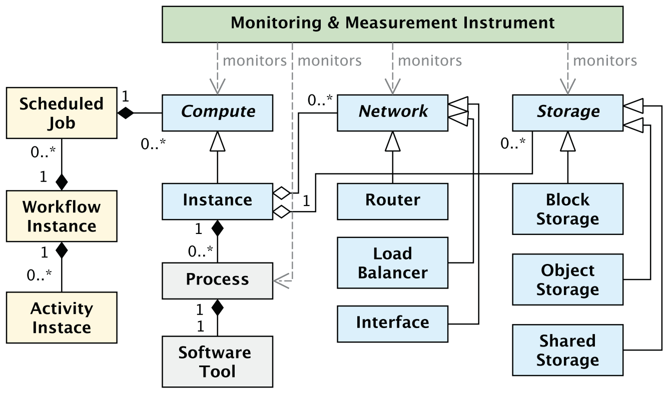


Fig. 5. The execution meta-model.

CPU-pinning, meaning that the virtual cores are mapped to the physical cores in a shared-nothing approach. Besides, some public cloud providers offer domain-specific types of virtual machines such as accelerated instances with GPU.

Furthermore, parallel execution of workflows may have considerable network overhead due to the continuous transfer of large chunks of data. That can easily saturate the bandwidth, slow down the execution, and interfere with other co-existing users. Network annotations enable the middleware to compose appropriate network architectures based on the available networking infrastructure.

3.2. Reflective capabilities and meta-models

The middleware follows the temporal correspondence approach into different meta-models [6]. There are four styles of reflection in InfraComposer represented by four meta-models, namely structural, deployment, execution and resource reflection.

Structural reflection. Structural reflection [7] results in a meta-model of the different static concepts in the workflows defined by the engineers, which represents the structure of workflows and tools within the middleware and the execution history. Fig. 3 illustrates the meta-model. This meta-model describes engineering workflows and composition of activities and tools, as well as annotations.

Deployment reflection. Deployment reflection results in a meta-model representing and reifying the concepts in the deployment model. Fig. 4 illustrates the deployment plan and the mappings between workflows, activities, tools (not illustrated), and cloud-based components such as compute nodes, storage and networking elements.

Execution reflection. Execution reflection results in a meta-model of the execution of activities and specific tools on specific nodes. This model reifies concepts such as execution per workflow, execution per activity, and execution per tool with respect to the engineer's given design parameters (see Fig. 5).

Resource reflection. Resource reflection results in a meta-model of the underpinning cloud infrastructure resources and domain resources (see Fig. 6). Cloud infrastructure resources include concepts such as processing (cores), compute nodes, memory, network and storage, which are reified for each execution by consumption pattern. Such reflection allows the monitoring and adaptation phase to benefit from coarse-grained or deep introspection of resources, leading to more efficient resource allocation in the future execution of the workflows.

3.3. Policy-driven adaptive architecture

InfraComposer monitors the execution of the workflows and builds an execution history, with which it fine-tunes the deployment plan and the configurations to make the future executions more efficient. Adding intelligence to the smart adaptation capabilities of the middleware requires the acquisition of new knowledge about the execution of the different tasks and tools. For example, an annotation suggested that a tool was disk intensive, but the execution history teaches us that it is CPU intensive and only uses two cores for input files smaller than 100 MB.

A recent survey [8] classifies the architectural patterns of self-adaptive autoscaling systems into three groups: feedback loop, observe-decide-act and MAPE(-K).

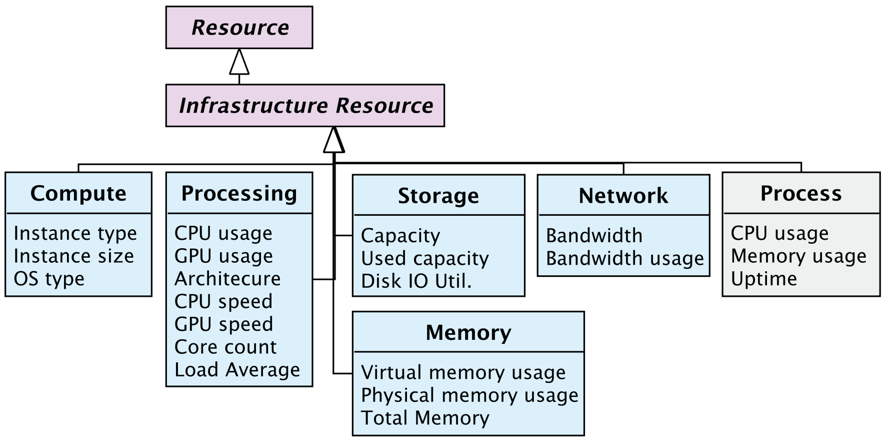


Fig. 6. The resource meta-model.

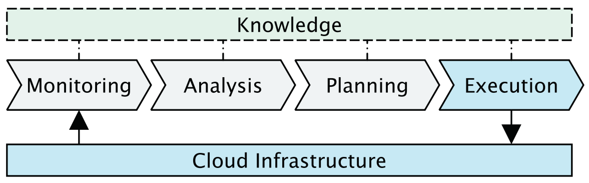


Fig. 7. Adaptive middleware using MAPE-K control loop.

InfraComposer is a policy-driven (self-)adaptive middleware following the MAPE-K [9] architecture of self-adaptive systems [10] (see Fig. 7).

Among different architectural patterns and decision making approaches, we employed MAPE-K on account of better separation of concerns. Furthermore, we opted for a *rule-based control loop* because of negligible overhead and manageable policy reconfigurations at runtime.

As illustrated in Fig. 2, the *monitoring* phase collects information from the cloud and workflow-specific resources. The data processor component parses, filters, and transforms the monitoring data and persists it in a TSDB. The *analysis* phase uses the persisted monitoring data by performing statistical aggregate queries (e.g., max, min, average, percentile, etc.), and it employs a policy engine to enforce adaptation policies to assess the previous executions and to determine potential future improvements. The *planning* phase reconfigures the deployment plans, using the history-driven scaling propositions produced in the previous phase. The *execution* phase redeploys the cloud resources and the software tools for another execution. The *knowledge* about the workflows, and the cloud infrastructure (i.e., public/private providers, available types of resources, the existing resources, etc.) is a cross-cutting aspect serving information to each phase.

