Defining and Guaranteeing Dynamic Service Levels in Clouds

Rafael Brundo Uriarte[a,*], Rocco De Nicola[a], Vincenzo Scoca[a], Francesco Tiezzi[b]

[a] *IMT School for Advanced Studies Lucca, Italy*
[b] *University of Camerino, Italy*

Abstract

In this paper, we introduce SLAC, a SLA definition language specifically devised for clouds as a formalism to support the whole SLA lifecycle. The main novelty of the language is the possibility of capturing within the SLA the dynamic aspects of the environment by defining the conditions and actions to change service levels at runtime. SLAC permits to make the most of cloud elasticity, reduces the need for renegotiation and provides guarantees for dynamic scenarios. The language has formal syntax and semantics, and it comes with effective software tools supporting the whole SLA management lifecycle. The impact of our language and of its software tools is assessed by considering a series of experiments that provide empirical evidences of the advantages of SLAC.

Keywords: Cloud Computing, Service Level Agreements, Dynamic SLAs

1. Introduction

In cloud computing, consumers outsource their business functions to entrusted cloud service providers [1]. This requires the definition of formal guarantees that the delivered services are compliant with the agreed terms, which are specified via the so-called Service Level Agreements (SLAs). SLAs not only provide guarantees to consumers, but are also the principal means for cloud providers to establish their credibility, and to attract or retain customers [2].

Nevertheless, the main market players, like Amazon, Google or Microsoft, do not offer machine-readable SLAs, but only natural language descriptions of service conditions. This conduct hinders the automation of the SLA management and may lead to controversial interpretations of given SLAs. To overcome this issue, academia and standardisation bodies have been proposing in the last years several domain-specific languages for defining SLAs [3, 4, 5, 6, 7, 8, 9, 10, 11]. These languages, however, do not support crucial features of the cloud domain, such as the dynamicity of the (changing) requirements of the parties; the role of the broker; the types of clouds (e.g., community or hybrid); and the wide range of service types. Adapting and extending such languages to consider these features would result in deep structural and conceptual changes, up to the point of denaturing them and adding new layers of complexity.

Therefore, we propose a novel SLA definition language, called SLAC, devised for comprehensively dealing with the specific and distinctive aspects of cloud services. SLAC builds upon earlier proposals [12, 13], in which the mentioned features were addressed separately and differently. In this paper we have revised the elements and mechanisms of the previous languages, and incorporated them into a new single language; we refer to Section 7 for a compar-

*Corresponding author
Email addresses: rafael.uriarte@imlucca.it (Rafael Brundo Uriarte), rocco.denicola@imlucca.it (Rocco De Nicola), vincenzo.scoca@imlucca.it (Vincenzo Scoca), francesco.tiezzi@unicam.it (Francesco Tiezzi)

ison with our previous papers. The key features of SLAC are the following:

- Formally defined syntax and semantics, guaranteeing non-ambiguous SLAs and enabling SLAs evaluation through constraint solving;
- Novel mechanisms for dealing with the dynamicity of SLA terms, to take full advantage of the elasticity of cloud services and guaranteeing flexibility to the involved parties;
- Linguistic support for multi-party SLAs, possibly involving brokering;
- Software tools supporting the whole lifecycle of SLAs, from editing to services deployment, from monitoring to enforcement.

While works addressing other phases of the service lifecycle, such as, scheduling and monitoring, take into account the dynamic nature of cloud services, this dynamism has been overlooked in the definition of SLA languages. More specifically, existing languages attempt to cope with it only by relying on renegotiating of the SLA terms, which is a heavy and expensive process. These languages are thus limited to the definition of the so-called *static SLAs*, where parties agree on contractual terms that remain the same for the whole contract lifetime. SLAC, instead, provides novel linguistic facilities that allow SLA's designers to define, in addition, *dynamic SLAs*. These offer the possibility of automatically modifying at runtime the service level according to predefined conditions stipulated in the contract itself. Such runtime changes may act on the values of the metrics involved in the current contractual terms of the SLA, or on the set itself of valid SLA terms (i.e., the currently enforced terms can be dynamically replaced or deleted, and new terms to be enforced can be added). This permits to guarantee additional flexibility to the parties by fully exploiting the elasticity of cloud services while avoiding, or at least reducing, human intervention. On the other hand, to have full control of these dynamic changes, the SLA parties have to agree, at negotiation time, on foreseeable scenarios. In this way, the SLA does not loose its contractual notion, since all the possible changes are specified within the contract.

To sum up, the main contribution of this paper is a comprehensive methodology supporting the whole lifecycle of SLAs for the cloud domain. Its specific contributions are:

- The SLAC language, devised for defining both static and dynamic SLAs (Section 4);
- The SLAC software framework, providing support to editing, monitoring and evaluating SLAs (Section 5);
- The validation of the SLAC approach via experiments on a cloud testbed (Section 6).

Furthermore, in Section 2 we introduce a simple scenario to motivate the use of dynamic SLAs. In Section 3 we illustrate our proposed lifecycle for dynamic SLAs. In Section 7 we provide a comparison with related works, showing the advancement with respect to the state of the art. In Section 8 we further discuss challenges and benefits of our approach, and suggest future research directions.

