The Journal of Supercomputing (2022) 78:13004–13039
https://doi.org/10.1007/s11227-022-04363-0

Conformance checking for autonomous multi-cloud SLA management and adaptation

Jeremy Mechouche[1,2] · Roua Touihri[1,2] · Mohamed Sellami[2] · Walid Gaaloul[2]

Accepted: 7 February 2022 / Published online: 14 March 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Satisfying cloud customers' requirements, i.e., respecting an agreed-on service level agreement (SLA), is not a trivial task in a multi-cloud context. This is mainly due to divergent SLA objectives among the involved cloud service providers and hence divergent reconfiguration strategies to enforce them. In this paper, we propose a hierarchical representation of multi-cloud SLAs: sub-SLAs associated with a system's components deployed on distinct cloud service providers and global-SLA associated with the whole system. We also enrich these SLA representations with state machines reflecting reconfiguration strategies defined by cloud customers. Then, we propose an autonomous multi-cloud resource orchestrator based on the MAPE-K adaptation control loop to enforce them and to avoid SLA violations. Finally, in order to check the conformity of this enforcement with defined multi-cloud SLA, we propose an approach for multi-cloud SLA reporting inspired by conformance checking techniques. An implementation of the approach is presented in the paper and illustrates the approach feasibility.

Keywords SLA · Cloud resources · Multi-cloud · Autonomous computing · Reconfiguration strategy · State machine

1 Introduction

Cloud computing has become a mature technology and the de facto solution for companies to host their IT systems in the last ten years. According to the last big outages of multiple cloud-based systems, relying on a single cloud service provider can be problematic due to risks such as availability zones going down or network failures. A solution is to leverage the multi-cloud paradigm to distribute a system's

✉ Jeremy Mechouche
jeremy.mechouche@devoteam.com

1 Devoteam, R&D, Massy, France

2 Samovar, Telecom SudParis, Institut Polytechnique de Paris, Palaiseau, France

components over multiple cloud service providers in order to mitigate these risks. Multi-cloud denotes the usage of resources from multiple and independent clouds by a customer (e.g., enterprise); contrary to a federated cloud where multiple cloud providers unify their resources to provide a common service to a customer. Cloud customers use multi-cloud for different reasons: (1) improving cost-effectiveness, (2) avoiding vendor lock-in, (3) ensuring backups to deal with disasters, and (4) consuming particular services that are not provided elsewhere. The *Gartner Hype Cycle for Cloud Computing 2020* forecasts that multi-cloud will be a mature concept within the next 5 years [1]. Also, the *RightScale State of the Cloud report* shows that 92% of enterprises have a multi-cloud strategy in 2021 [2]. Furthermore, several European collaborative research projects propose approaches for consuming multi-cloud services such as *Melodic* [3], *Decide H2020* [4], *Cyclone* [5], *MODA-Clouds* [6] and *SeaCloud* [7]. Many industry products aim to adopt the multi-cloud paradigm. These products are carried by cloud services providers such as Google cloud with Anthos[1], Amazon Web Service with EKS/ECS Anywhere[2] and Microsoft Azure with Arc[3].

Despite this great interest in the multi-cloud paradigm, maintaining a certain level of service for a multi-cloud application is still a non-trivial issue. One solution is to formalize the required level of service needed as a service level agreement (SLA). In a multi-cloud context, this latter SLA is denoted as a *multi-cloud SLA* which allows representing requirements for a multi-cloud application. A few works have looked at the modeling of multi-cloud SLA, such as [6] or [8]. However, these representations do not consider the dynamicity and elasticity of such multi-cloud applications, which is a major characteristic of a cloud application [9]. Indeed, dynamicity can have an impact on the SLA and violate service level objectives, e.g., scaling-out a resource to comply with an availability objective can lead to the violation of a cost objective. Another, non-trivial activity due to the distributed nature of the multi-cloud and the heterogeneity of service level objectives is SLA reporting. This activity, which consists of identifying what happened during the enforcement of the SLA, is also partially covered in the literature [10].

In order to solve these issues, we first model a multi-cloud SLA associated with a composite application as: (1) a global SLA that contains the requirements for a whole multi-cloud application, and (2) several sub-SLAs, one for each component of the application, that contains the component's requirements. Then, to address the dynamicity issue, we propose a formalism based on the state-machine semantics to enrich the multi-cloud SLA. This state-machine formalism allows representing different user requirements and how providers can technically address these requirements. Hence, it allows representing the fine-grained service level objectives of a multi-cloud SLA. We denote the resulting multi-cloud SLA as an enriched multi-cloud SLA.

[1] https://cloud.google.com/anthos.

[2] https://aws.amazon.com/eks/eks-anywhere/.

[3] https://azure.microsoft.com/en-us/services/azure-arc/.

Next, to enforce an enriched multi-cloud SLA, we propose a multi-cloud resources orchestrator which is compliant with the state-machine formalism. This orchestrator follows an autonomous approach based on the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) adaptation control loop [11] to enforce the enriched multi-cloud SLA. MAPE-K is a widely used reference control model for autonomic and self-adaptive systems. This autonomous approach avoids SLA violations by reconfiguring the application if needed. For instance, when a scale-out leads to the violation of a cost requirement, our approach validates the reconfiguration strategy upstream and notifies the cloud consumer of the inconsistency.

Finally, we propose a multi-cloud SLA reporting approach based on conformance checking techniques. Reporting of multi-cloud SLA is a complex task due to the distributed nature of the multi-cloud and the heterogeneity of service level objectives across multiple cloud service providers. However, multi-cloud SLA enforcement is a process that produces event logs during its execution. Therefore, we rely on process mining techniques, conformance checking in particular, to validate and adapt the multi-cloud SLA enforcement. Indeed, conformance checking is a set of techniques to analyze of business process, it relies on logs and process models to check if the actual execution, as recorded in the event log, conforms to the model and vice versa.

The remainder of this article is organized as follows. First, Sect. 2 introduces the multi-cloud service level agreement representation we consider in this work. Then, Sect. 3 discusses a motivating example. After that, Sect. 4 presents an overview of our approach. Next, Sect. 5 discusses related works. Then, Sects. 6 and 7 illustrate the enrichment of multi-cloud SLA with state machines and its enforcement with an autonomous orchestrator, respectively. Then, Sect. 8 presents the multi-cloud SLA reporting approach we propose in this work. Next, Sect. 9 describes an implementation and an experimentation of our approach. Finally, Sect. 10 concludes the paper with an outlook on future works.

2 Cloud and multi-cloud service level agreement

Cloud users establish a service level agreement (SLA) defining the required service quality provided by the cloud providers. An SLA defines a commitment between a customer and a service provider. This latter agreement is composed of a set of objectives defining an agreed-upon quality of service (QoS). The main phases of the SLA lifecycle are *negotiation*, *deployment*, *monitoring*, *reporting* and *termination* [12]. Particularly, in a cloud context, an SLA (Definition. 2) is mainly defined by four parts (1) parties, (2) service terms, (3) service level objectives, and (4) penalties [13]. First, parties define stakeholders of the agreement. Second, service terms characterize the service covered by the agreement, e.g., virtual server and database. Third, service level objectives (SLO), as defined in Definition 1, present the agreed-upon QoS terms. Finally, penalties define a form of compensation in case of the agreement's violation. These latter SLOs of the application express non-functional requirements (NFR) such as availability, performance, cost, or security [12]. In this paper, we choose to describe these SLAs with the ISO-19086 standard [14, 15], to deal with the heterogeneity issue of SLA [16]. This latter standard support objective description in multiple formats. ISO-19086 is already used in security SLA [17], for example in the public STAR (Security, Trust, and Assurance Registry) repository[4].

Definition 1 (SLO) A service level objective (SLO) is a term made by a service provider for a specific service characteristic. It is defined as a 5-tuple (c, t, v, u, op) where: c denotes a specific measurable characteristics of the SLA (e.g., availability, response time, cost, or security); t defines c's type, i.e., quantitative or qualitative; v defines c's expected value; u represents c's measurement unit; and $op \in \{<, >, \geq, \leq$ or $=\}$ defines the comparison operator of c.

Definition 2 (SLA) An SLA is a contract between one service provider and a customer that defines their agreement terms. It is defined as a couple (n, O) where: n is a name identifying the SLA and $O =< k, v >$ is a hash table representing a list of offered cloud services and their associated SLOs. The key $k \in K$ identifies a covered service (e.g., service used by an application's component). The associated value $v \in V$ is represented by a couple $(csp, \mathcal{SLO})$ where csp defines k's provider and $\mathcal{SLO}$ defines the set of k's associated SLOs as defined in Definition 1.

Multi-cloud refers to the usage of resources by a customer from multiple and independent clouds. A multi-cloud SLA is composed of two categories of SLA: a *global-SLA* and multiple *sub-SLAs*. As defined in Definition 3, a global-SLA represents the client's requirements for a multi-cloud application. A sub-SLA describes the SLA of a multi-cloud application component with the same SLA format of Definition 2. The decomposition of the multi-cloud SLA allows representing finely the requirements of the entire multi-cloud application and their components.

Definition 3 (Global-SLA) A global-SLA is a contract between **many** service providers and a customer that defines their agreement terms for a multi-cloud application. It is defined by a set of $\mathcal{SLO}$ as defined in Definition 1, associated with the multi-cloud application.

3 Motivating example

To illustrate the motivation for this work, we consider a cloud application composed of three components: **User Interface**(*UI*), **Authentication**(*Auth*) and **Storage**(*Stor*). This application respects the principles of composite cloud applications design pattern [18]. The cloud application's components are provided by three cloud service providers: CSP_1, CSP_2, and CSP_3. A cloud architect is in charge of the deployment of the required cloud resources and has to respect predefined final users requirements for the multi-cloud application in terms of availability, response time, and cost. The user's workload fluctuates. Therefore, in order to handle scheduled and unscheduled

[4] https://cloudsecurityalliance.org/star/.

Table 1 Service level objectives associated with Global$_{SLA}$

Service level objective	$Gslo_1$	$Gslo_2$	$Gslo_3$
Characteristics	Response time	Availability	Cost
Type	Quantitative	Quantitative	Quantitative
Value	5	99.7	10
Unit	ms	%	$/h
Operator	$\leq$	$\geq$	$\leq$

Table 2 Service level objectives associated with UI's Sub$_{SLA}$

Service level objective	slo_1	slo_2	slo_3
Characteristics	Response time	Cost	Availability
Type	Quantitative	Quantitative	Quantitative
Value	4	2	99.9
Unit	ms	$/h	%
Operator	$\leq$	$\leq$	$\geq$

peaks of activities, the cloud architect defines three different reconfiguration strategies associated with the applications' components. These strategies are denoted as **normal needs**, **high needs** and **low needs**. A global-SLA, defined according to Definition 3, denotes the predefined final users' requirements for a response time less than 5 ms, an availability greater than 99.7% and a cost per hour below 10$. $Global_{SLA}$ is represented as $< Gslo_1, Gslo_2, Gslo_3 >$ where the different $Gslo_i$ are defined in Table 1.

For this example, we consider the User Interface sub-SLA (Definition 2) for the sake of simplicity. It defines three SLOs: (1) a response time lower than 4 ms, (2) a cost per hour below 2$ and (3) an availability rate at 99.9%. The *UserInterface* sub-SLA is represented as $< UI, (CSP_1, slo_1, slo_2, slo_3) >$ where the different slo_i are defined in Table 2.

The three reconfiguration strategies for UI are defined as follows: *normalNeeds* with 2 virtual machines, *highNeeds* with 4 virtual machines, *lowNeeds* with 1 virtual machine, and *final* with 0 virtual machine. To implement such reconfiguration strategies and assess their compliance with the SLA of the multi-cloud application, there is a need for (1) representing the requirements of the multi-cloud application as a multi-cloud SLA expressing the agreement between the different involved parties, (2) ensuring that the reconfiguration strategy associated to a sub-SLA respects global-SLA, and (3) reporting that the agreed-upon SLA corresponds to the provided service.