In the InfraComposer adaptive architecture, the analysis and the planning phase are crucial. We employed a reactive [11] policy-based approach to adjust the deployment plans and resource allocation based on certain thresholds provided in policies. For example, a processing metric driven by statistics based on the execution history exceeds a particular threshold defined in the rules. This shows that the activity is a computationally intensive task. Then the configurator could reconfigure the deployment plan for that node to employ CPU-focused virtual machines.

Adaptation policies consist of rescaling rules. Each rule is composed of zero or more conditions, as well as a number of actions as a consequence. As depicted in Fig. 2, the metrics aggregator component executes complex statistical queries on time series data gathered during workflow executions, and it provides the aggregate data to the policy engine component through its interface. The scope of these reflective data includes cluster-wide metrics regarding the nodes (`Cluster`), the network (`Net`) and the storage systems (`Storage`), together with virtual-node related metrics concerning an average executing node (`VM`) and the coordinator node (`Master`). For instance, a condition can be `Net.bandwidth_utilization<1Gbps`. When a condition is satisfied, the policy engine performs rescaling actions (e.g. `resize`, `add`), which provides re-configuration hints to the configurator component. For instance, the resize action can be about nodes (e.g. number of cores, or the amount of memory), or it can be cluster-wide (e.g. number of nodes).

3.4. Monitoring data management

We first present more details about the data management components, and then we describe the scalability concerns and the existing possible solutions.

3.4.1. Monitoring data management components

The monitoring data management of InfraComposer consists of (i) a set of telemetry data collectors (probes or agents) to inspect processes and system-wide metrics, (ii) data processing pipelines to parse incoming data streams, filter out irrelevant (or erroneous) entries, and transform them to the required format, (iii) a time-series database (TSDB) to store the execution history and (iv) a metric aggregator component including a data visualization layer to obtain aggregate inquiries, and eventually make them available for the policy engine.

For example, when a workflow engine dispatches jobs across a cluster of nodes, log entries regarding job events (e.g., started, completed, failed, etc.) are streamed through the monitoring components and stored in the TSDB. The middleware is now capable of observing job count, job executions, throughput, and so forth. An example query can be the average percentage of maximum IO-wait during periodic time windows of 30 seconds for a specific python process, running on all worker nodes, between the start and the end of the previous round of the workflow execution.

3.4.2. Scalability of the stack

The data processor component is perceived to be scalable depending on the monitoring stack. Telemetry data collectors (monitoring probes) should perform their tasks independently on worker nodes. The Data Processing pipeline can scale out horizontally since its functionalities are stateless and limited to parsing, filtering and transforming the input and eventually forwarding the stream to the time series database. Lastly, the chosen TSDB (e.g., InfluxDB, Elasticsearch, etc.) should be horizontally scalable. Recently, Jensen et al. [12] carried out a comprehensive survey about different angles of such databases including their architectural patterns.

Typically the data processing pipeline (e.g., Logstash) is the weakest link and it is likely to be CPU and network intensive [13]. More specifically, the processing functionalities include numerous CPU intensive executions of regular expressions through filtering plugins (e.g., Grok). In addition, it is network intensive, in the sense that if the input rate exceeds the maximum limit of the processing pipeline instance capacity, it will then throttle itself.

The first remedy to alleviate the workload is to perform preprocessing at the data collectors side. For example, data collectors should execute regular expressions, detect and filter unnecessary logs; in other words, they should lift the workload of the data processing pipeline before shipping the logs. Depending on the volume of workload, preprocessing is not always sufficient; in fact, the pipeline may again throttle itself against large-scale settings. The second additional solution is to employ messaging queues [13]. In this scenario, the processing pipeline instances need to get scaled out horizontally and follow a pull model against the queue (e.g., a Kafka topic).

4. Prototype implementation and use case validation

In this section we assess and validate our adaptive and reflective middleware for the cloudification of simulation workflows based on the use cases defined in Section 2. More specifically, we demonstrate a set of particular adaptive deployment scenarios and validate how both the reflective capabilities as well as the adaptation capabilities of the middleware can cope with each adaptive deployment scenario. As a proof of concept, we implemented the concepts and architecture of InfraComposer in our prototype middleware and developed our two use cases (i.e., (i) the EWIS design and (ii) the design of the hinge system of an aircraft rudder (see Section 2)) as two simulation workflows that leverage the production workflow engine and actual simulation tools of the companies.

Simulation and optimization workflows, to achieve optimal objectives, go through an iterative execution of experiments. Each run of the experiments comes with different input design variables which are provided by the workflow engine at runtime. Input design variables are most of the time slightly different compared to the previous rounds. The behavior of a run is most likely predictable and more and less the same as the previous round. We can not be certain that these workflows

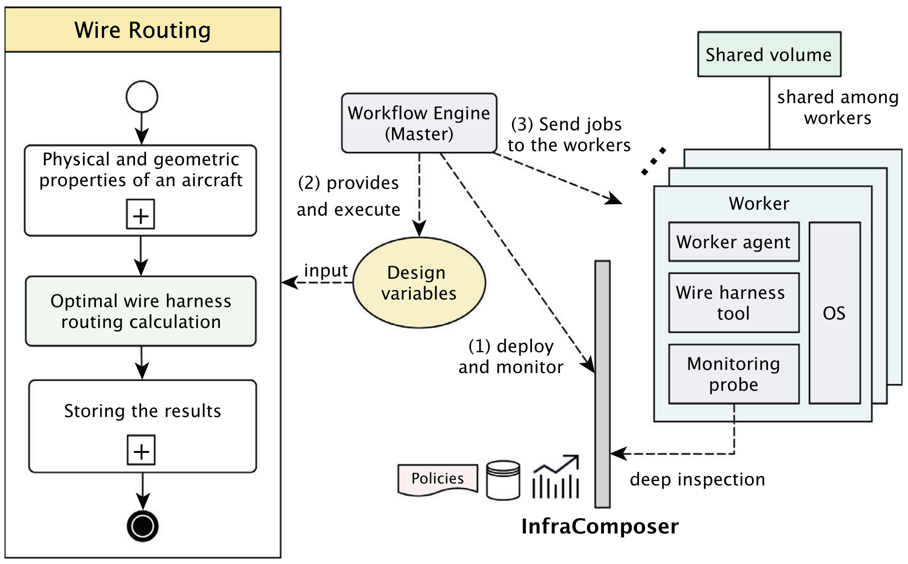


Fig. 8. Overview of the workflow of wire harness routing simulation and optimization in the cloud. The workflow engine (master node) and the worker nodes are deployed in virtual machines.

are insensitive to the input data, however based on our observations, the behavior change is not very extreme. Therefore, in this paper we assumed that every iteration of a workflow exhibits roughly the same behavior.