2. Uses Cases and Motivating Scenario

In this section we illustrate the benefits of dynamic SLAs by describing four use cases and a motivating scenario that will be used throughout this paper to present the SLAC language and the related software framework. These are just simple illustrative examples; obviously, dynamic SLAs may be fruitfully applied in many situations, ranging from simple scalability problems to complex SLA composition.

Use cases. Let us imagine that a municipality decided to outsource the processing of its SmartCity project, which relies on sensors spread around the city (used, e.g., for traffic congestion monitoring, indoor positioning, smoke detection and smart lighting). The need of processing varies considerably according to, e.g., time, weather conditions and season. With static SLAs, in case of high demand, the cloud service provider may impose a new, higher price to scale the service, or he may even refuse to offer the service, which would imply additional costs to find

a new provider and transfer there the service. With dynamic SLAs, instead, the municipality could have a bidding document defining all the conditions to automatically change the valid terms of the SLA.

Now, let us consider a hospital that relies on a cloud service provider to perform diagnostics based on Magnetic Resonance Imaging. In most cases the diagnoses can be made within two days without problems. However, in cases of emergencies, the hospital may need diagnoses within an hour without any limit on the number of simultaneous requests. In their dynamic SLA, the parties can specify the conditions to pass from a normal to an emergency state, and hence to change the response time and the related service costs.

Dynamic SLAs can be useful also for cloud providers when they have overbooked their resources and the demand raises unexpectedly. In such case, to prevent SLA violations, paying fines and losing clients' trust, the provider might want to activate a clause in the SLA that allows him to reduce the resources provided to some clients (e.g., change of the type or the number of VMs) while offering monetary discounts to compensate the service reduction.

Brokers, who have a key role in multi-cloud environments, may also take advantage of dynamic SLAs. In a multi-cloud scenario, like, e.g., the one described in [14], a broker is responsible for the service but outsources the actual execution of the service to a provider. Dynamic SLAs, in this case, are essential to guarantee that providers will be able to adapt service provision according to what was sold to the final consumer.

Motivating scenario. We consider as running example a scenario regarding the provision of IaaS services for hosting an e-commerce site. Alice contracts this service from the IMT provider. Since she does not have a reliable infrastructure to verify if the service is compliant with the terms of the agreement, she requests an auditor, Bob, to do it and defines the monitoring terms. Although she believes that her business will grow significantly in the future, she requires at the beginning a small virtual machine (S_VM), which provides a limited amount of CPU units and RAM memory. Anyway, she also already defines the contractual terms for upgrading the service, during its execution, up to 2 large virtual machines (L_VM_IMT), with more CPUs and memory, improved availability, and an additional requirements in terms of response time delay, at a higher service price. To avoid surprises and plan the costs of her business, she agrees on a pre-defined price for all these requirements of the service. On the other hand, IMT reserves itself the right to outsource the service to UNICAM when Alice is using at least one large VM, thus acting also as a broker. Moreover, IMT allows Alice to freely scale up and down the service level with one exception: once Alice scaled up to a large VM, she must explicitly ask IMT the authorisation for moving to a small VM again. The reason being that small VMs are expensive to maintain in specific conditions and IMT cannot outsource them to UNICAM which does not provide small VMs.

This SLA is represented, in Figure 1, as an automaton with five states, each representing a possible service level. State transitions are labelled by the events triggering a change of state. This behaviour is an example of how vertical (e.g., replacing a small VM with a large one) and horizontal (e.g., adding more VMs) scalability can be handled and regulated by SLAC. The example also shows the flexibility of dynamic SLAs and the expressive power of SLAC, which may represent important business advantages for all involved parties. We will present the SLAC specification of this SLA in Section 4, where we will use it to illustrate the key features of the language.

3. Lifecycle of Dynamic SLAs

In this section, we first provide some background notions on cloud SLA management, then we present a lifecycle specifically defined for dynamic SLAs.

The management of clouds attempts to optimise preferences and policies of stakeholders according to the state of the cloud and the agreed SLAs.

Five types of actors are involved when defining cloud SLAs [15, 16]:

- *Consumer* is the user or beneficiary of the service (in our running scenario, Alice plays this role);

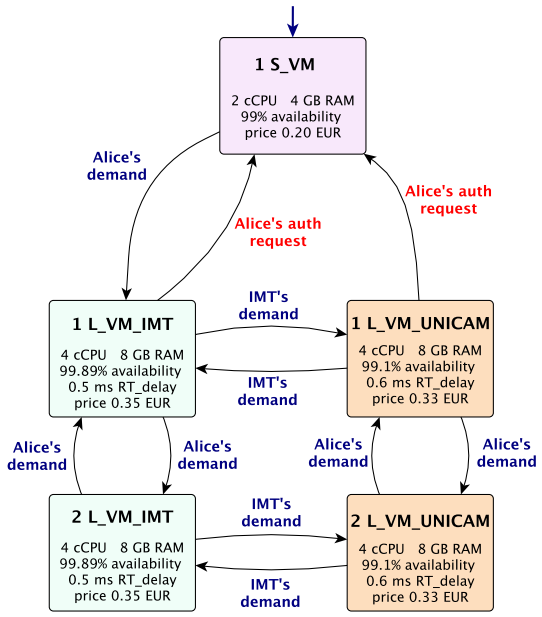

Figure 1: Automaton for the dynamic SLA of the motivating scenario.