Indeed, these reconfiguration strategies are defined by a user, and thus are error-prone, i.e., eventually does not comply with users requirements. These incorrect reconfiguration strategies can result from a wrong resource definition or an unconsidered workload change of a multi-cloud application or one of its components. Such incorrect strategies can lead to an SLA violation which can have negative consequences for a cloud service provider such as reputation damage or penalties. For

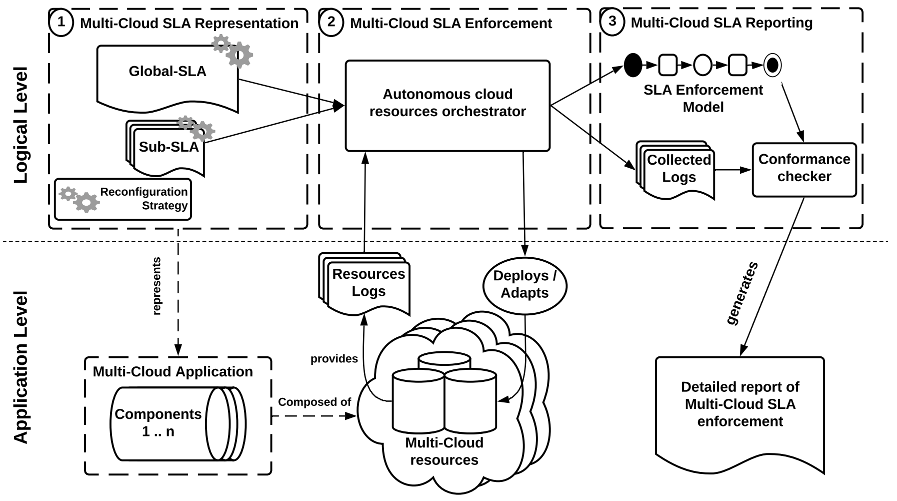


Fig. 1 Approach overview

avoiding such issues, the SLA-enforcement behavior (i.e., the behavior depicted by the reconfiguration strategy) of the *application*, and its *underlying components*, have to be considered in the SLA representation. After that, a method to enforce this latter SLA representation has to be defined. Finally, the enforcement has to be reported to verify if the SLA has been respected or not. In previous work, we proposed an approach that represents multi-cloud SLA [19]. In this paper, we propose to enforce multi-cloud SLA with an autonomous orchestrator based on MAPE-K. Then, we propose to ensure this latter enforcement using conformance checking techniques. An overview of our approach is depicted in the next section.

4 Approach overview

In a multi-cloud context where resources are distributed over multiple cloud service providers, maintaining the objectives defined in a Global-SLA is not a trivial task. Moreover, multi-cloud SLA reporting needs a significant effort in terms of time when the architecture becomes complex. In this article, the objectives of our proposed approach are to represent multi-cloud SLA, to avoid multi-cloud SLA violations and to analyze the enforcement of multi-cloud SLAs.

- To represent multi-cloud SLA, we rely on the model presented in Sect. 2. However, the latter representation needs to be enriched with reconfiguration strategies to handle the dynamicity of the multi-cloud environment.
- To avoid SLA violations, we rely on an autonomic cloud orchestrator based on the *MAPE-K* loop [20] for the multi-cloud SLA enforcement.
- To analyze multi-cloud SLA enforcement, we rely on *process mining* techniques [21]. In particular, we rely on conformance checking techniques [22], used in the

business process management community, in order to check if the actual execution of a business process conforms to its model.

As represented in Fig. 1, our approach is composed of three steps: (1) the representation of SLA associated with multi-cloud applications, (2) the enforcement of multi-cloud SLA using an orchestrator based on the MAPE-K loop, and (3) the reporting of the multi-cloud SLA enforcement using a conformance checking technique.

In (1), the multi-cloud SLA representation is enriched with its reconfiguration strategy represented as a state machine. The enrichment of multi-cloud SLAs enables to manage the dynamicity of multi-cloud applications. A detailed description of this multi-cloud SLA representation is provided in Sect. 6.

In (2), this latter multi-cloud-SLA is enforced with an autonomous cloud resources orchestrator that interacts with the cloud resources of the multi-cloud application. The latter interactions consist of deploying, adapting, and collecting event logs from cloud resources from several providers. Our proposed orchestrator relies on an autonomic approach that will add a self-adaption behavior to the resources according to the fluctuating workload of the users. Indeed, through this autonomic approach, we give the orchestrator the capability to extend an SLA's state machine, i.e., representing its associated reconfiguration strategy, which represents the applications' reconfiguration strategy, at run-time in order to adapt these strategies if needed. A detailed description of this orchestrator's architecture is provided in Sect. 7.

Finally, in (3), the third part of Fig. 1 illustrates our proposed multi-cloud SLA enforcement reporting approach. As stated before, we rely on a conformance checking technique to analyze the SLA enforcement. As input for the conformance checking, we consider: (1) *collected logs* from our orchestrator, the cloud service provider, and the cloud resource, and (2) *SLA enforcement model* representing the entire SLA enforcement behavior to be reported. Conformance checking techniques will allow us to get statistics on the enforcement and to determine a rate of conformance between the *SLA enforcement model* and *collected logs*. This latter technique will provide a detailed report on the SLA enforcement as output.

Compared to the related works and the literature (Sect. 5), the approach we proposed handles the dynamicity and the SLA enforcement of a multi-cloud context. We also report the multi-cloud SLA by checking the conformance of the multi-cloud SLA enforcement with its representation. A detailed description of this multi-cloud SLA reporting technique is provided in Sect. 8.

5 Related work

In line with the steps of our approach, we divide our literature review into three parts: multi-cloud SLA (Sect. 5.1), autonomic SLA enforcement in the cloud (Sect. 5.2), and SLA reporting with process conformance checking techniques (Sect. 5.3).

5.1 Multi-cloud service level agreement

Cloud SLAs and SLAs, in general, have received considerable attention. However, little attention is given to the heterogeneity and dynamicity related issues faced while specifying these SLAs in a multi-cloud context [16]. The most used SLA specification language representation is WSLA [23] which was proposed in 2003 by *IBM* for the specification of service level agreements for Web Services. Based on WSLA, various propositions of SLA specification languages have been made for the cloud context; SLA* [24], rSLA [25], and ySLA [26], for example. However, they did not propose mechanisms to handle the cloud's dynamic nature. Other propositions have been made to handle this dynamicity in cloud SLAs: SLAC [27] and CSLA [28]. Uriarte et al. proposed with SLAC, an SLA language where stakeholders can define at design-time multiple levels of service. Kouki et al. consider, in CSLA, dynamicity in the SLOs with the concepts of *fuzziness* and *confidence*. *Fuzziness* defines acceptable margins for the SLO and *confidence* denotes a percentage of compliance of objectives.

Cloud SLA specification languages need to be adapted to meet the specificities of the multi-cloud context. Son et al. defined cloud SLA relationships (i.e., which stakeholder is responsible for what) in a multi-cloud environment based on different cloud resources consumption models [29]. They analyze different multi-cloud models such as : *peer-to-peer cloud federations*, *centralized cloud federation* or *distributed cloud* and describe the SLA relationship between consumers and cloud providers for each of them. Moreover, some works cover multi-cloud service composition based on SLA but assume that SLAs are homogeneous. Such as Farokhi et al. in [8] where they proposed a hierarchical SLA-based service selection for multi-cloud environments. This approach, similar to our approach for multi-cloud SLA representation, uses multiple SLAs to represent the SLA associated with different cloud service providers in a multi-cloud environment. Multi-cloud SLA is also considered in some open-source and European projects which aim to orchestrate resources over multiple cloud service providers. This is the case of the *MODAClouds* project, which proposed a model-driven approach for the design and the execution of applications on multiple clouds [6]. Ardagna et al. refers, in [6], to a two-level SLA system: a first level describing the SLA between customers and cloud providers and a second level describing the QoS expected from the cloud provider. This latter SLA specification language is also used in the *SeaClouds* project [7]. In previous work [19], we introduced state-machine enriched multi-cloud. This latter representation takes into account the dynamicity of multi-cloud components. We also proposed an approach to validate their compliance with the user requirements. However, the dynamicity of the entire multi-cloud application, the enforcement, and reporting of the multi-cloud SLA was not considered.

Compared to these works, we consider the representation and the dynamicity of multi-cloud SLA in its definition.

5.2 Autonomic SLA enforcement in the cloud

MAPE-K is a widely used autonomic architecture style in cloud SLAs management approaches which has proved to be a powerful tool to build self-adaptive systems [12]. Indeed, as presented by Faniyi et Bahsoon in [12], there are four key motivations for using autonomic computing: (1) large size system, (2) heterogeneous context, (3) dynamic user workload fluctuation, and (4) uncertainty about the state of the environment.

Emeakaroha et al. proposed in [30] an autonomic approach, *DeSVI*, for monitoring and detecting SLA violations. This approach consists of two components: an *automatic VM deployer* responsible for resource allocation and *LoM2HiS*, responsible for monitoring the application execution and translating low-level metrics into high-level SLAs (e.g., availability). Mosallanejad et Atan [31] present a model for hierarchical SLA specific to the cloud domain. In this model, each SLA can monitor its attributes and communicate with dependent SLA (which depend on them) in a different layer of cloud (IaaS/PaaS/SaaS). The approach allows independent management of SLA (autonomic SLA) by considering SLA as an active entity (i.e., an SLA is responsible for monitoring its objectives). However, this approach considers only SLA monitoring. Casalicchio et al. presented in [32] an autonomic QoS-aware service provisioning architecture based on the MAPE-K loop. This work proposes to compare four different solutions designed for controlling the application based on the MAPE-K loop and how this is implemented with the functionalities of an IaaS provider. This design has been evaluated using features and services of the Amazon Elastic Compute Cloud (EC2) infrastructure. Ghobaei-Arani et al. proposed in [33] a framework for autonomic resource provisioning based on the control MAPE-K loop. In their approach, they also use a reinforcement-learning based method as a *decision-maker* for the planning phase. However, this latter approach is limited to a sole IaaS provider and do not consider a multi-cloud context. Sfondrini et Motta [34] proposed to reduce the SLA violation rate and cover all SLA management activities in an SLA-aware Lean Information Service Architecture (*LISA*). This architecture includes an autonomic resource allocation engine that manages the analysis, plan, and execution phases. Monitoring and knowledge are managed by other modules. This resource allocation engine is alerted of SLA breaches by the monitoring system. Then, to reduce the impact on the business continuity analysis of the concerned entity is performed. This analysis results in a potential identification of the SLA violation root cause and proposes a solution. Kosińska et Zieliński proposed in [35] an Autonomic Management Framework for Cloud-Native Applications (*CN App*), *AMoCNA*. The focus of this approach is on containerized applications. They proposed to implement the autonomic control-loop with a rule-engine to replace the analyze and plan function. So, their control-loop is called MRE-K for: Monitoring, *Rule Engine*, Execute and Knowledge. Rouf et al. [36] proposed a framework called *COTS* based on the MAPE-K loop to manage multi-cloud platforms. The main objective of this paper is to explore the feasibility of developing an autonomic MAPE-K framework for multi-cloud platforms by integrating existing services. Their framework architecture is composed

Table 3 Related works summary of autonomic SLA enforcement (Sect. 5.2) and SLA reporting using business process management (Sect. 5.3)

Paper	Scope	Cloud resources	SLO
Enforcement			
Emeakaroha et al. [30]	Cloud	VM	Limited range
Casalicchio et al. [32]	Cloud	VM	Limited range
G-A et al. [33]	Cloud	VM	Limited range
AMoCNA [35]	CN App	Container	Limited range
COTS [36]	Multi-Cloud	Any (IBM CAM[a])	Limited range
LISA [34]	Multi-Cloud	Any (Cloudify[b])	WS-Agreement [23]
Our approach	Multi-Cloud	Any ([49])	ISO-19086 [14, 15]
Paper	**Scope**	**Employed techniques**	**Process description**
Reporting			
Van Eck et al. [40]	Generic	Process discovery	CSM
Sutrisnowati et al. [41]	Big data	Process discovery	Not specified
Chesani et al. [42]	Cloud	Process monitoring	Declare
Acampora et al. [44]	Cloud	Online process discovery	Declare
Song et al. [45]	Multi-Cloud	Process discovery	Scientific Workflow
Calcaterra et al. [46]	Cloud	Resource orchestration	BPMN
Azumah et al. [48]	Hybrid-Cloud	Resource scheduling	Declare
Our approach	Multi-Cloud	Conformance checking	Petri-Net with Data

[a]https://www.ibm.com/us-en/marketplace/cognitive-automation

[b]https://cloudify.co/

of three components: Cloud Monitor, State Rule Engine, and Workflow Engine. These three components can be compared to the MRE-K control loop proposed in *AMoCNA* [35].