Moreover, we used OpenStack as a state-of-the-art cloud platform. Apart from the industrial domain tools in the use cases, the middleware is written in Java using the Spring framework, and the other technologies involved in each MAPE-K phase were: (1) Elastic Stack [13] as monitoring stack in the *monitoring phase*, including a set of telemetry data collectors (Filebeat and Metricbeat), a data processing pipeline (Logstash), a time series database (Elasticsearch) and a data visualisation layer (Kibana), (2) Drools [14] as rule engine in the *analysis phase*, (3) TOSCA [15] as orchestration and topology specification in the *planning phase*, and (4) Cloudify [16] as cloud orchestration and configuration management system in the *execution phase*.

In our validations, InfraComposer performs reconfigurations and adaptations through rescaling of cloud resources. We employed a policy-based approach for the analysis and the planning phase. We consider two styles of rescaling:

- *Vertical rescaling* supports to add more (or remove) resources to a single node in a system [17]. The middleware reconfigures resources to become larger or smaller resources. For example, a virtual machine with 2 CPU cores and 4 GB of RAM is reconfigured to 8 CPU cores and 8 GB of RAM.
- *Horizontal rescaling* supports to add more (or remove) nodes to a distributed system [17]. The middleware reconfigures the deployment plans to duplicate the nodes. For example, one experiment executes over 16 virtual machines instead of 4 virtual machines.

In the following sections, we revisit the use cases and validate the adaptation scenarios and monitoring results leveraging our adaptive and reflective middleware.

4.1. Electrical wiring interconnection system design

Following 3 different iterations (steps) in the MAPE-K loop, we performed some experiments presenting a number of adaptation scenarios. We assume that an engineer has made a set of inappropriate assumptions about the workflow and its requirements in each scenario. Each step is an iteration of the MAPE-K loop.

4.1.1. Scenario 1: The workflow is memory intensive

The engineer annotates the wire harness tool as memory intensive, but the reflective monitoring shows that the tool does not load a very large chunk of data to the memory before and during the workflow execution. Consequently the deployment plan is altered to scale down and to employ virtual machines with a lower amount of memory.

In step 1, the workflow engineer specifies both direct deployment and resource consumption annotations. The former includes the use of the wire harness optimization tool, a single installation per node, a total of five worker nodes, and a shared storage for geometrical data. The latter describes the workflow as *memory* and network intensive. The workflow is then sent to InfraComposer to deploy the cloud infrastructure. As depicted in Fig. 8, the execution then starts and the workflow engine (the master node) sends optimization jobs to the worker nodes. Given the above configuration, the total execution took 62.37min (see Table 2).

The engineer then wants to re-execute the workflow with more experiments. InfraComposer monitored the execution of step 1, and it

Table 2
Configuration details of each execution round of the EWIS workflow. `Step1` settings are provided by the engineer, but the other steps are reconfigured by InfraComposer based on the adaptation scenarios and policies. Each execution step is an iteration of the MAPE-K look which contains 45 experiments. Each node can take a job (optimal wire harness routing calculation) at a time in this use case. The number of experiments is specific domain knowledge, and InfraComposer observes the 45 experiments as one large experiment.

	vCPU	RAM (GB)	Nodes	Intensiveness	Execution	Jobs per VM
Step 1	2	16	5	Memory Network	62.37 min	9
Step 2	4	8	10	CPU Network	33.10 min	~5
Step 3	4	4	15	CPU Network	22.02 min	3

Table 3
Policy-based adaptation scenarios based on execution history. The system performance analysis is performed using the Utilization Saturation and Errors (USE) [18] methodology. Aggregated values are maximum values, and `VMs.load_average` maps to the system load during the last one-minute periods in Linux.

S	Adaptations	Policies
1	– Assumption: memory intensive – Adaptation: scale up/down the VM	**input**: VM **if** *VM.free_memory* > 4*GB* *and VM.total_memory* >= 4*GB* **then** | resize(VM, VM.total_memory/2) **if** *VM.swap_memory* > 0 **then** | resize(VM, VM.total_memory*2)
2	– Assumption: CPU intensive – Adaptation: scale up/down the VM	**input**: VM **if** *VM.load_average* < *VM.cores* *and VM.CPU_utilization* < 40% *and VM.cores* >= 4 **then** | resize(VM, VM.cores/2) **if** *VM.load_average* > *VM.cores* *or VM.CPU_utilization* > 60% **then** | resize(VM, VM.cores*2)
3	– Assumption: few number of nodes – Adaptation: scale in/out the VMs	**input**: Cluster, VM, Workflow, Net, Master, license, quota **if** *(Workflow.execution* > 30*min* *or VM.accepted jobs* > 1) *and Net.bandwidth_utilization* < 1*Gbps* *and Net.packet_loss* < 5% *and is_sufficient (license, quota)* *and not master_node_saturated(Master)* **then** | quantity ← license.tools < 5 ? license.tools : 5 | resize(Cluster, Cluster.VMs.count + quantity) **if** *Net.bandwidth_utilization* >= 1*Gbps* *or Net.packet_loss* >= 5% *or master_node_saturated(Master)* **then** | resize(Cluster, Cluster.VMs.count - 5)

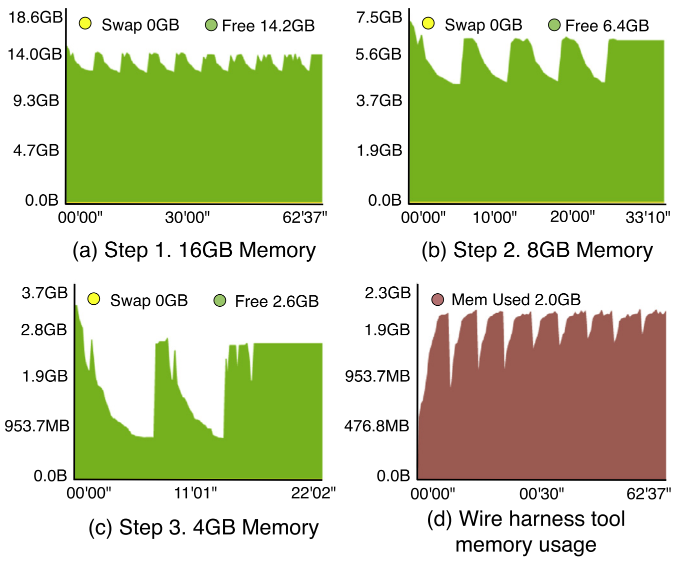


Fig. 9. Different steps of the EWIS case: free memory vs. swapping.