- *Provider* makes services available to interested parties (both IMT and UNICAM play this role);
- *Carrier* provides connectivity and transports the services from providers to consumers (our SLA example does not involve this role);
- *Auditor* assesses the service provided in term of, e.g., performance (Bob plays this role);
- *Broker* negotiates relationships between providers and consumers, and may manage the service use and delivery (IMT plays this role).

In the cloud domain, services are offered with two main SLA modalities:

- *Service-Based SLAs* refer to non-negotiable and ready-to-sign SLAs available for all consumers. This is the most common type of SLA in clouds.
- *Customer-Based SLAs* have negotiable terms and are agreed with individuals or groups (also large organisations or brokers) to adapt the provided services to their needs. Although more flexible, this modality is significantly more complex and definitely less used.

To fulfil their dynamic needs, in both cases the involved parties might want to change the terms of the SLA for some predicted scenarios. However, current SLA models are static, that is, once agreed, the SLA remains the same till its expiration date or till all the involved parties agree to terminate or renegotiate it. Often, elasticity is offered through a generic SLA valid for all instances of service (defined in natural language), or through the possibility of terminating the previous and starting a new SLA.

In fact, the terms *scalability*, *elasticity* and *dynamism* are commonly confused in the domain. *Scalability* refers to the capacity to cope with the increase of workload (new services or increasing demand of the existing ones). This aspect is strongly related to the provider of the service. *Elasticity* refers instead to the flexibility to adapt the service to the current demand as closely as possible, being autonomically managed by the provider, or decided and requested (possibly also autonomically) by the consumer [17]. It concerns only the technical aspects of adapting the service but does not refer to the definition of the terms and authorisations related to them. *Dynamism* is related to the changes in the needs of users and providers. While elasticity enables providers and consumers to maintain the quality of services within the specified range (e.g., adding memory to keep the response time lower than 3 ms), dynamic SLAs formalise the capacity to change the valid terms of the service defined in the SLA, e.g. the response time itself.

Currently, the parties specify dynamism in SLAs using the natural language, but the inherent ambiguity of the latter could lead to misunderstandings. One party could manually translate the SLA in a digital form and (unilaterally) automatise the service provisioning. However, there might be different understanding of the SLA and significant efforts from both sides would be needed to reach an agreement. To limit disagreements, we propose SLAC that, in addition to modelling traditional SLAs, also introduces the possibility of dynamically changing the set

of valid terms of agreements according to predefined conditions stipulated in the SLA. It permits the formalisation of the elasticity of cloud services in the SLA, guarantees flexibility to the parties who can agree, at negotiation time, on foreseeable scenarios, and yet maintains its contractual notion. Notably, dynamic SLAs do not refer to the state of the cloud or of the service, but only to the state of the SLA itself.

We propose a SLA lifecycle, based on the one presented in [18], to support this dynamism. Its phases are described below and depicted in Figure 2.

- *Discovery and Negotiation.* Consumers and providers define requirements and offers, and search for the most appropriate ones. For Customer-Based SLAs, even if the terms are not 100% compatible, the parties can negotiate to reach an agreement.
- *Deployment.* Parties commit the SLA and the service is deployed.
- *Monitoring.* The compliance of the service performance with the specification of the SLA is constantly monitored to detect violations. If some specified conditions are met, the service level may be modified at runtime, by deploying or withdrawing resources and reconfiguring the monitor.
- *Billing and Penalty Enforcement.* This phase is part of a small inner cycle with the monitoring phase and is triggered by the parties, at the end of the SLA, regular periods of time or in a pre-defined date. This differs from the existing lifecycles (including the one in [18]) that invoke penalties and billing only after termination.
- *Termination.* Services and the associated configurations are removed by the provider due to an agreement between the parties, after a violation or due to the expiration of the SLA.

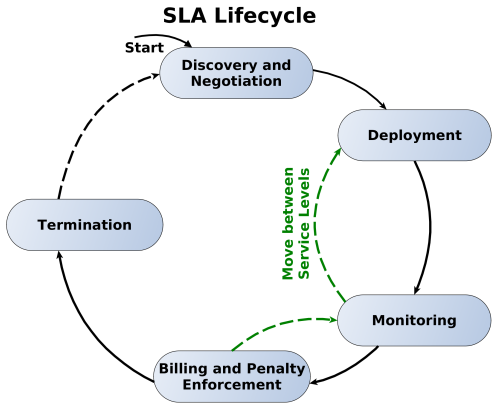

Figure 2: Service lifecycle of dynamic SLAs in the cloud domain (dotted arrows correspond to optional transitions).

4. The SLAC language

In this section we introduce SLAC. For the sake of readability, we focus on an informal description of the language by resorting to an illustrative example based on the motivating scenario defined in Section 2. The formal definitions of the SLAC syntax and semantics, specified in EBNF and denotational style, respectively, are available in an online technical report [19].