The aim of these approaches is close to ours; however, our approach considers an abstract view of cloud resources and focus on multi-cloud-SLA enforcement. We present a comparison between autonomic SLA management approaches using a MAPE-K loop in Table 3 in the *Enforcement* part. For this table, we consider three comparative criteria, namely: *Scope* which defines the action context of the approach, *Cloud Resources* which defines the type of the resource considered in the approach and *SLO* which defines the kind of objectives considered.

5.3 Service level agreement reporting

There is little attention given to SLA reporting [12] even though it is a significant need for real-world use cases. Ismail et al. defined in [37] a generic SLA management framework. Then, proposed to model this latter SLA management framework using business process management notation in order to simplify SLA offerings. The major point of this approach is the mapping between WSLA [23] and BPM notation

[38] which is a concrete implementation with a widely used SLA format. However, this approach only considers the SLA definition and do not validate SLA enforcement. A first proposition using process mining techniques to report SLA has been made in [39]. This approach proposes the detection of bottlenecks in the process by a fine-grained analysis of the time perspective. However, in this latter approach, only control-flow and time perspectives are considered.

Some other works about cloud resources orchestration get inspired by business process management techniques. Van Eck et al. introduced in [40] the concept of composite state machines (CSMs) to describe multiple related processes, depicted as perspective. They proposed an algorithm of process discovery and a method to simplify the views for CSMs which can be quite complex. Sutrisnowati et al. presented in [41] a tool for performing process mining techniques over Hadoop Map-Reduce (Big Data framework[5]). The BAB framework aims to propose, like Prom, a tool that combines the Hadoop Map-Reduce algorithm with cloud and process mining. They implemented a process discovery algorithm in this context and the framework allowed being extended with other functionalities. However, it is limited to Hadoop Map-Reduce. Chesani et al. proposed in [42] to apply well-known process mining techniques, Mobucon EC [43], to monitor properties of a Map-Reduce execution. Then, this monitor is used to determine the health of the system and determine auto-scaling action for nodes composing the Map-Reduce cluster. This paper presents encouraging results for the use of process mining techniques in the cloud context. However, this approach only considers the Map-Reduce application. Acampora et al. proposed in [44] a cloud controller that performs auto-scaling action over a cloud computing infrastructure. This cloud controller exploits information from business processes discovered from the logs. Processes are represented using Declare language. Online process mining techniques are performed for extracting the required information from event logs. Song et al. presented in [45] an approach considering the extraction of intra and inter-cloud scientific workflow from event logs using process mining techniques. This algorithm is proposed as a ProM plug-in to perform the process discovery of this scientific workflow. Calcaterra et al. proposed in [46] a fault-aware orchestrator of cloud resources using the business process modeling notation language (BPMN) [47] for describing the scheme of service provision workflow. In this paper, they put a focus on the failures during the provisioning process. Azumah et al. proposed in [48] to use a process mining to schedule tasks in a hybrid cloud context complying with a set of given business constraints.

To the best of our knowledge, reporting of multi-cloud SLA using conformance checking techniques is not covered in the literature. We proposed to use these latter techniques to check the validity of SLA enforcement and get a detailed report which considers the entire multi-cloud SLA. We present a comparison between approaches using business process management techniques in a cloud context in Table 3 in the *Reporting* part. For this table, we consider three comparative criteria, namely: *Scope* which defines the action context of the approach, *Employed Techniques* which

[5] https://hadoop.apache.org/.

Table 4 UI sub state-machines events

Id	Type	Predicate
Evt1	Resource-related	cpuUsage, average, >, 85, %, 60 s
Evt2	Resource-related	cpuUsage, average, <, 30, %, 60 s
Evt3	Resource-related	cpuUsage, average, <, 30, %, 60 s
Evt4	Resource-related	cpuUsage, average, >, 85, %, 60 s
Evt5	Temporal-event	Everyday at 9:00 pm

defines the process mining techniques used in the approach and *Process Description* which defines the process description method employed.

6 Multi-cloud SLA representation

This section details the first step of our approach. In the following, we discuss how we enrich multi-cloud sub-/global-SLAs with state machines representing reconfiguration strategies in order to consider multi-cloud SLA violations.

6.1 State machines for representing reconfiguration strategies

SLAs are static and do not consider the dynamicity of covered services which is, according to NIST [9], a key concept for cloud computing. This dynamicity is expressed through reconfiguration strategies defined by cloud consumers to manage the application's dynamic behavior, such as activity peaks. In order to handle the dynamicity of service covered by an SLA, we propose to use state machines that represent reconfiguration strategies. State machine is chosen due to its widespread adoption, intuitiveness, and effectiveness. Please note that this intuitive method is commonly used when modeling dynamic systems behavior and can be formally validated.

A reconfiguration strategy is defined through one or multiple events which trigger one or several reconfiguration actions. These reconfiguration actions correspond to the actions required to reconfigure a service. An event (Definition 4) represents the occurrence of any change that results in triggering specific actions [49]. There are four types of triggering events: (1) temporal, (2) resource-related, (3) user action, and (4) composite. First, Temporal Events occurs on a specified date or after some time. Second, Resource Related Events happens once a resource metric meets a predefined reference value. Third, User Action Events appears on user demand. Finally, Composite Events are a specific type that is composed of multiple triggering events specified earlier.

Definition 4 (Event) A triggering event is represented as a 3-tuple (id, t, p) where: id is the triggering event's identifier; $t \in \{temporal, resourceRelated, userDefined, composite\}$ is the predicate's type and p represents the predicate, i.e., the body, of the events according to the predefined predicate type.

Table 5 UI sub state-machine's action

Id	Type	[Action attributes]
A1	HorizontalScaling	[UI, scale-out, 1]
A2	HorizontalScaling	[UI, scale-out, 2]
A3	HorizontalScaling	[UI, scale-in, 1]
A4	HorizontalScaling	[UI, scale-in, 2]
A5	HorizontalScaling	[UI, scale-in, 4]

Table 6 UI sub state-machine's states

Label	Type	Resources				
		id	idResource	CSP	size	nb
NormalNeeds	isInitial	UI	VM1	CSP1	M	2
HighNeeds	isNormal	UI	VM1	CSP1	M	4
LowNeeds	isNormal	UI	VM1	CSP1	M	1
End	isFinal	UI	VM1	CSP1	M	0

For instance, we consider events associated with our motivating example triggering the UI reconfiguration strategies presented in Sect. 3. According to Definition 4, these events will be represented as follows (Table 4):

A reconfiguration action (Definition 5) specifies how a resource should behave when specific triggering events occur to respect an SLA. We define six types of reconfiguration actions: Horizontal Scaling, Vertical Scaling, Migration, Application Reconfiguration and Basic Action. These reconfiguration actions represent the elasticity mechanisms associated with cloud resources. Each of these action types requires a specific set of *action attributes* to be executed, e.g., *resource-target* or *attribute-target* [49].

Definition 5 (Reconfiguration Action) A reconfiguration action is defined as 3-tuple (id, t, AA) where : id identifies the reconfiguration action; t defines the type of the reconfiguration action and AA is a set of action attributes required to perform the action.

To illustrate reconfiguration actions representation, we refer again to our motivating example. We consider actions triggered by the events of the UI reconfiguration strategies presented in Table 4. These actions are represented in Table 5.

Thus, a reconfiguration strategy is a set of reconfiguration actions and associated events that we represent with the state-machine formalism. Indeed, a resource is characterized by states with reconfiguration actions to transit from a state to another. A state machine is defined through two main elements: **state** defined in Definition 6 and **transition** defined in Definition 7.

Definition 6 (State) A **state** is represented as a 3-tuple (l, t, R) where : l is the state's name or **label**; $t \in \{isInitial, isNormal, isFinal\}$ is the state **type** and indicates

Table 7 UI sub state-machine's transitions

Id	Source	Target	Events	Actions
*T*1	NormalNeeds	HighNeeds	Evt1	A2
*T*2	HighNeeds	Normal	Evt2	A4
*T*3	NormalNeeds	LowNeeds	Evt3	A3
*T*4	LowNeeds	Normal	Evt4	A1
*T*5	NormalNeeds	End	Evt5	A4
*T*6	HighNeeds	End	Evt5	A5
*T*7	LowNeeds	End	Evt5	A3

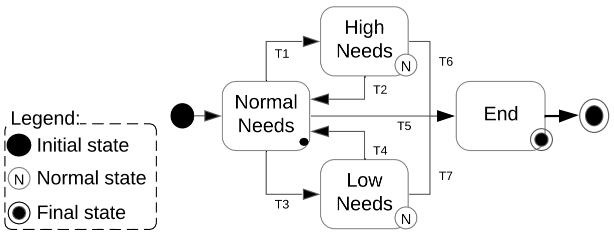


Fig. 2 Graphical representation of UI Sub-SLA state machine

whether *s* is a start state, an intermediate state, or an end state, respectively; and **R is the set of cloud resources** *r* composing an application's components or a composite application characterizing the state.

For instance, we consider states of our motivating example UI reconfiguration strategies presented in Sect. 3 and represented in Table 6.

Definition 7 (Transition) A **transition** is represented as a 5-tuple (id, s_s, s_t, E, A) where **id** is the transition's identifier, $\mathbf{s_s}$ is the source state, $\mathbf{s_t}$ is the target state, **E** is the set of events (Definition. 4) that could trigger the transition, and **A** is the set of actions (Definition. 5) to be executed when certain triggering events happen.

For instance, we consider the state machine representing UI reconfiguration strategies as mentioned in our motivation example (Sect. 3) is depicted in Fig. 2 and where its associated transitions are shown in Table 7.

6.2 Global and sub SLA enrichment

As introduced in Sect. 2, we consider a multi-cloud SLA composed of: (i) one global-SLA that defines, as part of this agreement, the cloud customer's requirements for the multi-cloud applications and (ii) one sub-SLA for each component of the multi-cloud application that denotes, as part of this agreement, its related requirements (i.e., vis-à-vis the relevant cloud service provider). To handle SLA

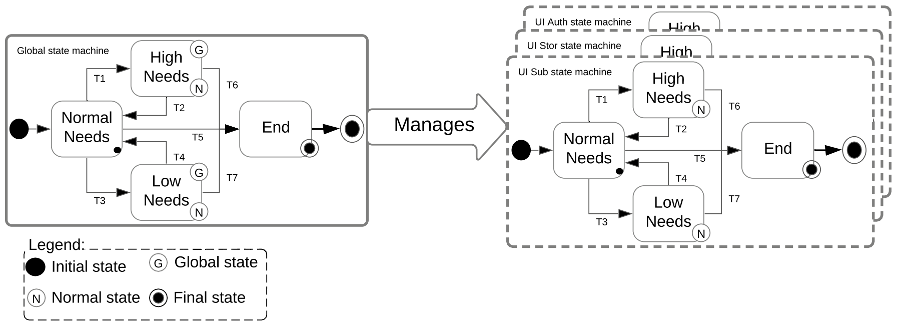


Fig. 3 Global state machine manages sub state machines

dynamicity, we propose to enrich multi-cloud SLA representations with a state machine showing reconfiguration strategies associated with the multi-cloud application and reflecting this dynamicity. We denote this latter SLA as an enriched SLA (Definition 8).