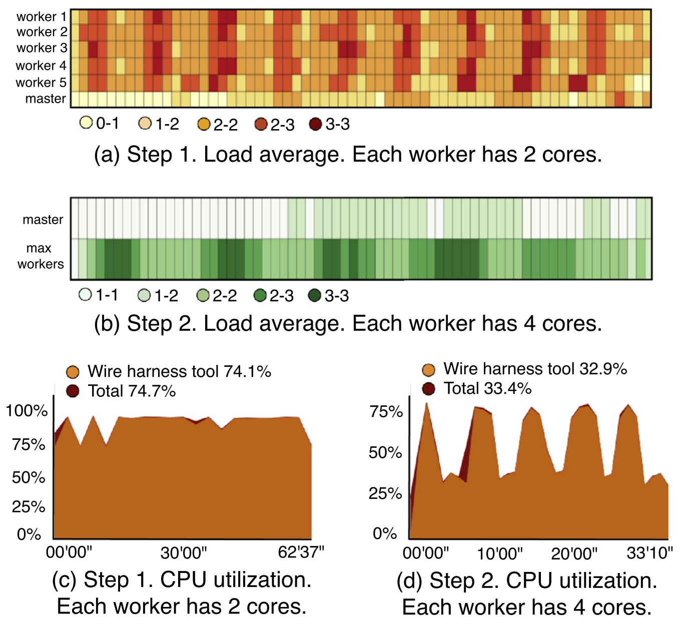


Fig. 10. CPU utilization and load average of the workers and the master node. Master node has 4 cores. Each core is a virtual core.

persisted the monitoring data after deep inspection of different processes, jobs, and system-wide metrics.

The monitoring component aggregates memory-related metrics (e.g., free, swap memory, etc.), and it transfers the results to the policy engine in order to enforce the business rules (see Table 3). As shown in Fig. 9, a large portion of the memory remained unused in step 1. Therefore, the virtual machine has been resized to employ less memory in order to avoid over provisioning.

4.1.2. Scenario 2: The workflow is not CPU intensive

The workflow (i.e., the wire harness optimization task in particular) is not annotated to be compute-bound, but the reflective monitoring data shows that the nodes and the tools utilize more than a particular threshold. More specifically, the virtual machine load average depicts system saturation by having a load number higher than the number of cores. Therefore, the deployment plan is updated to scale up and employ more cores.

To re-execute the workflow after step 1, InfraComposer enforces some compute-related rules using its policy engine (see scenario 2 in Table 3). Fig. 10 illustrates the CPU utilization as well as the system load during the last one-minute periods (Linux load average 1min). The latter indicates system saturation and it should normally be less than the number of cores. In step 1, the virtual machine employed 2 cores and accordingly the system is saturated. Therefore the workflow is reconfigured to employ 4 cores instead, both for step 2 and step 3. As a result, the system load remained below 4 (the number of cores).

Table 4
Configuration details of each execution round of the hinge system design and optimization workflow. `Step1` settings are provided by the engineer, but the other steps are reconfigured by InfraComposer based on the adaptation scenarios and policies. Each execution step contains 45 experiments.

	vCPU	RAM (GB)	Nodes	Intensiveness	Execution	Jobs per VM
Step 1	8	16	1	Memory CPU	109.20 min	315
Step 2	4	8	5	Disk	22.23 min	63
Step 3	2	4	10	Disk	12.12 min	~ 32
Step 4	2	4	14	Disk	9.75 min	~ 23

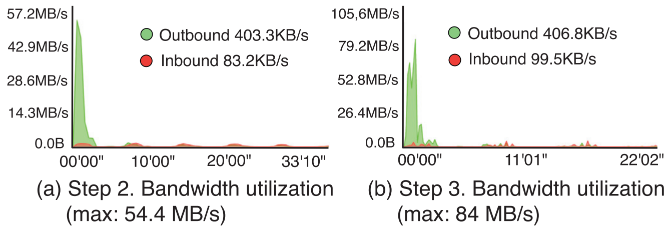


Fig. 11. Cluster-wide bandwidth utilization of the network where the shared volume is operating.

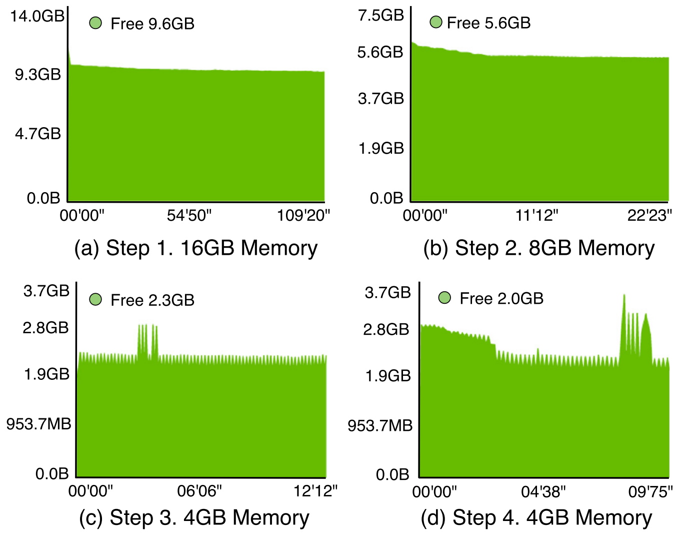


Fig. 12. The Hinge System case: average free memory vs. memory swapping in different steps.