SLAC is a domain specific formal language specifically designed for cloud services that supports the main cloud deployment models, supports multiple parties and all roles in the cloud service provision, permits dynamic service level modifications according to pre-defined conditions, and is equipped with a set of software tools supporting SLAs management. These distinctive features, together with its ease of use and its expression power, enable SLAC to cover most scenarios in the cloud domain.

Figure 3 depicts the sections of a SLA defined using SLAC. We informally present them via our running example SLA, reported in Table 1, which refers to a brokered provision of a IaaS service, assessed by an auditor.

SLA Description. This section defines the validity period (start and expiration time) and the parties involved in the provision of the services. A party is defined by its name and its roles. In the running example, the validity of the SLA (line 1) is one year. The involved parties (lines 2-6) are four: two providers (`IMT` and `UNICAM`), a consumer (`Alice`) and an auditor (`Bob`). The `IMT` party has two roles; it is a

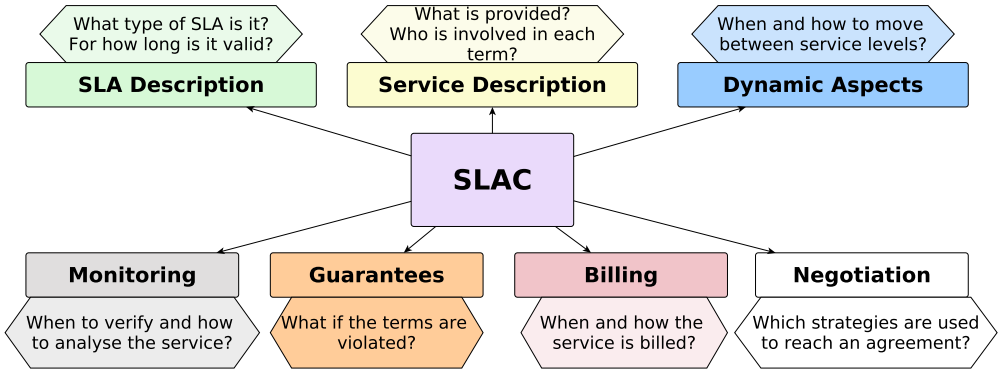

Figure 3: Overview of the sections of a SLA written in SLAC.

provider and it acts also as a broker since, when necessary, it may resort to `UNICAM` for offering the large VM service to `Alice`.

Service Description. This section specifies the details of the service and its quality. It consists of two parts: groups of terms and instantiation of the valid terms (lines 7-27 of the example). The terms of the SLA express the characteristics of the service along with their expected values. Each term requires the specification of the involved parties, i.e., the party responsible for fulfilling the term (a single party) and the consumers of the service (one or more). In our example, the term `IMT` → `Alice:Availability 99.89%` indicates that the party `IMT` provides the service to `Alice` (i.e., one or more virtual machines) with an availability of (at least) `99.89%`. Explicitly defining the involved parties contributes to support multi-party SLAs, reducing ambiguity and leveraging the role of the broker.

Terms/groups marked with * are visible only to the parties involved in the term, to improve security and privacy. For example, the large VM service provided by `UNICAM` (lines 13-19) defines the cost that `IMT`, as a broker, has to pay to `UNICAM` but this information is not accessible to the consumer, who is kept unaware of the SLA terms between the broker and the providers.

Notably, the language offers a set of pre-defined metrics devised for IaaS. Yet, new metrics and their measurement definition can be easily defined without affecting the general language. For a detailed account on this extension procedure we refer to [20].

Additionally, terms can be specified in different granularities by using groups of terms. A group of terms is identified by a name (unique in the SLA) and consists of one or more terms that are valid only inside the group. In the example, we have three groups that represent different types of virtual machines: the VMs provided by `IMT` (`S_VM` and `L_VM_IMT`) and the one by `UNICAM` (`L_VM_UNICAM`).

To use a group in a SLA definition, it is not sufficient to define it, but it is also necessary to instantiate it by specifying the number of instances. For example, the code in lines 26-27 requests the deployment of one instance of the `S_VM` group.

Dynamic Aspects. Users can define runtime changes of the enforceable SLA terms, through the definition of Event-Condition-Actions (ECA) rules. When an event occurs (e.g., a party makes an explicit demand), a condition (defined by an expression) is checked and one or more actions are executed (e.g., the change of the value of a given metric). These rules permit, for instance, agreeing on unilateral or authorized changes from one of the parties. In our example, the consumer can autonomously upgrade the type of VM (lines 29-30), request the authorisation by the provider to move back from a large to a small VM (line 31-34), and add or remove as many large VMs as necessary (lines 35-39). Instead, the broker can freely outsource the provision of large VMs (lines 40-