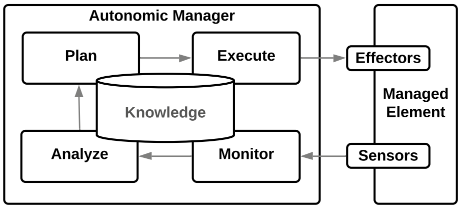
Definition 8 (Enriched SLA) An `enriched SLA` is an SLA enhanced with a state machine representing its associated reconfiguration strategy. It is defined as a 4-tuple (n, $\mathcal{SLO}$, S, T) where: n is a name identifying the enriched SLA, $\mathcal{SLO}$ defines the set of SLOs (Definition 1), S is the set of states of the state machine, and T is the set of transitions of the state-machine.

For instance, consider the UI sub-SLA presented in our motivating example (Sect. 3). The associated enriched sub-SLA is expressed as follows: $< UI, (CSP_1, slo_1, slo_2, slo_3), S, T >$ where UI is the sub-SLA's name, $(CSP_1, slo_1, slo_2, slo_3)$ is the SLO's related to UI sub-SLA, S is a set of four states shown in Tables 6 and 7 the set of transitions depicted in 7. In the same way, we enrich the global-SLA by defining a global reconfiguration strategy that manages, i.e., by triggering transitions if needed, all enriched sub-SLAs. Figure 3 illustrates the enriched global-SLA managing the sub state machines associated with our motivating example (Sect. 3). For instance, if the global state machine transits to a state, then all the sub state machines will transit to the same state. To do so, we enrich states definition (Definition 6) with a new type of state : Global State. So in Fig. 3, the state Normal Needs of the global SLA is of type Normal and type global meaning that when this state-machine transitions to Normal Needs all managed state machines, i.e., part of the sub-SLAs, will transit to Normal Needs too.

In this example, to simplify, we assume that the same reconfiguration strategy is used for all CSPs. But, in the case when the studied strategies are different between several components, we need to map the global states of the state machines to the states of the sub state machines.

However, these state machines are defined by a user who could forget cases or make errors in the state-machine definition. For instance, consider a high peak of needs when the actual state is on *lowNeeds*. The state machine will need to change

Fig. 4 MAPE-K Loop



state to *highNeeds* but according to Table 7, there is no direct transition between those two states. This last issue is addressed in the next section by introducing the autonomous multi-cloud orchestrator that can adapt to these state machines if needed.

7 Multi-cloud SLA enforcement

In this section, we present the architecture of the proposed SLA-aware multi-cloud autonomic resources orchestrator. The orchestrator's architecture is based on the MAPE-K loop and comes as part of the second step of our approach. In the following, Sect. 7.1 introduces the concept of autonomic computing then Sect. 7.2 details the architecture of the autonomic orchestrator we propose to enforce multi-cloud SLAs.

7.1 Autonomic computing

Autonomic computing helps to address complexity by using technology to manage technology [20]. The concept of autonomic is derived from human biology. The autonomic nervous system is responsible for managing vital functions without conscious effort. This is the same idea in autonomic computing; however, systems are managed according to policies defined by IT professionals.

These functions are represented as control loops that collect data from managed elements and act accordingly. There are four control-loop function categories [20] which can be defined as: Self-configuring can dynamically adapt to changing environments, Self-healing can discover, diagnose, and react to disruptions, Self-optimizing can monitor and tune resources automatically, and Self-protecting can anticipate, detect, identify, and protect against threats.

IBM proposed a control-loop architecture, usually denoted as MAPE-K loop, composed of four functions (Fig. 4) to implement an autonomic manager:

- *Monitor* collects, aggregates, filters, and reports data from managed elements through sensors associated with the managed elements,
- *Analyze* permits to correlate and model complex situations,
- *Plan* constructs the actions needed to achieve goals and objectives, and
- *Execute* executes the actions that control the managed elements through associated effectors.

All these functions are based on the knowledge module, which represents the shared data management for all components.

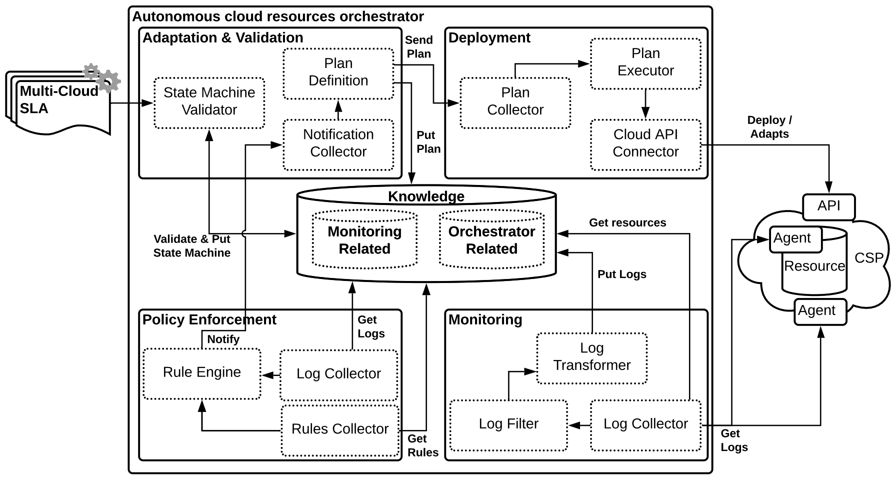


Fig. 5 Autonomous Orchestrator Architecture

7.2 Multi-cloud autonomous orchestrator

To enforce the previously proposed multi-cloud SLA (Sect. 6), an SLA-aware orchestrator is needed. So based on the reference architecture for cloud resources orchestrator framework proposed in [10] and the MAPE-K loop in [20], we propose a multi-cloud SLA autonomous orchestrator whose architecture is depicted in Fig. 5. We rely on these two proposals to propose an autonomous approach for multi-cloud resources. The latter orchestrator is composed of four components: Monitoring, Policy Enforcement, Adaptation & Validation and deployment engine.

The loop begins with verification of multi-cloud SLA by the Adaptation & Validation component. Then, the Deployment engine deploys resources at their initial state, according to the verified multi-cloud SLA state machines. Next, the Monitor component monitors the SLOs defined in the multi-cloud SLA and stores collected logs in Knowledge. Finally, in case of transition triggered or SLOs about to be violated, Policy Enforcement notifies the Adaptation & Validation component to define a plan for (Re-)Deployment to execute the action(s) associated with the transition or avoid SLOs violation.

Each component is associated with a MAPE-K function and is presented in detail in the following.

- *Adaptation and Validation* This component is associated with the Plan function of the MAPE-K loop. The functions of this component are twofold: (1) to validate state-machine correctness (State Machine Validator module in Fig. 5) according to global-SLA requirements using our approach proposed in [19] and (2) to define a plan (Plan Definition module in Fig. 5) for the managed element configuration according to a notification sent by the policy enforcement component (Notification Collector module in Fig. 5). A plan defines the actions to be performed by the deployment component to (re-)deploy a cloud resource. These latter actions are defined as in Definition 5. The definition of a re-deployment plan occurs when: (1) a transition triggered, the actions associated with this transition are executed and (2) no transitions are triggered but an SLO is about to be violated. In the second case, an objective analysis is performed and a new state and/or transition is(are) added to the state machine. The new state is created based on the resources impacted, its current state, and SLOs. This component stores in the Knowledge component validated state machines and updates them with new states and/or transitions.

- *Deployment* This component is associated with the Execute function of the MAPE-K loop. The functions of this component are to deploy and adapt resources according to (re-)deployment plans defined by the Adaptation & Validation(Plan Collector module in Fig. 5). It communicates with cloud service providers through their proprietary APIs (Cloud API Connector module in Fig. 5) and can perform the following actions on resources [10](Plan Executor module in Fig. 5): *Create*, *Start*, *Scale-up/down*, *Stop* and *Delete*. These actions are generic and can be applied to any kind of resource. This component logs in the Knowledge component the trace of its performed actions.
- *Monitoring* This component is associated with the Monitor function of the MAPE-K loop. In this approach, we consider three sources of logs to enforce a multi-cloud SLA: (1) the orchestrator, (2) the cloud service providers, and (3) the cloud resources. These three sources of logs allow us to consider the entire SLA enforcement process. We denote as *SLA enforcement process* the flow of events composing the (re-)deployment of a resource: (1) the orchestrator requests a resource, (2) the cloud service provider deploys these resources, (3) the resources are started. It collects event logs (Log Collector module in Fig. 5), it filters events in the logs that are not necessary for the enforcement of multi-cloud SLAs (Log Filter module in Fig. 5) and transforms the log formats (Log Transformer module in Fig. 5), if they are heterogeneous, to uniform them. Then, the collected logs are stored in the Knowledge component.
- *Policy Enforcement* This component is associated with the Analyze function of the MAPE-K loop. It is used to trigger transitions in state machines. We use a rule engine to detect when an event (Definition. 4) occurs. Events are converted into rules in the latter rule engine. These rules are used to check the logs and detect whether an event has occurred. Logs and rules are collected from the Knowledge component using, respectively, the *Log Collector* and *Rules Collector* modules. Then, the Adaptation & Validation component is notified to trigger a transition or to adapt the state machine to avoid an SLO violation.
- *Knowledge* Knowledge is a shared component between all the components of the orchestrator which aggregates all data used/generated by the orchestrator's components. There are two categories of data, *orchestration related*, and *monitoring related*. *Orchestrator related* regroups all necessary data for orchestration such as the enriched multi-cloud SLA. *Monitoring Related* contains all collected logs from resources, cloud service providers, and the orchestrator itself. The orchestrator components interact with Knowledge via an API to request and store data.

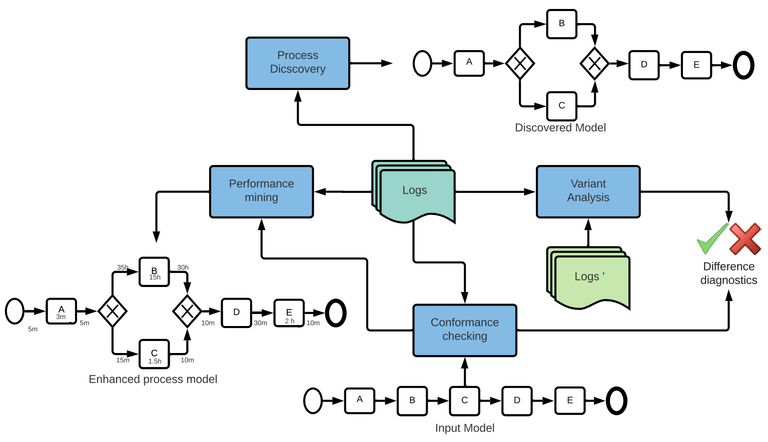


Fig. 6 Process mining techniques categories (inspired from [38])

After deploying the multi-cloud application's required resources, our approach verifies that the execution has taken place according to the defined multi-cloud SLA and identifies potential violations. This is addressed as part of our approach's third step is presented in the next section.

8 Multi-cloud SLA reporting

This section depicts how we leverage process mining to ensure SLA reporting. In process management, process mining techniques support the analysis of operational processes based on a process model and event logs generated by its execution [21]. As shown in our approach overview (Fig. 1), we consider a *multi-cloud SLA enforcement model* and *collected logs* as inputs for a process mining technique to ensure multi-cloud SLA reporting. In the following, we briefly introduce process mining and especially conformance checking techniques in Sect. 8.1 then we detail our process mining based multi-cloud SLA reporting approach in Sect. 8.2.