4.1.3. *Scenario 3: The workflow needs few number of nodes*

The engineer anticipates that the horizontal scale of the wire harness virtual machines should be limited. He has to comply with his quota of total number of cores and memory in the distributed setup, and initially wants it to be deployed cheaply on a few number of instances. However, the reflective monitoring shows that there are a considerable number of parallel runs (i.e., scheduled jobs) due to the large size of input parameters and datasets (about the physical features of the aircraft). This results in a long-running overall execution. Consequently, the deployment plan is reconfigured to scale out the nodes in number and scale them down in memory size. As such the workflow can employ more instances of the tools, and can thus satisfy the expected execution time requirement while respecting quota and budget.

Policies in Table 3 for scenario 3 present horizontal re-scaling constraints as rules. In addition to execution time and number of accepted jobs per VM (9 jobs per VM in step 1), networking metrics, licensing and quota limitations are important too. Cluster-wide bandwidth utilization has an actual upper limit, and reaching that limit causes network saturation in terms of packet loss, networking errors and segments retransmissions. Furthermore, there should be available resources in the master node due to the execution of the workflow and the job scheduling. Policies make sure that the master node is not saturated in terms of average load, available memory, and disk IO rates.

Fig. 11 illustrates the cluster-wide bandwidth utilization of the network where the shared volume is operating. The maximum inbound rate (34.9 MB/s in step 1 (not shown), 54.5 MB/s in step 2 and 84 MB/s in step 3) is lower than the overall bandwidth of the network, and therefore packet loss was negligible. In addition, disk IO utilization of the nodes were 52.4% in step 1, 74.82% in step 2 and 70.05% in step 3. The network latency of the shared volume has impact on these values. Therefore the policy engine scaled out and reconfigured the number of nodes to 10 in step 2 and 15 in step 3, and accordingly the execution time was 29.27min faster in step 2 and 40.25min faster in step 3 in comparison with step 1.

4.2. *Design of the hinge system of an aircraft rudder*

This workflow employs seven engineering tools. These tools are responsible for meshing and stress analysis of different hinge components. Similar to the previous use case to obtain an optimized result (see Section 4.1), the workflow goes through iterative executions with different input design variables.

The overview of the different experiments and results results are listed in Table 4. In step 1, the tools are annotated to be collocated on a single, large, powerful node, and the entire workflow is presumed memory and CPU intensive.

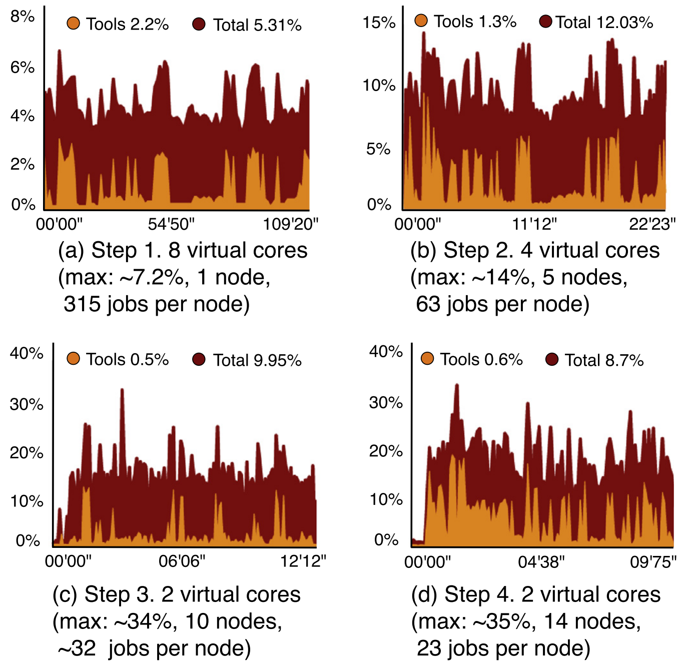
4.2.1. *Scenario 1: The workflow is memory intensive*

The engineer annotates the entire workflow as memory intensive, but the reflective monitoring shows overprovisioning of resources. As illustrated in Fig. 12, the policy engine of InfraComposer scales down the deployment plan from 16 GB memory in step 1 to 4 GB memory in Step 3. As a result, the execution has a smaller yet more cost-effective footprint. Policies are listed in scenario 1 of Table 3.

4.2.2. *Scenario 2: The workflow is CPU intensive*

The engineer annotates the workflow to be compute-bound, but the reflective monitoring shows that the compute resources are underutilized. In step 1, the deployment plan is set to employ 8 virtual cores, which results in maximum utilization of 7.2% for 315 sequential jobs (see Fig. 13). For re-execution of the workflow, the policies of the second scenario (see Table 3) trigger the InfraComposer configurator to scale down from 8 cores in step 1 to 2 cores in step 3. However, the CPU still remains unsaturated[1]

[1] Since the workflow has Windows-based worker nodes, there was no load average metric in place. The OS of the master node is Linux for both use cases.



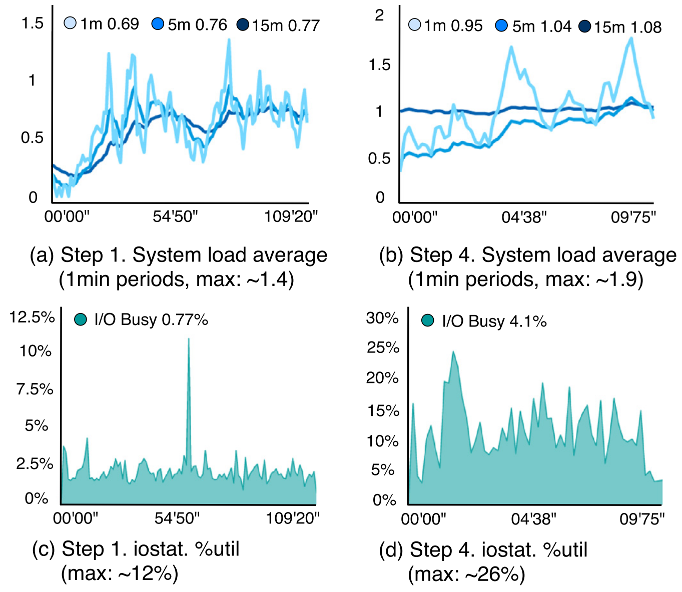
(a) Step 1. 8 virtual cores (max: ~7.2%, 1 node, 315 jobs per node)

(b) Step 2. 4 virtual cores (max: ~14%, 5 nodes, 63 jobs per node)

(c) Step 3. 2 virtual cores (max: ~34%, 10 nodes, ~32 jobs per node)

(d) Step 4. 2 virtual cores (max: ~35%, 14 nodes, 23 jobs per node)

Fig. 13. Maximum CPU utilization of the VMs and the tools.