Table 1: Motivating scenario written in SLAC.

```
1  Effective From:  01/01/19, Expiration Date:  01/01/20
2  Parties:
3   IMT     Role:  Broker, Provider
4   UNICAM  Role:  Provider
5   Alice   Role:  Consumer
6   Bob     Role:  Auditor
7  Term Groups:
8   S_VM:
9     IMT → Alice:cCpu 2 #
10    IMT → Alice:RAM 4  GB
11    IMT → Alice:Availability 99%
12    IMT → Alice:Price 0.20 EUR *
13  L_VM_UNICAM:
14    UNICAM → Alice:cCpu 4 #
15    UNICAM → Alice:RAM 8  GB
16    UNICAM → Alice:Availability 99.1%
17    UNICAM → Alice:RT_delay 0.6 ms
18    UNICAM → IMT:Price 0.26 EUR *
19    IMT → Alice:Price 0.33 EUR *
20  L_VM_IMT:
21    IMT → Alice:cCpu 4 #
22    IMT → Alice:RAM 8  GB
23    IMT → Alice:RT_delay 0.5 ms
24    IMT → Alice:Availability 99.89%
25    IMT → Alice:Price 0.35 EUR *
26 Terms:
27   1 of S_VM
28 Dynamism
29  on Alice demand:
30   replace S_VM with L_VM_IMT & migrate
31  on Alice authorization request:
32   if L_VM_IMT = 1:
33    replace L_VM_IMT with S_VM & migrate or
34    replace L_VM_UNICAM with S_VM & migrate
35  on Alice demand:
36   replace value of L_VM_IMT with _old+1 or
37   replace value of L_VM_IMT with _old-1 or
38   replace value of L_VM_UNICAM with _old+1 or
39   replace value of L_VM_UNICAM with _old-1
40  on IMT demand:
41   replace L_VM_IMT with L_VM_UNICAM & migrate or
42   replace L_VM_UNICAM with L_VM_IMT & migrate
43 Invariants:
44  L_VM_IMT in [1,2]
45  L_VM_UNICAM in [1,2]
46 Monitoring:
47  S_VM.Availability:
48   Frequency:  10 s, Window: 1 month, By:  Bob
49  L_VM_UNICAM.Availability:
50   Frequency:  6 s, Window: 15 days, By:  Bob
51  L_VM_IMT.Availability:
52   Frequency:  6 s, Window: 15 days, By:  Bob
53 Guarantees:
54  on violation of any.Availability:
55    if Availability > 98%:
56      IMT → Alice:  Bonus 20 EUR
57    else:
58      IMT → Alice:  Bonus 40 EUR
59  on violation of any.Frequency:
60    Bob → IMT,Alice:  notify
61 Billing:
62  accounting:  hourly
63  billing:  monthly
```

42). Replacement actions can use a reserved variable name `_old` to refer to the current value of the term metric or to the number of instances of a given group. Various events are supported by SLAC in the ECA rules (see [19] for a complete account), including, e.g., a request for authorisation from the counter-parties, demands of modifications, which do not require authorisation of other parties, and violations of specific terms. In our example, Alice may produce both demand and request events; in the latter case, a condition is used to restrict the application of the ECA rule to the case where only one large VM is provided. Moreover, the parties can explicitly request the migration of the service to the instantiated resources by using `& migrate` in the action definition. In our example, migration is necessary when the type of VM is changed (from small to large, and back) or when the service provider is changed (from IMT to UNICAM, and back); migration is not necessary in case of scaling up and down of the same service type with the same provider.

The `Invariants` constrains the effects of the `Dynamism` rules, by fixing bounds for terms of the SLA. When an event triggering changes of the SLA is detected, the corresponding changes are applied only if they are compliant with the invariants terms. In our example, invariants are used to specify that the consumer cannot demand more than 2 large VMs (lines 43-45).

Monitoring specifies the information necessary for evaluating the service and for configuring the monitoring system. In the example, the `Availability` metric is monitored by the `Auditor` with different accuracy depending on the virtual machine type. For example, the availability of large VMs is checked by `Bob` more frequently and within a smaller time window with respect to the availability of the small VM.

Guarantees. It specifies the actions to be taken in case of violations and are defined in the form of ECA rules. In our running example, if the `Availability` of any (large or small) VM is violated, but it is still greater than 98%, the broker needs to give a bonus of 20 EUR to the consumer, while if it is less than or equal to 98% the penalty is doubled (lines 54-58). Moreover, if a requirement about the monitoring fre-

quency is violated (due to, e.g., maintenance activity of the provider), the auditor is required to notify the issue to the provider and the consumer (lines 59-60).

Billing. Since SLAC enables modification of the valid terms at runtime, it is necessary to define periods of accounting and the frequency of billing. In our scenario we defined the accounting period in hours and the billing monthly. Thus, if the consumer used a small VM for 48 hours and then demands a large VM, after one month the service will be accounted for 48 hours at the price 0.20 EUR/h, and at 0.35 EUR/h for the other days of the month.

Negotiation. SLAC also supports the definition of templates for discovering compatibility of offers and requests. Templates have a slight different syntax. To provide flexibility in the search of compatible offers, they supports intervals in the numeric metrics, and weights may be assigned to terms to express the priorities of the parties in the negotiation. Templates can be used also to define protocols and strategies to be used in the negotiation, possibly using OWL-Q [21]. However, we do not cover negotiation aspects in this paper, and refer to [20] for a detailed treatment.