8.1 Process mining

Process mining can be viewed as a link between data science and business process management. Process mining techniques aim to discover, monitor, and improve operational processes by extracting knowledge from event logs [21]. An event log stores data about the occurrence of activities that were recorded by information systems while supporting the execution of a process. There are four categories of process mining techniques as depicted in Fig. 6:

- *Process discovery* techniques produce a process model from a given set of event logs.

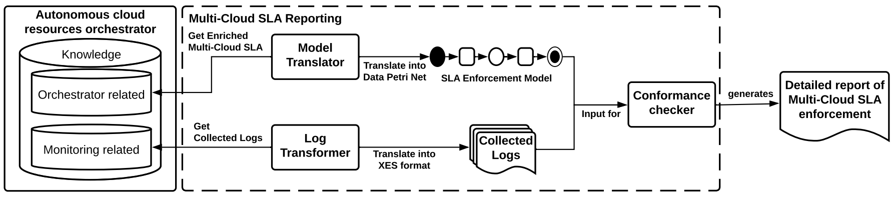

Autonomous cloud resources orchestrator
Knowledge
Orchestrator related
Monitoring related
Multi-Cloud SLA Reporting
Get Enriched Multi-Cloud SLA
Model Translator
Translate into Data Petri Net
SLA Enforcement Model
Get Collected Logs
Log Transformer
Translate into XES format
Collected Logs
Input for
Conformance checker
generates
Detailed report of Multi-Cloud SLA enforcement

Fig. 7 Multi-Cloud SLA reporting approach overview

- *Performance mining* techniques produce an enhanced process model from an input model and an event log. These techniques permit us to understand the process behavior and identify issues such as bottlenecks.
- *Variant analysis* techniques produce different diagnostics from two different event logs. These techniques generally compare event logs containing all cases that end positively with event logs containing all cases that end negatively.
- *Conformance checking* techniques produce different diagnostics between a process model and an event log to find commonalities and discrepancies. These techniques permit to compute conformance measures, i.e., find the level of conformance between process models and event logs, i.e., the executions of the models.

8.2 Conformance checking for multi-cloud SLA reporting

In this section, we present our proposed approach for multi-cloud SLA enforcement reporting using conformance checking techniques. Our main objective is to analyze what happened during multi-cloud SLA enforcement and investigate its conformity to the agreed-on model. We rely on conformance checking techniques to compare an *SLA enforcement model* and logs collected during this model enforcement. An overview of this approach is depicted in Fig. 7. As shown in Fig. 7, we retrieve these input data, i.e., the enriched multi-cloud SLA and the collected logs, from the knowledge component of the autonomous cloud resource orchestrator (Sect. 7). Then, we translate the enriched multi-cloud SLA into an SLA enforcement model represented as a Petri net with data (the Model Translator component, Fig. 7). This latter model translation is performed to take advantage of existing conformance checking techniques. We also transform the collected data into the standard XES format for event logs representation [50] (the Log Transformer component, Fig. 7). Finally, the conformance checker component takes the latter data as input to perform the compliance checking and generates a detailed report of the multi-cloud SLA enforcement. The following sections details these different components.

Table 8 Translation rules between *SM* and *DPN-net*

State machine	DPN-Net
State ($s \in S$)	Place ($p \in P$)
Transition ($t \in T$)	Transition ($t \in T$)
Transition source and destination ($t._t$ and $t._s$)	Arc ($f \in F$)
Transition event metric ($t.E.p.m$)	Variable ($v \in V$)
Transition event predicate ($t.E.p$)	Guard ($g \in G$)

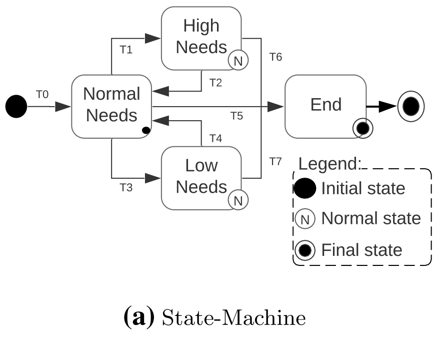
8.2.1 SLA enforcement model

A multi-cloud SLA enforcement model represents the flow of actions executed to enforce a multi-cloud SLA and is based on the enriched multi-cloud SLA. As it stands, the state machine is not the optimal format for leveraging existing conformance checking techniques. The usual format used by process mining techniques is Petri net [21]. So, we propose to translate state machines associated with multi-cloud SLAs into Petri nets with data (Model translator component). A Petri net with data (Definition.9) allows formally representing the multi-cloud SLA enforcement process. Specifically, we use Petri net with data, instead of traditional Petri nets, to represent the SLOs of a multi-cloud SLA in its data perspective, since Petri nets allow only representing the control-flow perspective of an enforcement process. This representation also allows us to use existing process mining techniques.

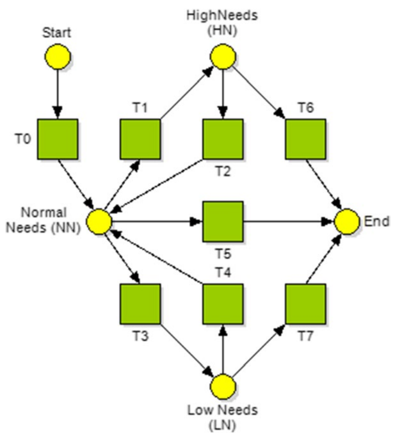
Definition 9 (Petri Net with data) A Petri Net with data *DPN-net* = (P, T, F, V, U, R, W, G) is composed of: a Petri net (P, T, F) where P is a set of places; T is a set of transitions; F is the set of flow relations describing the arcs between places and transitions; a set V of variables; a function U that defines the values admissible for each variable; a read function R that labels each transition with the set of variables that it must read; a write function W that labels each transition with the set of variables that it must write, and a guard function G that associates a guard with each transition.

We translate the state machine (SM) associated with the enriched multi-cloud SLA into a Petri Net with data (DPN-Net). This state machine is composed of states (Definition. 6) and transitions (Definition 7). This translation is based on the translation rules depicted in Table 8. As defined in this latter, a state in a *SM* is associated with a place in a DPN-net, a transition in a *SM* is associated with a transition in a DPN-net, the link between a place and a transition in a *SM* is represented by an arc in a DPN-net, transition event metrics in a *SM* correspond to variables in a DPN-net, e.g., CPU usage, and transition event predicates in a *SM* correspond to guards in DPN-net.

To illustrate this translation, we consider the UI sub-SLA state machine of our motivating example represented in Fig. 8a. We translate this latter state machine into a Petri net with data using our translation rules (Table 8). A graphical representation



(a) State-Machine



(b) Petri Net with Data

Fig. 8 UI sub-SLA

of the translated Petri net with data in Yasper[6] is given in Fig. 8b and its representation according to the definition. 9 in Table 9.

8.2.2 Collected event logs

In this section, we discuss the collected logs that are necessary to apply a conformance checking technique and their format. These logs represent "what happened" during the enforcement of a multi-cloud SLA. It is necessary to define a standard format for event logs in our heterogeneous multi-cloud context. In a business process management context, an event log contains events associated with cases of a business process model. Cases are associated with collections of events, i.e., traces. Each event describes an execution step in the business process case [51]. In our context, an event log contains events related to an instance of an SLA enforcement model (Sect. 8.2.1). We refer to these logs as *collected logs* which are retrieved from the knowledge component. To ensure the quality of these logs, we rely on the data quality definition proposed by Van der Aalst et al. in [21]. In this work, the mandatory event attributes are defined that a log should contain as *case*, *activity*, and *timestamp* or *position*. *Case* is defined by a sequence of events, *activity*, aka task in a business process, that identifies the event, and *timestamp* or *position* ordering the events. In our approach, we define the log format as composed of:

- *Timestamp* specifying the event execution time,
- *Source* denoting the event originator: *orchestrator*, *CSP*, or *Resource*,
- *States/Transition* (*S/T*) denoting the state-machine's state or transition associated to the event,

[6] Yasper is a tool for Petri net representation, https://www.yasper.org/.

Table 9 UI sub-SLA Petri net with data definition

Transition(t)	Src(p)	Dest(p)	Guards(g)
(a) Definition			
T0	Start	NN	–
T1	NN	HN	cpuusage > 85%
T2	HN	NN	cpuusage < 30%
T3	NN	LN	cpuusage < 30%
T4	LN	NN	cpuusage > 85%
T5	NN	End	timestamp == 9 : 00 pm
T6	HN	End	timestamp == 9 : 00 pm
T7	LN	End	timestamp == 9 : 00 pm

Variable(v)	Type
(b) Variables	
cpuusage	Integer
timestamp	Time format

Table 10 A fragment of UI sub state-machine collected logs

Time- stamp	Source	S/T	Resource provider	Resource name	Event type	Metric	Value
00:00	ORCH	T0	AWS	VM1	Create	–	–
00:03	CSP	T0	AWS	VM1	Create	–	–
00:05	RES	NN	AWS	VM1	R-Usage	cpu	15%
00:35	RES	NN	AWS	VM1	R-Usage	cpu	5%
01:05	RES	NN	AWS	VM1	R-Usage	cpu	95%
01:35	RES	NN	AWS	VM1	R-Usage	cpu	95%
02:05	RES	NN	AWS	VM1	R-Usage	cpu	95%
02:06	ORCH	T1	AWS	VM1	Scale-out	–	–
02:08	RES	HN	AWS	VM1	R-Usage	cpu	15%

- *ResourceProvider* denoting the name of the cloud service provider that provides the related resource,
- *ResourceName* is the resource name,
- *EventType* defines the action or event type as presented in Definitions 4, 5,
- *Metric* provides the event-related metric such as *cpuusage*, and
- *Value* corresponds to the metric value.

To illustrate the considered log format, we present an example of a fragment of a UI sub state machine associated collected logs. These logs represent the following execution of the state machine: transition $T0$ the resource $VM1$ is created (Timestamp: $00:00$ to $00:03$), then the logs generated by the resource $VM1$ at the *NormalNeeds* (*NN*) state (Timestamp: $00:05$ to $02:05$), and finally the transition $T1$ triggered by the CPU exceeding 85%for a minute and a state transition to the *HighNeeds* (*HN*) state (Timestamp: $02:06$ to $02:08$). In addition, multiple *events* from different sources, e.g., *Orchestrator*(*ORCH*), *CSP*, *resources*(*RES*) can be related to the same *state or transition* [52], e.g., event with timestamp 00:00 and 00:03 related to transition T0 in Table 10.

8.2.3 Multi-cloud SLA enforcement conformance checking

In this section, we describe the conformance checking technique [53], that we use for multi-cloud SLA reporting. Conformance checking is a set of techniques that compare an existing process model with an event log of the same process [54]. Two families of conformance checking techniques exist: *token replay* and *alignments*. *Token-Replay* is a heuristic technique, which uses four counters (produced tokens, consumed tokens, missing tokens, and remaining tokens) to compute the fitness of a trace with a given model. Although this technique is easy to understand and can be implemented efficiently, since *token-replay* takes a local decision, it may lead to misleading results. *Alignment* is a technique, which performs an exhaustive search to find out the *optimal alignment* between an observed trace and a process model. Hence, it returns the closest model execution to the trace.