(a) Step 1. System load average (1min periods, max: ~1.4)

(b) Step 4. System load average (1min periods, max: ~1.9)

(c) Step 1. iostat. %util (max: ~12%)

(d) Step 4. iostat. %util (max: ~26%)

Fig. 14. System load average and percentage of Disk I/O utilization of the master node which has 4 vCPUs and 8 GB RAM.

4.2.3. *Scenario 3: The workflow needs few number of nodes*

The engineer anticipates that a larger virtual machine is more efficient than a higher quantity of nodes. That results in an undesirable execution time of 109.20min. Beside other metrics, deep inspection of the workflow execution shows that this node received 315 jobs. Therefore, InfraComposer reconfigures the deployment plan to scale out horizontally and yet stay compliant with quota limitations when the engineer intends to re-execute the workflow.

As listed in Table 3, horizontal resource rescaling is constrained by cluster-wide metrics (e.g., overall bandwidth utilization) and master-node metrics (e.g., load average, free memory and disk I/O utilization). The network bandwidth utilization of the cluster has an acceptable rate of 7.8 KB/s in step 1, 39.1 KB/s in step 2, 293 KB/s in step 3, and 100 KB/s in step 4. As illustrated in Fig. 14, (a) and (b) represent the system load average during the last one-minute periods of the master node (i.e., with a maximum value of 1.4 in step 1 and 1.9 in step 4). Therefore, the master node remained unsaturated while managing 14 worker nodes. Furthermore, the master node never experienced memory paging, and it had free memory in all of the steps (i.e., with a maximum value of 6.4 GB in step 1, 6.1 GB in step 2, 5.8 GB in step 3 and 5.9 GB in step 4). Lastly, the Linux manual [19] defines disk I/O utilization as “percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%”. As depicted in Fig. 14, the percentage is around 12% in step 1 and 26% in step 4 at peak load moments.

As a result, the policy engine triggers the configurator component to resize the number of nodes to 5 in step 2, 10 in step 3 and 14 in step 4, and accordingly the execution time was reduced with 86.97min in step 2, 97.08min in step 3, and 99.45min in step 4. The distributed setup thus indeed scales well horizontally and the workflow executes much faster in comparison with step 1.

5. Related work

5.1. *Execution environment reproducibility*

Reproducibility and repeatability of workflow execution environment are crucial aspects in scientific and engineering workflow deployment. In that regard, Santana-Perez et al. [20] describe the execution environment of workflows using semantic vocabularies to produce annotated workflows (i.e., logical preservation of execution environment). TOSCA [15] is an OASIS standard to describe the topology of cloud-based applications towards portable, reproducible application deployments. Qasha et al. [21] combine two execution-environment reproducibility techniques (i.e., the logical and physical preservation) of scientific workflows using TOSCA in a container-based approach. In addition to the plain reproducibility concerns, our middleware architecture employs reflection concepts to reconfigure deployment plans, resulting in *efficient* execution environments.

5.2. *Reflective middleware*

The concept of reflection was first introduced by Smith [22] in programming languages. Other works [23,24] adopted the concept in the middleware platforms, focusing on reconfigurability and openness of such systems. Blair et al. [23] present two types of reflection: *structural reflection* and *behavioural reflection*. Structural reflection is concerned with the structure and the content of the component, which is represented by two distinct meta-models, namely the encapsulation and composition meta-models. Behavioural reflection is concerned with activity in the system, which is represented by the environmental meta-model. Existing literature [25] extends the architecture by presenting the resource meta-model to address the reification of resource management. Weyns et al. [26] present a comprehensive reference model for distributed self-adaptive systems with a special attention to the reflection perspective. As applications of the reflection concepts in web services, [27,28] present adaptive and reflective middleware systems which are able to expose their functionalities to application developers. Our meta-models are inspired by these studies.

5.3. *Auto-scaling of cloud resources*

Deployment plans are reconfigured by rescaling of resources based on execution history, either horizontally or vertically. A recent survey by Chen et al. [8] classifies the decision making tactics of the self-adaptive cloud autoscaling systems into three major approaches: (i) rule-based control, (ii) control theoretic approaches and (iii) search-based optimisation. Alternatively, Lorido et al. [11] categorise auto-scaling techniques generally into *reactive* (i.e., based on rules and current data) and *proactive* (i.e., based on prediction) approaches, as well as a more fine-grained classification, resulting in: (1) threshold based rules, (2) reinforcement learning, (3) queuing theory, (4) control theory, and (5) time series analysis. Recently, Dhuraibi et al. [29] conducted a comprehensive survey on elasticity in cloud computing with a special attention to containers. InfraComposer employed a policy-based, history-driven approach akin to the threshold-based approach. Thresholds are statically defined, similar to the other existing literature [30–33]. Most related work [31,32] use single or multiple metrics, but Hasan et al. [34] employed several metrics from several domains such as compute, storage and network. InfraComposer also forms its policies based on multiple domains varying from workflow primitives to various cloud primitives.

5.4. Scientific workflows in the cloud

Among open challenges of migration and execution of scientific workflows on the cloud [35], computation and data management are crucial. Processing large scientific data has impact on execution mechanism of the workflow engines. Kacsuk et al. [36] present efficient data pipelines by using a service choreography concept instead of the enactor-based workflow concept. Furthermore, data locality has influence on performance and overall execution time [37,38]. Regarding adaptive execution of the workflow in the cloud, Oliveira et al. [39] introduce an adaptive approach to dynamically tuning the workflow activity size to achieve better performance, and Wang et al. [40] present an adaptive workflow management through dynamic iterative optimisation framework. Although engineering workflows are inherently different in comparison to scientific workflows, some of the challenges regarding data management and adaptive execution share common requirements and concerns.