4.1. Semantic Overview

A SLA is formulated as a Constraint Satisfaction Problem (CSP), which is evaluated, at design-time, to identify inconsistencies in the specification and, at run-time, to check compliance with the monitoring data collected from the cloud system. To support dynamic SLAs, we adopt dynamic CSP [22]. Dynamic CSP is necessary since not only the value of the terms change (e.g., response time) but the set of terms itself (with the addition or removal of term). Therefore, a dynamic SLA is represented as an *automaton*, whose states, representing the SLA states, are labelled by a set of constraints, and whose transitions are labelled by events that trigger the state changes.

At run-time, data representing the status of the cloud are collected by the monitoring system and rendered as a CSP as well. Then, the CSPs of (the current state of) the SLA and of the monitoring data are combined for evaluation. After that, the guarantees are evaluated and, possibly, actions are executed. Similarly, the ECA rules of the `Dynamism` section are evaluated; for each applicable rule, the terms of the `Invariants` section are checked in order to apply only those changes that are compliant with the invariants.

More formally, the semantics of a static SLA, or of a single state of a dynamic SLA, is given by a function $[\![\cdot]\!]$ that, given the SLA terms, returns a pair composed of a set of group definitions and a constraint. This pair constitutes the CSP associated to (a state of) the SLA, that will be solved by means of a standard constraint solver. The automaton representing the semantics of a dynamic SLA is generated as follows. The initial state of the automaton is a CSP obtained by applying $[\![\cdot]\!]$ to the terms of the `Term Groups` and `Terms` sections of the SLA specification, considering the invariants. Then, all possible new states are created considering events, conditions and the triggered changes to the SLA constraints specified in the `Dynamism` section and their invariants.

Table 2 presents an excerpt of the SLA defined in our example scenario and the resulting constraints. Only terms and group terms of the SLA are considered for constraints generation, while additional information, such as monitoring frequency and monitoring window, are used only as parameters (not constraints) for the monitoring system.

The automaton resulting from the interpretation of the dynamic aspects of our example is reported in Figure 1.

5. The SLAC Software Framework

To support the use of SLAC, we developed an open source software framework[1] that covers the whole SLA lifecycle illustrated in Figure 2, and all actors involved in the cloud service provision. Figure 4 illustrates the main components of our framework, each of which is described below by relating it to the corresponding phase of the lifecycle and the parties involved. Notably, this figure shows only the parties

[1]The SLAC Management Framework is a free, open-source software that can be downloaded from the SLAC project's website [23].

Table 2: Semantics at work on our running example.

SLA	**Constraints:**
`Term Groups:`	#SLA Terms
S_VM:	$1 \leq$ `S_VM` ≤ 1
IMT → Alice:`cCpu 2 #`	
IMT → Alice:`RAM 4 GB`	#Constraints in the S_VM Group
IMT → Alice:`Availability 99%`	$2 \leq$ `S_VM:IMT,Alice:cCpu:0` $\leq 2 \wedge$
IMT → Alice:`Price 0.20 EUR`	$4 \leq$ `S_VM:IMT,Alice:RAM:0` $\leq 4 \wedge$
`Terms:`	$99 \leq$ `S_VM:IMT,Alice:Availability:0` ≤ 100
`1 of` S_VM	$20 \leq$ `S_VM:IMT,Alice:price:0` ≤ 20

and the components directly related to the SLA management. It is worth noting that even if all parties, apart from the provider, have the same components, they can use them differently. For example, the SLA Inspector component is particularly important for the auditor, since it is his responsibility to verify the compliance of the service provision with the SLA, while the consumer might choose not to exploit that component when an auditor is hired.

Service Discovery. The *SLA Editor* is used by consumers, providers and brokers to define either SLAs, or service offers and requests. The editor performs validation of SLAC models, error highlighting, syntactic checks and autocompletion. It has been developed using Xtext[2], a framework for the implementation of domain specific languages. With the resulting SLAs, *Brokers* discover services by using the solution introduced in [24], which checks the compatibility between offers and requests.

Negotiation. All involved parties have a *Negotiator* component that proposes SLAs and evaluates requests for (re-)negotiation or modifications in the SLA. This component supports dynamic SLAs [24] and may use multiple round negotiations.

Deployment. The *SLA Inspector* parses the SLAs by using ANTLR4[3] and, by relying on the EBNF grammar of SLAC, generates a set of constraints that is sent to the *Service Manager*, which deploys the service. Our implementation of the Service Manager is integrated with the OpenNebula toolkit[4] to enable an automatic SLA/service deployment.

Monitoring. The SLA Inspector provides the *Monitor* (in our case, based on Panoptes [25]) with the information necessary to retrieve data concerning the metrics related to the SLA. The SLA Inspector parses the data received from the Monitor and generates set of constraints whose satisfiability is checked against the SLA constraints using the Z3 solver [26]. In case of non-satisfiability, the agreed guarantees are evaluated and due actions are activated. All parties have a monitoring component that may actively collects data and assess it using its own Inspector, which constantly listens to events or requests, processes modifications, logs them, and requests changes of terms to the Service Manager.

Billing and Penalty Enforcement. The *Biller* is responsible for this phase of the lifecycle. It calculates costs and penalties, and bills the parties regularly or when events (e.g., violations or service changes) are received.