We consider alignment techniques in order to set up our conformance checking approach for multi-cloud SLA reporting. In our context, an alignment technique allows identifying an alignment between an SLA enforcement model and its associated collected logs. In other words, the events in the event log need to be associated with model elements and vice versa. An alignment relates *"moves" in collected logs* to *"moves" in an SLA enforcement model* in order to define an alignment between them. This is not a straightforward task because the log can deviate from the model and not all events may have been modeled or recorded. These moves, denoted as *Legal moves*, represent possible alignment moves and are categorized as follows [51]: a move in **log** only, a move in **model** only, a move in **both with incorrect variables**, and a move in **both with correct variables**. Moves belonging to this last category are called **synchronous** and the others **non-synchronous**. An alignment is considered as *complete* when for each event in its log trace an alignment exists. Any non-synchronous *move* is considered a deviation, i.e., when a move in the logs cannot be related to a move in the model. In order to define the severity of a deviation, a cost is associated with each legal move. This last cost is defined by the user in order to weigh the deviations. Since several *complete* alignments can exist, an *optimal* alignment is a *complete* alignment with minimal cost. An alignment cost is defined as the sum of the costs of all its moves.

In our work, we use an alignment technique to align collected logs stored in the knowledge component along with a multi-cloud SLA enforcement model. This allows us to evaluate the adequacy level between them and to identify SLA violations. We adopt the conformance checking technique proposed by de Leoni et van der Aalst in [55]. This technique consists of finding an optimal alignment considering a DPN-net and a set of logs as input. We present the pseudo-code of this alignment technique in Algorithm 1 and we refer readers to [55] for a detailed description.

This alignment technique is composed of 3 phases: (1) initially, build an alignment between the Petri Net and the log trace taking into consideration only the control-flow perspective (line 3 in Alg. 1). This alignment is performed using the *A* algorithm* as proposed in [56]. (2) Enrich the firings of transitions with the write operations defined in the DPN-net (lines 4–5). The search for suitable write operations is formulated as the solution of an integer linear programming problem. And, (3) compute the fitness value considering the data perspective (lines 6–7). The latter fitness value is computed (Eq. 1) as the comparison of the cost of the current alignment to the cost of the worst possible alignment, i.e., all non-synchronous moves, thus determining its adequacy level.

$$fitnessValue = 1 - \frac{\text{AlignCost}}{\text{WorstAlignCost}} \tag{1}$$

Table 11 Alignment excerpt with collected logs and SLA enforcement model

Move position	1	2	3	4	5	6
Collected logs	*T*0	*T*0	*NN* {CpuU = 95%}	*T*1	>>	*HN* {CpuU = 5%}
SLA enforcement model	*T*0	*T*0	*NN* {CpuU = 95%}	*T*1	*T*1	*HN* {CpuU = 5%}

Algorithm 1 Pseudo-code of [55] alignment technique

```
1: Input Set of collected logs and a DPN-net
2: Output Alignment and associated $fitnessvalue$
                                ▷ $(i)$ Build control-flow perspective alignment
3: Apply $A*$ algorithm to build the alignment [56]
                                ▷ $(ii)$ Enrich alignment with the firings of transitions
4: Enrich firing transitions with write operations
5: Find write operations formulated as an ILP problem
                                ▷ $(iii)$ Compute the fitness value
6: Compute alignment cost
7: Compare with the worst possible cost alignment (Equation: 1)
```

The output of this alignment and its associated fitness value constitute the *detailed report of multi-cloud SLA enforcement* as depicted in Fig. 8b. This report is provided to a cloud consumer to eventually enhance its multi-cloud SLA. This data could also be stored in the knowledge component to autonomously enhance the SLA enforcement model by the orchestrator, but this use case is out of the scope of this work.

An example of alignment between the SLA enforcement model associated with the UI sub state machine (Fig. 8b) and the sample collected logs of Table 10 is represented in Table 11. As stated before, in addition to the control flow perspective [phase (1)], this alignment considers the data perspective by checking the correctness of reading and write operations on variables [phase (2)], e.g., the *cpu_Usage* read operation between braces {} in moves 3 and 6 (Table 11). For instance, in this example, the four first moves are synchronous while the fifth move is an enforcement model move only since no associate move in the log was identified (represented by >>).

After building this alignment, the fitness value is computed (phase (2)) using the following cost function(κ):

$$\kappa = \begin{cases} 1, \text{ if synchronous move with correct variables} \\ 2, \text{ if synchronous move with incorrect variables} \\ 5, \text{ if log or process move only} \end{cases}$$

The built alignment (Table. 11) contains 5 **synchronous moves** and 1 **model only** move. The cost of this latter alignment is as follows: $AlignCost = 5 \times 1 + 1 \times 5$.

This alignment is composed of six moves, so the worst possible alignment is six **log/model** moves only: *WorstAlignCost* $= 6 \times 5 = 30$. Hence, the fitness value is computed as follows:

$$\text{fitnessValue} = 1 - \frac{\text{AlignCost}}{\text{WorstAlignCost}} = 1 - \frac{10}{30} = 0.777$$

9 Implementation and experiments

In the following, we discuss the implementation of the three steps of our approach: multi-cloud SLA representation, multi-cloud SLA Enforcement, and multi-cloud SLA reporting. In the last section, we experiment our approach with a use case. The source code of the developed prototypes is available online at https://framagit.org/JeremyMechouche/multi-cloud-orchestrator.

9.1 Multi-cloud SLA representation

We model our enriched multi-cloud SLA model based on the common cloud SLA buildings blocks defined in ISO-19086 [14, 15] using the eclipse modeling framework (EMF)[7]. EMF permits to (1) formally define the multi-cloud SLA semantics, (2) validate an SLA accordingly, and (3) translate other SLA models into our format.

According to our model, a multi-cloud SLA is composed of one global-SLA and at least one sub-SLA. An SLA is composed of *covered services* which describes the agreed-on cloud service described with the cloud resource description model (cRDM) proposed by our team [49]. Each covered service in this model is associated with a set of *Service Objectives* (Definition. 1) expressing the service level objectives of the SLA. Our model associates also a state machine (Sect. 6) describing the multi-cloud application dynamicity to a multi-cloud SLA. Our EMF-based prototype allows defining multi-cloud SLAs, using the YAML syntax. The reason behind the choice of YAML consists of readability, ease of use compared with other syntaxes such as XML, and its wide use in the cloud domain [26]. A detailed description of our model, its graphical representation, and the source code of our prototype are at https://framagit.org/JeremyMechouche/multi-cloud-orchestrator.

9.2 Multi-cloud SLA enforcement

We developed a prototype of an autonomous orchestrator according to the architecture proposed in Sect. 7, following the micro-services paradigm and supported by docker containers [57]. Figure 9 illustrates the *autonomous cloud resources orchestrator prototype*. Its main components are: **monitoring**, **knowledge**, **policy**

[7] https://www.eclipse.org/modeling/emf.

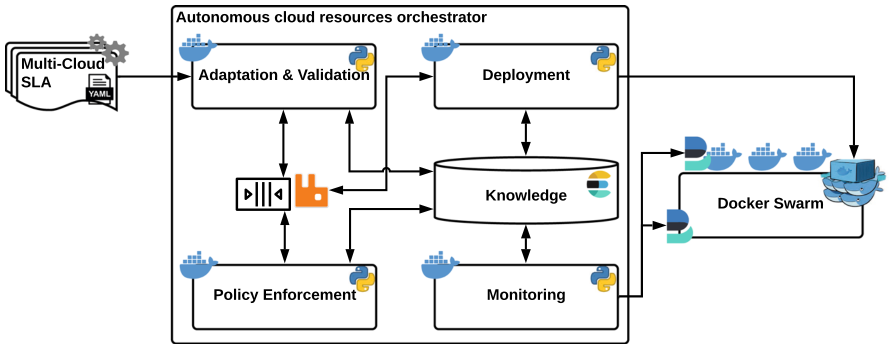
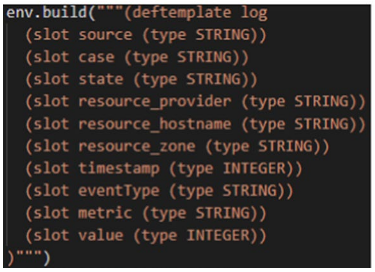


Fig. 9 Autonomous cloud resources orchestrator prototype

enforcement, **adaptation & validation** and **deployment**. We implemented the different components of our orchestrator in python[8] and we used docker[9] to deploy them. The intra-components communication is ensured through the RabbitMQ[10] message-queue. We have used a dedicated RabbitMQ server hosted in a dedicated docker container. The orchestrator deploys simulated resources as services using docker swarm, the cluster orchestrator of docker. The above solution permits simulating cloud resources and all their lifecycle management operations (e.g., create, scale, and delete), and hence allows reproducing the interaction between the orchestrator and cloud resources. The micro-service-based architecture of our prototype allows easily extending it to consider deployment with public CSPs.

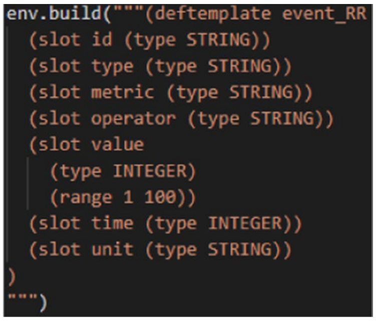
Monitoring and Knowledge We implemented the monitoring component in python. We used the elastic stack[11] (ELK) tools to store data consisting of the following modules: (1) *elasticsearch* a search engine which can contain all logs event from our three sources (i.e., orchestrator, cloud service provider, and resources) and support the knowledge component, (2) *logstash* permits to convert logs into the required format for the third step of our approach (i.e., conformance checking), and (3) *beats* a series of data shippers compatible with many data sources. The elasticsearch module implements the knowledge component of our autonomous architecture. The *Logstash* and *beats* modules are associated with the monitoring component. The latter component interacts with the knowledge component through an elastic python library, provided by the elastic development team, in order to retrieve and process data. Then, the data is filtered according to its source and sent to the policy enforcement component.
Policy Enforcement The policy enforcement component is based on CLIPS [12], a tool for building expert systems. This tool has been adapted into a python library

8 https://www.python.org/.
9 https://www.docker.com/.
10 https://www.rabbitmq.com/.
11 https://www.elastic.co/en/elastic-stack/.
12 http://www.clipsrules.net/.



```
env.build("""(deftemplate log
   (slot source (type STRING))
   (slot case (type STRING))
   (slot state (type STRING))
   (slot resource_provider (type STRING))
   (slot resource_hostname (type STRING))
   (slot resource_zone (type STRING))
   (slot timestamp (type INTEGER))
   (slot eventType (type STRING))
   (slot metric (type STRING))
   (slot value (type INTEGER))
)""")
```

(a) Fact template of log



```
env.build("""(deftemplate event_RR
   (slot id (type STRING))
   (slot type (type STRING))
   (slot metric (type STRING))
   (slot operator (type STRING))
   (slot value
      (type INTEGER)
      (range 1 100))
   (slot time (type INTEGER))
   (slot unit (type STRING))
)
""")
```

(b) Rule template of event

Fig. 10 CLIPS templates

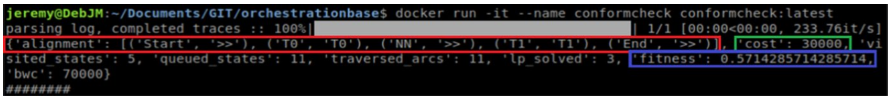
CLIPSPy[13] that brings the capabilities of CLIPS within the Python ecosystem. There are two element categories in CLIPS and expert systems in general: *facts* and *rules*. In our context, a *fact* is a monitoring log and a *rule* is an event triggering a transition. We have defined a template of fact for representing a log as depicted in Fig. 10a. A log is composed of the different elements as described in Sect.8.2.2. We also defined a template of rules for representing an event as depicted in Fig. 10b. An event is composed as defined in the Definition 4. Then, we use these templates to instantiate events composing the state machine and the collected logs. Finally, CLIPS validate if the rules are triggered by the log and if so, notify the other components of the prototype.

Adaptation and validation The adaptation & validation component purpose is twofold: (1) validate the verification of state machine represented in SLAs at design time and (2) propose at run-time reconfiguration for resources near to violate its SLA by triggering a transition, defining new states, or new transitions. These functionalities are implemented in python. The first functionality is implemented following the proposed algorithm in our previous work [19]. The second functionality consists of performing a validation of the service level objectives compared to the actual state of running resources. This component defined plan for the deployment component to enforce according to the information provided by the policy enforcement component. This latter plan is stored on the elasticsearch module.