6. Limitations and future opportunities

In this section, we outline our design decisions and future research opportunities regarding (i) the granularity of componentization, and (ii) the policy-based decision making mechanism in the context of engineering workflow cloudification.

6.1. Granularity of Componentization

InfraComposer is presented with virtual machines as the granularity of componentization; however, the concepts presented in the architecture are not necessarily tied to any specific component model (e.g., virtual machines or containers). Our future work includes applying the concepts presented in this paper to a containerised environment, which requires few considerations: (i) the reflective architecture and in particular the meta models (e.g., the resource meta model) should be adjusted to the new settings. (ii) The current architecture relies on a cloud orchestrator and a unified, modular, reusable deployment topology and orchestration specification such as TOSCA. Since mapping TOSCA to containers is not fully realised [41], there is a need for such an abstract specification enabling the integration with the state of practice container orchestration systems (e.g., Kubernetes, Mesos, Docker Swarm, etc.). (iii) Engineering workflows employ diverse set of domain tools executable on various operating systems; therefore, containers should be fully supported in those operating systems. (iv) Lastly a new autoscaling mechanism should be proposed in this context.

6.2. Dynamic Adaptation Policies

The current InfraComposer architecture is based on a policy-based approach to define rules and in particular their thresholds (see Section 3.3, 4). This approach heavily relies on the domain knowledge of engineering workflows in order to prepare the policies, and on top of that, thresholds are statically defined. To dynamize the decision making machanism, Chen et al. [8] classifies the state of the art into (i) control theoretic approaches and (ii) search-based optimisation. The control theoretic approaches (e.g., Kalman control, Fuzzy control or PD) are well studied; however, they fall short of handling multi-objectivity (e.g., execution time, cost, etc.) and performing well where many rescaling decisions are supposed to be made.

On the contrary, search-based optimisation approaches such as reinforcement learning are promising based on the recent surveys [8,11]. In this context, the complex nature of the engineering workflows cloudification and the vast optimisation space makes these approaches candidate to approach the problem. The optimisation space spans from single application configurations to co-location of services on nodes, processing components (VM or container), storage, networking, and so on. These tactics are appropriate because of well-studied related work in the domain of auto-scaling with regard to multi-objectivity through weighted sum, Pareto relation, and so forth, which is a requirement in our context.

Cloudification of engineering workflows requires a hybrid-level of control granularity (e.g., cloud resources and application level) entailing a large number of cloud primitives to monitor and tune in the adaptation process. Selecting relevant features to tune for the running workflow at hand is not an easy and trivial task to do manually. Therefore, an automatic detection of distinguishing abstract features in the execution history is an important improvement in this process (e.g., Chen et al. [42]; vPerfGuard [43]).

In summary, we envision the roadmap of a smarter deployment and execution of engineering workflows on the cloud within the MAPE-K control loop, which encompasses (i) exploring different granularity of componentization including containers and VMs; (ii) employing a smart, dynamic decision making approach such as search-based optimisation; (iii) detecting distinguishing features for the problem (workflow) at hand and their correlations with the given objectives; and lastly (iv) evaluating and validating the outcome in synthetic and most importantly real world cases and scenarios such as what has been achieved in this work regarding the policy-based approaches.

7. Conclusions

In this paper, we introduced InfraComposer, a policy-driven, adaptive and reflective middleware, which enables simulation and optimization workflows to achieve smart and optimized deployment on cloud infrastructures. Our step-wise approach includes: (i) obtaining the engineers' input about initial, direct deployment of workflows through annotations in the first place, and (ii) the acquisition of new knowledge based on the actual execution history employed to produce improved deployments. Policies encapsulate knowledge of cloud engineers and system administrators to optimize the deployments based on execution histories. Such policies reason about a time series of reflective data and act upon it by reconfiguring and resizing the execution environment for next iterations of the engineering workflow. As a validation and evaluation, we presented specific adaptive deployment scenarios in real-life application cases in the domain of aeronautics. We validated how both the reflective and the adaptation capabilities of the middleware can cope with each scenario.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.sysarc.2019.03.001.

References

[1] P. Mell, T. Grance, The NIST definition of cloud computing, 2011.
[2] E.H. Beni, B. Lagaisse, W. Joosen, Adaptive and reflective middleware for the cloudification of simulation & optimization workflows, in: Proceedings of the 16th Workshop on Adaptive and Reflective Middleware, ARM '17, ACM, New York, NY, USA, 2017, pp. 2:1–2:6.