Termination. The SLA termination is handled by the Biller component and by the Service Manager, which un-deploys the service.

All parties have a *Knowledge Manager* component, which contains information about the environment and parties' preferences and priorities, and supports all components in all phases.

[2] `https://eclipse.org/Xtext/`
[3] `http://www.antlr.org/`
[4] `https://opennebula.org/`

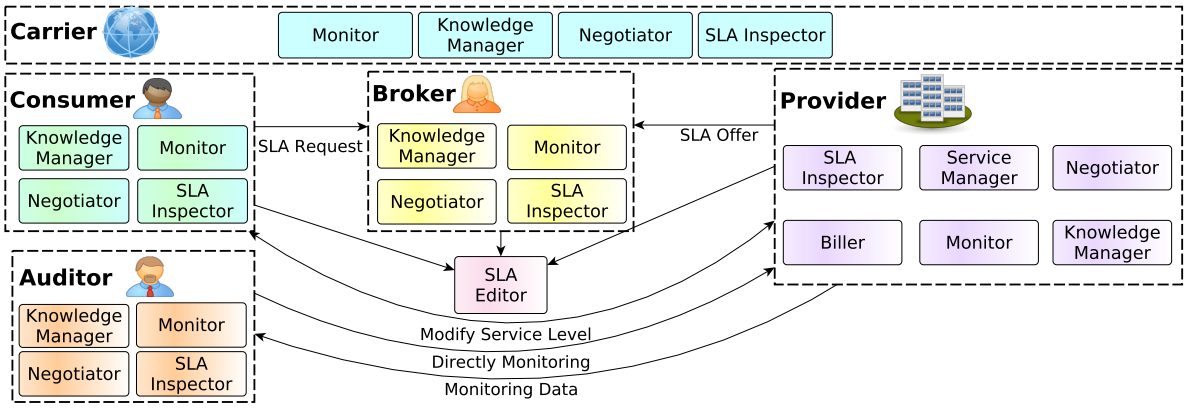

Figure 4: SLAC Management Framework.

6. Experiments

We assess the advantages of SLAC and its framework from the:

- *Consumers' Perspective*: We show the advantages of dynamic services and the flexibility provided by dynamic SLAs from the consumer perspective, by comparing static services, commonly used before clouds, with a scenario with scale up/down possibilities and with a more flexible scenario based on previous reservation of resources.
- *Providers' Perspective*: We focus on the benefits for providers, due to the increased flexibility in SLAs management; we will show that the capacity to move between service levels reduces the penalties for SLA violations by 66%.

6.1. Benefits for consumers

Capturing the dynamism of clouds in the SLA provides flexibility and significant economic and business advantages to consumers. In this section we focus on the economic advantages of dynamic SLAs from the consumer's perspective. We define an IaaS provider inspired by services offered by Amazon[5]: *On-demand* and *Reserved Instances*. In the former, consumers instantiate VMs according to their needs for a fixed price. In the latter, consumers reserve (and pay) in advance for a fixed number of VMs for a period of time, but receive a significant discount. Service offers by Amazon are however specified in natural language. Existing (semi-)formal SLA languages only capture the initial state of these offers. SLAC, instead, is able to formalise also the dynamic aspects and supports the automation of the whole service lifecycle.

In our scenario, we cover different service offers:

- *Static SLA* that defines a static service offer where consumers specify a fixed number of VMs for the whole duration of the service, i.e., the service level cannot change;
- *On-demand* VMs defined using dynamic SLAs, where at every hour consumers can scale up and down their requests according to their needs;
- *Reserved Instances*, which are also defined using dynamic SLAs and allow users not only to scale requests up and down, but also to reserve a fixed minimal number of VMs at a discounted price.

The cost of a non-reserved VM in all offers is \$0.058 and of a reserved instance \$0.036 (the prices are based on Amazon AWS). In our scenario, consumers use VMs to run websites and each VM, equivalent to the micro instance in Amazon AWS, is able to process a

[5] `https://aws.amazon.com/`

maximum of 30000 requests/hour. We assume that there is a charge for not fulfilling a request fixed at \$0.043 for every missing VM. We show in Table 3 an excerpt of the SLAC SLA for the *Reserved Instances* case.

For modelling the scenario we use 20 days of HTTP requests traces of Wikipedia [27]. Each time the experiment is run, we randomly choose 50 pages of the Wikipedia traces, and select among all HTTP requests only those referring to the selected pages to create a time-series of the total requests/hour of each page. The same set of pages is used by all service offers.

Figure 5 shows the flow diagram of our scenario. If the tested service offer is not *On-demand*, we use the information of the first 10 days for predicting the load of the following days of the dataset, which range from 1 to 10, by using the Load Prediction module of our framework. We adopted the time-series forecasting tool Prophet[6], which is based on an additive regression model that considers important variables related to work load prediction, such as, growth, seasonality and holidays. Then, in the case of *Static SLAs*, we simulate the cost of the predicted scenario for different numbers of VMs and select the best scenario by considering predicted load, VM costs and penalties. After deciding the fixed number of VMs for the remaining days, we run the scenario with the requests extracted from the Wikipedia traces and verify every hour whether the select number of VMs is sufficient to satisfy all requests. If not, beside adding the cost of the predefined number of VMs, we add the penalty multiplied by the number of missing VMs.