Deployment and Docker Swarm The deployment component interacts directly with service providers. The input for this component is the multi-cloud SLA and the plan defined by Adaptation & validation component. These two input data are stored in the elasticsearch component as orchestrator-related data. In this implementation, the deployment component interacts with docker swarm[14] acting as a resource provider. In swarm terminology, a *service* is a composition of contain-

[13] https://github.com/noxdafox/clipspy.

[14] https://docs.docker.com/engine/swarm/.



```
jeremy@DebJM:~/Documents/GIT/orchestrationbase$ docker run -it --name conformcheck conformcheck:latest
parsing log, completed traces :: 100%|                                        | 1/1 [00:00<00:00, 233.76it/s]
{'alignment': [('Start', '>>'), ('T0', 'T0'), ('NN', '>>'), ('T1', 'T1'), ('End', '>>')], 'cost': 30000, 'vi
sited_states': 5, 'queued_states': 11, 'traversed_arcs': 11, 'lp_solved': 3, 'fitness': 0.5714285714285714,
'bwc': 70000}
########
```

Fig. 11 Execution of conformance checking algorithm

ers that can be scaled. It provides all capabilities of a cloud resource we need to test our approach. However, the implementation has been intended to be modular and can interact with API of major cloud service providers such as Google Cloud Platform[15] or Amazon Web Services[16].

9.3 SLA reporting with conformance checking

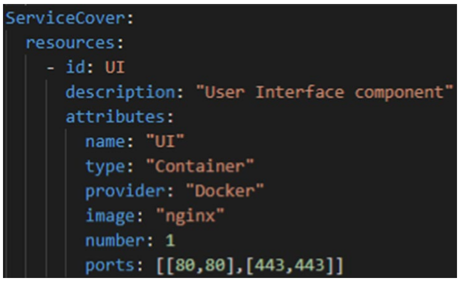
The final step of our approach is performed using a conformance checking technique. We implement a prototype according to our approach presented in Fig. 7. Usually, process mining is performed with *Prom* [58] the de-facto open-source software for process mining, but in our case, we need an automatic interaction. In this aim, we use a python library, *pm4py* [59], for leveraging conformance checking techniques. As described in Sect. 8, we perform (1) a model translation from the state machine representing multi-cloud SLA dynamicity into a DPN-net, (2) a transformation of the collected logs into the XES format and (3) an alignment using these data to generate a multi-cloud SLA enforcement report. We used the DPN-net model, implemented in the *pm4py* library, for the translation of our state-machine format (Model translator component in Fig. 7). We implemented this translation as described in Sect. 8.2.1. Then, we transformed the logs into the XES format using a function defined in the *pm4py* library. Finally, we have made the alignment using the algorithm defined in this same library and in line with our proposed approach (Sect 8.2.3). Figure 11 depicts the result of conformance checking execution using our prototype. For the sake of simplicity, we considered for this example an SLA enforcement model with three states (*Start*, *NN*, and *End*) and define a simple log with missing states to generate deviations. The output of this execution is composed of: the alignment (framed in red in Fig. 11), the alignment cost (framed in green), and the fitness value (framed in blue). In this alignment, we observe three model moves only.

9.4 Experiments

In this section, we are experimenting with our approach for multi-cloud SLA enforcement reporting with a multi-cloud application as a use case. The testbed for

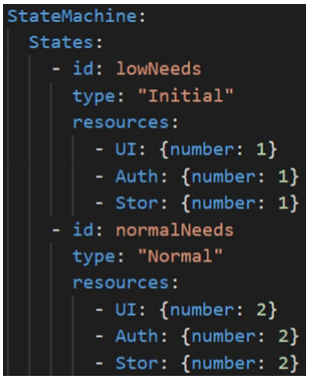
[15] https://cloud.google.com/.

[16] https://aws.amazon.com/fr/.



```
ServiceCover:
  resources:
    - id: UI
      description: "User Interface component"
      attributes:
        name: "UI"
        type: "Container"
        provider: "Docker"
        image: "nginx"
        number: 1
        ports: [[80,80],[443,443]]
```

Fig. 12 **User Interface** resource description



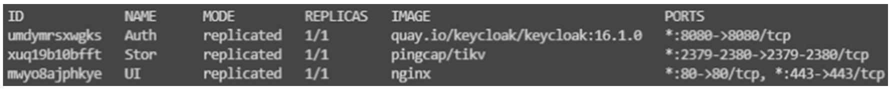
```
StateMachine:
  States:
    - id: lowNeeds
      type: "Initial"
      resources:
        - UI: {number: 1}
        - Auth: {number: 1}
        - Stor: {number: 1}
    - id: normalNeeds
      type: "Normal"
      resources:
        - UI: {number: 2}
        - Auth: {number: 2}
        - Stor: {number: 2}
```

Fig. 13 *lowNeeds* and *normalNeeds* states of the state machine

these experiments consists of three docker swarm services that represent the multi-cloud application described in our motivating example (Sect 3). The first service represents the **User Interface** component (*UI*) deployed as a *nginx*[17] container. *Nginx* is a widely-used open source web server application. The second service represents the **Authentication** component (*Auth*) deployed as a *keycloak*[18] container. *Keycloak* is an open-source identity and access management solution. The last service represents the **Storage** component (*Stor*) deployed as a *TiKV*[19] container. *TiKV* is an open-source, distributed, and transactional key-value database. Our experimentation is composed of three steps: (1) representing the multi-cloud SLA following the previously defined multi-cloud application, (2) enforcing the multi-cloud SLA and stressing the application, and finally (3) reporting its enforcement to check the conformance with the representation.

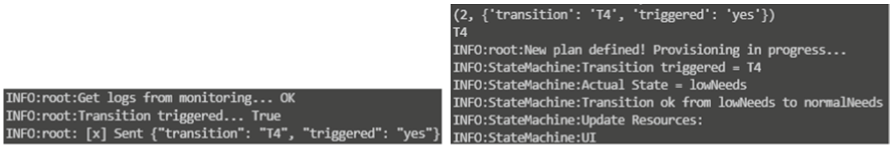
Multi-Cloud SLA Representation To verify the capacity of our approach for representing multi-cloud SLA, we first represent the covered services and state machines of the multi-cloud application use case. To do so, we represent the multi-cloud SLA in YAML using our EMF-based SLA representation prototype (Sect. 9.1). Figure 12 illustrates an excerpt of the multi-cloud SLA. This excerpt presents the resources composing the User Interface component, which is docker

[17] https://www.nginx.com/.
[18] https://www.keycloak.org/.
[19] https://tikv.org/.



```
ID            NAME    MODE        REPLICAS   IMAGE                               PORTS
umdymrsxwgks  Auth    replicated  1/1        quay.io/keycloak/keycloak:16.1.0    *:8080->8080/tcp
xuq19b10bfft  Stor    replicated  1/1        pingcap/tikv                        *:2379-2380->2379-2380/tcp
mwyo8ajphkye  UI      replicated  1/1        nginx                               *:80->80/tcp, *:443->443/tcp
```

Fig. 14 Low needs state



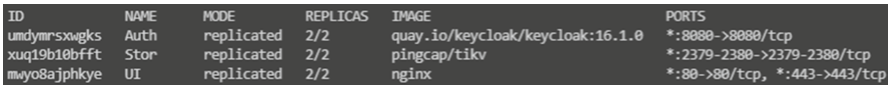
```
INFO:root:Get logs from monitoring... OK
INFO:root:Transition triggered... True
INFO:root: [x] Sent {"transition": "T4", "triggered": "yes"}
```

(a) Policy Enforcement

```
(2, {'transition': 'T4', 'triggered': 'yes'})
T4
INFO:root:New plan defined! Provisioning in progress...
INFO:StateMachine:Transition triggered = T4
INFO:StateMachine:Actual State = lowNeeds
INFO:StateMachine:Transition ok from lowNeeds to normalNeeds
INFO:StateMachine:Update Resources:
INFO:StateMachine:UI
```

(b) Deployment

Fig. 15 Cycle of a transition detection



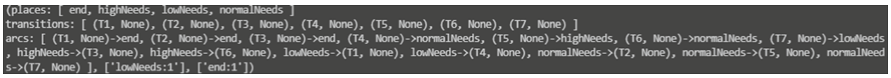
```
ID            NAME    MODE        REPLICAS   IMAGE                               PORTS
umdymrsxwgks  Auth    replicated  2/2        quay.io/keycloak/keycloak:16.1.0    *:8080->8080/tcp
xuq19b10bfft  Stor    replicated  2/2        pingcap/tikv                        *:2379-2380->2379-2380/tcp
mwyo8ajphkye  UI      replicated  2/2        nginx                               *:80->80/tcp, *:443->443/tcp
```

Fig. 16 Normal needs state

containers using *Nginx* image and listening on port 80 and 443. We consider a state machine with four states: *lowNeeds*, *normalNeeds*, *highNeeds* and *end*. This latter state machine represents the dynamicity required to handle the peak of activities of the application. An excerpt of this state machine is represented in Fig. 13.

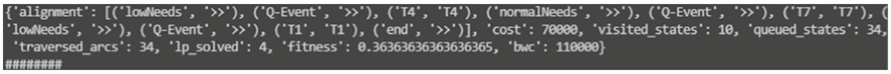
Multi-Cloud SLA Enforcement For this second step, we validate the enforcement capability of our approach and that the transition between states is well executed. To do so, we enforce the state machine represented previously using our autonomous orchestrator prototype. The prototype takes as input the multi-cloud SLA. Then, this latter prototype validates the state machine in the adaptation & validation component. Next, the plan for deploying the initial state of our state machine is sent to the deployment component. We validate that the initial state, *lowNeeds*, is deployed as depicted in Fig. 14. We can observe that each service is composed of one container(*replicas*). The monitoring component now monitors the deployed resources and sends any logs received to the policy enforcement component. In order to validate the performance of the state machine, we need to stress the multi-cloud application. We use the *apache ab* tool[20] which will generate HTTP requests to the User Interface component. The **policy enforcement** component will detect the increase in CPU usage and trigger a transition from the *lowNeeds* state to the *normalNeeds* state and scale-out resources. The **policy enforcement** component triggers the transition as shown in Fig. 15a. The latter

[20] https://httpd.apache.org/docs/2.4/programs/ab.html.



```
(places: [ end, highNeeds, lowNeeds, normalNeeds ]
transitions: [ (T1, None), (T2, None), (T3, None), (T4, None), (T5, None), (T6, None), (T7, None) ]
arcs: [ (T1, None)->end, (T2, None)->end, (T3, None)->end, (T4, None)->normalNeeds, (T5, None)->highNeeds, (T6, None)->normalNeeds, (T7, None)->lowNeeds
, highNeeds->(T3, None), highNeeds->(T6, None), lowNeeds->(T1, None), lowNeeds->(T4, None), normalNeeds->(T2, None), normalNeeds->(T5, None), normalNeed
s->(T7, None) ], ['lowNeeds:1'], ['end:1'])
```

Fig. 17 Output of the data Petri Net translator



```
{'alignment': [('lowNeeds', '>>'), ('Q-Event', '>>'), ('T4', 'T4'), ('normalNeeds', '>>'), ('Q-Event', '>>'), ('T7', 'T7'), (
'lowNeeds', '>>'), ('Q-Event', '>>'), ('T1', 'T1'), ('end', '>>')], 'cost': 70000, 'visited_states': 10, 'queued_states': 34,
 'traversed_arcs': 34, 'lp_solved': 4, 'fitness': 0.36363636363636365, 'bwc': 110000}
########
```

Fig. 18 Conformance checking results of experiment

component then notifies the others that this transition has been triggered. Upon receiving this notification, the **deployment** component deploys the new resources required for the *normalNeeds* state, as shown in Fig. 15b. Once the deployment is complete, each service is scaled-out to two containers, as depicted in Fig. 16. At the end of the SLA validity period, a transition is triggered to the final state and all resources are de-provisioned.