[3] M.F.M Hoogreef, Advise, formalize and integrate MDO architectures: a methodology and implementation, 2017.
[4] A.R. Kulkarni, Development of Knowledge Based Engineering Tool to Support Fin-rudder Interface Design and Optimization, Master's Thesis, in: Delft University of Technology, Faculty of Aerospace Engineering.
[5] P. Voigt, A. Von dem Bussche, The EU general data protection regulation (GDPR), A practical guide, 1st Ed., Cham: Springer International Publishing, 2017.
[6] G. Bracha, D. Ungar, Mirrors: design principles for meta-level facilities of object-oriented programming languages, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 331–344.
[7] P. Grace, G. Coulson, G.S. Blair, B. Porter, A distributed architecture meta-model for self-managed middleware, in: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM'06), ACM, 2006.
[8] T. Chen, R. Bahsoon, X. Yao, A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, ACM Comput. Surv. (CSUR) 51 (3) (2018) 61.
[9] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.
[10] F.D. Macías-Escrivá, R. Haber, R.D. Toro, V. Hernandez, Self-adaptive systems: a survey of current approaches, research challenges and applications, in: Expert Systems with Applications, volume 40, 2013, pp. 7267–7279.
[11] T. Lorido-Botran, J. Miguel-Alonso, J.A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, J. Grid Comput. 12 (4) (2014) 559–592.
[12] S.K. Jensen, T.B. Pedersen, C. Thomsen, Time series management systems: a survey, IEEE Trans. Knowl. Data Eng. 29 (11) (2017) 2581–2600.
[13] F. Lee, Architectural points of extension and scalability for the elk stack, 2018, https://fabianlee.org/2016/11/28/, accessed.
[14] P. Browne, JBOss Drools Business Rules, Packt Publishing Ltd, Birmingham, B27 6PA, UK, 2009.
[15] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0., OASIS, 2013.
[16] Cloudify – Pure-Play Cloud Orchestration and Automation Based on TOSCA. Accessed 20 April 2017. https://cloudify.co/.
[17] H. El-Rewini, M. Abd-El-Barr, Advanced Computer Architecture and Parallel Processing, Vol. 42, John Wiley & Sons, Hoboken, New Jersey, USA, 2005.
[18] B. Gregg, Thinking methodically about performance, Commun. ACM 56 (2) (2013) 45–51.
[19] S. Godard. iostat(1) - linux man page.
[20] I. Santana-Perez, R.F.d. Silva, M. Rynge, E. Deelman, M.S. Pérez-Hernández, O. Corcho, Reproducibility of execution environments in computational science using semantics and clouds, Future Gener. Comput. Syst. 67 (2017) 354–367.
[21] R. Qasha, J. Cała, P. Watson, A framework for scientific workflow reproducibility in the cloud, in: E-science (E-science), in: 2016 IEEE 12th International Conference on, IEEE, 2016, pp. 81–90.
[22] B.C. Smith, Procedural Reflection in Programming Languages, Massachusetts Institute of Technology, 1982 Ph.d. thesis.
[23] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, An architecture for next generation middleware, in: Jochen Seitz (Ed.), Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Springer-Verlag, London, UK, UK, 2009, pp. 191–206.
[24] M. Roman, F. Kon, R.H. Campbell, et al., Reflective middleware: from your desk to your hand, IEEE Distributed Systems.
[25] G. Blair, F. Costa, G. Coulson, F. Delpiano, H. Duran, B. Dumant, F. Horn, N. Parlavantzas, J.B. Stefani, The design of a resource-aware reflective middleware architecture, in: Meta-Level Architectures and Reflection, Springer, 1999, pp. 115–134.
[26] D. Weyns, S. Malek, J. Andersson, Forms: unifying reference model for formal specification of distributed self-adaptive systems, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7 (1) (2012) 8.
[27] T. Furtado, E. Francesquini, N. Lago, F. Kon, A middleware for reflective web service choreographies on the cloud, in: Proceedings of the 13th Workshop on Adaptive and Reflective Middleware, ACM, 2014, p. 9.
[28] E.H. Beni, B. Lagaisse, W. Joosen, Wf-interop: adaptive and reflective rest interfaces for interoperability between workflow engines, in: Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware, ACM, 2015, p. 1.
[29] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: state of the art and research challenges, IEEE Trans. Serv. Comput. 11 (2) (2018) 430–447.
[30] M. Maurer, I. Brandic, R. Sakellariou, Enacting slas in clouds using rules, in: European Conference on Parallel Processing, Springer, 2011.
[31] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, I. Truck, From data center resource allocation to control theory and back, in: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, 2010.
[32] R. Han, L. Guo, M.M. Ghanem, Y. Guo, Lightweight resource scaling for cloud applications, Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, IEEE, 2012.
[33] R. Han, M.M. Ghanem, L. Guo, Y. Guo, M. Osmond, Enabling cost-aware and adaptive elasticity of multi-tier cloud applications, Future Gener. Comput. Syst. 32 (2014) 82–98.
[34] M.Z. Hasan, E. Magana, A. Clemm, L. Tucker, S.L.D. Gudreddi, Integrated and autonomic cloud resource scaling, Network Operations and Management Symposium (NOMS), 2012 IEEE, IEEE, 2012.
[35] Y. Zhao, X. Fei, I. Raicu, S. Lu, Opportunities and challenges in running scientific workflows on the cloud, in: Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on, IEEE, 2011, pp. 455–462.
[36] P. Kacsuk, J. Kovács, Z. Farkas, The flowbster cloud-oriented workflow system to process large scientific data sets, J. Grid Comput. 16 (1) (2018) 55–83.
[37] S.N. Srirama, J. Viil, Migrating scientific workflows to the cloud: through graph-partitioning, Scheduling and Peer-to-Peer Data Sharing, IEEE, 2014.
[38] S. Caíino-Lores, A. Lapin, P. Kropf, J. Carretero, Methodological approach to data-centric cloudification of scientific iterative workflows, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2016, pp. 469–482.
[39] D. de Oliveira, E. Ogasawara, K. Ocaña, F. Baião, M. Mattoso, An adaptive parallel execution strategy for cloud-based scientific workflows, Concurr. Comp-Pract. E. 24 (13) (2012) 1531–1550.
[40] L. Wang, R. Duan, X. Li, S. Lu, T. Hung, R.N. Calheiros, R. Buyya, An iterative optimization framework for adaptive workflow management in computational clouds, in: Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on, IEEE, 2013, pp. 1049–1056.
[41] P. Derek, T. Kapil, Understanding tosca and containers, in: OASIS.
[42] T. Chen, R. Bahsoon, Self-adaptive and online qos modeling for cloud-based software services, IEEE Trans. Softw. Eng. 43 (5) (2017) 453–475.
[43] P. Xiong, C. Pu, X. Zhu, R. Griffith, Vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ACM, 2013, pp. 271–282.

Emad Heydari Beni is a Ph.D. candidate in the Department of Computer Science at KU Leuven in Belgium, and a member of the research group imec-DistriNet. His research activities are under the supervision of Prof. Dr. Wouter Joosen and Dr. Bert Lagaisse. He received his Master's degree in computer science from the University of Antwerp in 2014. His main research interests are in the area of adaptive and reflective middleware, cloud platforms and applied cryptography.

Bert Lagaisse is industrial research and valorization manager at the imec-DistriNet research group in which he manages a portfolio of applied research projects on cloud technologies and security middleware in close collaboration with industrial partners. He has a strong interest in distributed systems, in enterprise middleware, cloud platforms and security services. He obtained his MSc in computer science at KU Leuven in 2003 and finished his Ph.D. in the same domain in 2009.

Wouter Joosen is a full professor in distributed software systems at the Department of Computer Science of KU Leuven, Belgium. He obtained a Ph.D. degree from KU Leuven in 1996. He has also co-founded spin-off companies of KU Leuven: Luciad, a company specializing in software components for Geographical Information Systems, and Ubizen (now part of Verizon Business Solutions), where he has been the CTO from 1996 till 2000, and COO from 2000 till 2002. His current research interests are in distributed systems and cloud computing, focusing on software architecture and adaptive middleware, as well as in security aspects of software.