Multi-Cloud SLA Reporting Finally, when the enforcement is complete, we report the multi-cloud SLA using the conformance checker component (Fig. 7) in order to validate the enforcement. Inputs for the conformance checker are an *SLA enforcement model* and *collected logs*. We implement an algorithm for translating our state machine format into a data Petri Net. The state machine is retrieved from the orchestrator-related data of the knowledge component. Figure 17 presents the model produced by our algorithm. This latter model shows the places, transitions and arcs composing the Data Petri Net. The *collected logs* are retrieved from the Monitoring related data of the knowledge component. These are merged and then converted into XES format. Finally, the conformance checking is performed between the SLA enforcement model and the collected logs, as described in Sect. 9.3. Figure 18 depicts the result of the conformance check of this experiment.

10 Conclusion

Maintaining the service level of a multi-cloud application is not a trivial task due to the dynamicity and heterogeneity of a multi-cloud context. In this article, we presented an approach considering two challenges for managing and adapting the multi-cloud SLA: considering and managing dynamicity in the representation of multi-cloud application requirements, and reporting what happened during SLA enforcement to identify violations and act accordingly. We propose (1) a hierarchical representation of multi-cloud SLAs: sub-SLAs associated with a system's components deployed on distinct cloud service providers and global-SLA associated with the whole system. We also enrich these SLA representations with state machines reflecting reconfiguration strategies defined by cloud customers. Then, we propose

(2) an autonomous multi-cloud resource orchestrator based on the MAPE-K adaptation control loop to enforce them and to avoid SLA violations. Finally, in order to check the conformity of this enforcement with defined multi-cloud SLA, we propose (3) an approach for multi-cloud SLA reporting leveraging conformance checking techniques. As a proof of concept, we implemented and experimented our approach to validate its feasibility. The prototypes have been implemented using Docker and Docker swarm. However, this article is limited to a declarative approach of resources and does not consider the scheduling of resources. We also do not consider the interconnection between resources across different cloud service providers. In terms of the short-term perspective for this work, we plan to adjust our implemented prototypes to use real cloud service providers instead of simulated resources. Later, we also plan to optimize the interactions of global on sub state machine. Next, we aim to consider online conformance checking techniques. Indeed, such techniques allow validating, and adapting if necessary, in real-time multi-cloud SLA enforcement. We also would like to consider other process mining techniques in order to enhance multi-cloud SLAs based on the conformance checking technique's output.

References

1. Comparing multicloud management and governance approaches. https://www.gartner.com/en/documents/3903683/comparing-multicloud-management-and-governance-approache. Accessed: 2021-03-26
2. State of the cloud report from flexera
3. MELODIC: multi-cloud management platform (2020). https://h2020.melodic.cloud/
4. DECIDE: multicloud application towards the digital single market (2020). https://decide-h2020.eu/
5. Cyclone (2020). https://www.cyclone-project.eu//
6. Ardagna D et al. (2012) MODAClouds: a model-driven approach for the design and execution of applications on multiple Clouds. In: 2012 4th International Workshop on Modeling in Software Engineering (MISE) . https://ieeexplore.ieee.org/document/6226014/
7. Brogi A et al (2015) Adaptive management of applications across multiple clouds: the SeaClouds approach. CLEI Electron J 18:2
8. Farokhi S, Jrad F, Brandic I, Streit A (2014) Hierarchical SLA-based service selection for multicloud environments. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science
9. Mell PM, Grance T (2011) SP 800–145. The NIST definition of cloud computing. Tech, Rep, NIST, Gaithersburg, MD, USA
10. Tomarchio O, Calcaterra D, Di Modica G (2020) Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. J Cloud Comput 9:1–24
11. Horn PJ (2001) Autonomic Computing: IBM's Perspective on the State of Information Technology. Tech, Rep, IBM
12. Faniyi F, Bahsoon R (2015) A systematic review of service level management in the cloud. ACM Comput Surv 48(3):1–27
13. Maarouf A, Marzouk A, Haqiq A (2015) A review of SLA specification languages in the cloud computing. SITA 1–6
14. ISO (2016) Information technology: Cloud computing-Service level agreement (SLA) framework - Part 1: Overview and concepts. Standard, International Organization for Standardization
15. ISO (2018) Information technology: Cloud computing-Service level agreement (SLA) framework - Part 2: Metric model. Standard, International Organization for Standardization
16. Ramalingam C, Mohan P (2021) Addressing semantics standards for cloud portability and interoperability in multi cloud environment. Symmetry 13(2):317

17. Taha A, Manzoor S, Suri N (2017) SLA-based service selection for multi-cloud environments. In: 2017 IEEE International Conference on Edge Computing (EDGE). 65–72
18. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud computing patterns: fundamentals to design, build, and manage cloud applications. Springer, Berlin
19. Mechouche J, Touihri R, Sellami M, Gaaloul, W (2021) Towards higher-level description of SLA-aware reconfiguration strategies based on state-machine. ICEBE to appear
20. Sinreich D (2006) An architectural blueprint for autonomic computing. Tech. Rep, IBM
21. Van der Aalst WMP (2016) Process mining: data science in action, 2nd edn. Springer, Berlin
22. Mannhardt F, De Leoni M, Reijers HA, Van Der Aalst WM (2016) Balanced multi-perspective checking of process conformance. Computing 98(4):407–437
23. Keller A, Ludwig H (2003) The WSLA framework: specifying and monitoring service level agreements for web services. J Netw Syst Manag 11:57
24. Kearney KT, Torelli F, Kotsokalis C (2010) SLA*: An abstract syntax for Service Level Agreements. In: 2010 11th IEEE/ACM International Conference on Grid Computing. 217–224
25. Ludwig H et al (2015) rSLA: monitoring SLAs in dynamic service environments. In: International Conference on Service-oriented Computing. 139–153
26. Engel R, Rajamoni S, Chen B, Ludwig H, Keller A (2018) ysla: reusable and configurable SLAs for large-scale SLA management. In: 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC). 317–325
27. Uriarte RB, Tiezzi F, De Nicola R (2014) Slac: a formal service-level-agreement language for cloud computing. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. 419–426
28. Kouki Y, Ledoux T et al (2012) CSLA: a Language for Improving Cloud SLA Management. CLOSER. https://doi.org/10.5220/0003956405860591
29. Son S, Choi HH, Oh BT, Kim SW, Kim, BS (2017) Cloud SLA relationships in multi-cloud environment: models and practices. In: Proceedings of the 8th International Conference on Computer Modeling and Simulation
30. Emeakaroha VC et al (2012) Towards autonomic detection of SLA violations in Cloud infrastructures. Fut Gener Comput Syst 28(7):1017–1029
31. Mosallanejad A, Atan R (2013) HA-SLA: a hierarchical autonomic SLA model for SLA monitoring in cloud computing. J Softw Eng Appl 6(3B):114
32. Casalicchio E, Silvestri L (2013) Mechanisms for SLA provisioning in cloud-based service providers. Comput Netw 57(3):795–810
33. Ghobaei-Arani M, Jabbehdari S, Pourmina MA (2018) An autonomic resource provisioning approach for service-based cloud applications: a hybrid approach. Futur Gener Comput Syst 78:191–210
34. Sfondrini N, Motta G (2021) LISA: a lean information service architecture for SLA management in multi-cloud environments. Int J Grid Util Comput 12(2):149–158
35. Kosińska J, Zieliński K (2020) Autonomic management framework for cloud-native applications. J Grid Comput 18(4):779–796
36. Rouf Y, Mukherjee J, Litoiu M, Wigglesworth J, Mateescu R (2021) A framework for developing devops operation automation in clouds using components-off-the-shelf. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. 265–276
37. Ismail B, Khalid M, Muty N, Ong H (2014) SLA object and SLA process modelling using wsla and bpm notations towards defining a generic SLA Orchestrator framework. In: The Seventh International Conference on Dependability
38. Dumas M, Rosa ML, Mendling J, Reijers HA (2018) Fundamentals of business process management, 2nd edn. Springer, Berlin
39. Mager C (2014) Analysis of service level agreements using process mining techniques. FHWS Sci J 1:49
40. van Eck ML, Sidorova N, van der Aalst WM (2016) Discovering and exploring state-based models for multi-perspective processes. In: International Conference on Business Process Management. 142–157
41. Sutrisnowati RA et al (2015) BAB Framework: process mining on cloud. Proc Comput Sci 72:453–460
42. Chesani F, Ciampolini A, Loreti D, Mello P (2016) Map reduce autoscaling over the cloud with process mining monitoring. In: International Conference on Cloud Computing and Services Science. 109–130

43. Montali M, Maggi FM, Chesani F, Mello P, Aalst WMVD (2014) Monitoring business constraints with the event calculus. ACM Trans Intell Syst Technol 5(1):1–30
44. Acampora G, Bernardi ML, Cimitile M, Tortora G, Vitiello A (2017) A fuzzy-based autoscaling approach for process centered cloud systems. In: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE). 1–8
45. Song W et al (2017) Scientific workflow mining in clouds. IEEE Trans Parallel Distrib Syst 28(10):2979–2992
46. Calcaterra D, Cartelli V, Di Modica G, Tomarchio O (2018) Exploiting BPMN features to design a fault-aware TOSCA Orchestrator. CLOSER. 533–540
47. OMG (2013) Business Process Model and Notation (BPMN), Version 2.0.2. Tech. Rep., Object Management Group. http://www.omg.org/spec/BPMN/2.0.2
48. Azumah KK, Sørensen LT, Montella R, Kosta S (2021) Process mining-constrained scheduling in the hybrid cloud. Concurr Comput Pract Exp 33(4):e6025
49. Hayet B, Achraf M, Walid G, Boualem B (2020) Toward higher-level abstractions based on state machine for cloud resources elasticity. Inform Syst 90:101450
50. Verbeek H, Buijs JC, Van Dongen BF, Van Der Aalst WM (2010) Xes, xesame, and prom 6. In: International Conference on Advanced Information Systems Engineering. 60–75
51. De Leoni M, van der Aalst WM (2013) Data-aware process mining: discovering decisions in processes using alignments. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. 1451–1461
52. Mannhardt F, De Leoni M, Reijers HA, Van Der Aalst WM, Toussaint PJ (2016) From low-level events to activities-a pattern-based approach. In: International Conference on Business Process Management. 125–141
53. Dunzer S, Stierle M, Matzner M, Baier S (2019) Conformance checking: a state-of-the-art literature review. In: Proceedings of the 11th International Conference on Subject-oriented Business Process Management. 1–10
54. Van der Aalst W, Adriansyah A, van Dongen B (2012) Replaying history on process models for conformance checking and performance analysis. Wiley Interdiscip Rev Data Min Knowl Discov 2(2):182–192
55. De Leoni M, Van Der Aalst WM (2013) Aligning event logs and process models for multi-perspective conformance checking: an approach based on integer linear programming. In: Business Process Management. 113–129
56. Adriansyah A, van Dongen BF, van der Aalst WM (2011) Conformance checking using cost-based fitness analysis. In: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference. 55–64
57. Jaramillo D, Nguyen DV, Smart R (2016) Leveraging microservices architecture by using Docker technology. SoutheastCon 2016:1–5
58. Van Dongen BF, de Medeiros AKA, Verbeek H, Weijters A & van Der Aalst WM (2005) The ProM framework: a new era in process mining tool support. In: International Conference on Application and Theory of Petri nets. pp 444–454
59. Berti A, van Zelst SJ, van der Aalst W (2019) Process mining for python (PM4Py): bridging the gap between process-and data science. arXiv preprint arXiv:1905.06169

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.