In the case of *Reserved Instances*, we also use the Prediction module to predict the load of the following days and, via simulations, we determine the number of instances to reserve by considering the VM discount and the extra costs in case some of the reserved VMs are not necessary to fulfil the requests. After defining this number, we run the scenario and at every hour we use the past information to predict the next hour load and scale the service accordingly. Then, we add the cost of the used pre-reserved instances, the price of the additional VMs used for that hour (the full VM price) and the penalties users could incur if they are not able to satisfy their customers.

Table 3: Excerpt of a SLA of the *Reserved Instances* scenario.

```
...
Term Groups:
  Reserved_VM
    IMT → Alice:cCpu 0.5 #
    IMT → Alice:RAM 1  GB
    IMT → Alice:Price 0.036 EUR *
  Non_Reserved_VM:
    IMT → Alice:cCpu 0.5 #
    IMT → Alice:RAM 1  GB
    IMT → Alice:Price 0.058 EUR *
Terms:
   1 of Reserved_VM
Dynamism
  on Alice demand:
    replace value of Reserved_VM with _old+1 or
    replace value of Reserved_VM with _old-1 or
    add term Non_Reserved_VM or
    remove term Non_Reserved_VM or
    replace value of Non_Reserved_VM with _old+1 or
    replace value of Non_Reserved_VM with _old-1
Invariants:
  Reserved_VM in [1,6]
...
```

The *On-demand* service model is similar to the *Reserved Instances* one, but it does not offer the possibility of pre-reservation; it requires paying the full price for each VM without any discount.

Figure 6 shows cost reduction of the on-demand and reserved instances approaches in relations to the static one with different simulation periods, from 1 to 10 days. In average, the cost reduction for consumers using dynamic SLAs with the *On-demand* service is 29%, while with *Reserved Instances* it is 42%. The results show: (i) an increasing cost reduction tendency for both approaches, when the analysed number of days grows, probably due to the lack of flexibility of the static approach; and a reduction on the difference in performance of the on-demand and reserved instances, since the forecasting accuracy is lower for further periods. Overall, these results demonstrate the potential of flexibility for reducing costs. Notably, although these scenarios are already present in commercial offers, they are still not covered by the other SLA specification languages.

[6]https://facebookincubator.github.io/prophet/

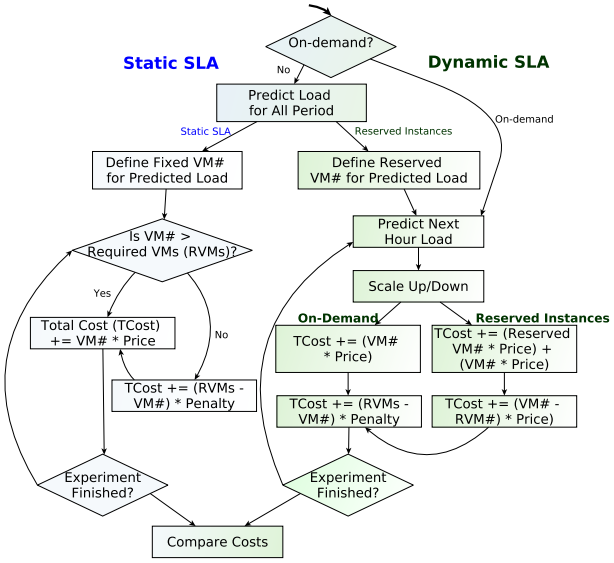

Figure 5: Experiment scenario used to measure the economic impact for consumers.

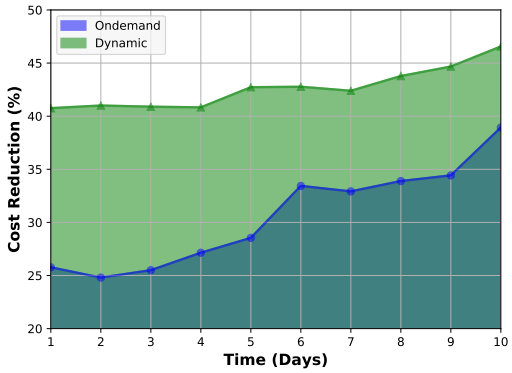

Figure 6: Cost reduction of the *On-demand* and *Reserved Instances* approaches in relation to the *Static* approach.

6.2. Benefits for providers

Here, we analyse the number of violations, the penalties and the total provider revenue to compare the advantages of three different approaches: *Static SLAs* (SLAs do not change during their lifetime), *Renegotiation* (parties can renegotiate the existing SLAs) and *Dynamic* SLAs (some terms can be dynamically changed). We propose a scenario where providers analyse the running services, and in case of high SLA violation risk, which implies high penalties and lost of trust, they may modify the service quality and cost in the Dynamic or Renegotiation approaches. The results, discussed below, demonstrate the flexibility of SLAC, its capacity to reduce the number of SLA violations and to improve the revenue of the involved parties.

6.2.1. Use Case Implementation

The main components, together with their interaction and implementation, are shown in Figure 7. The SLAC Inspector parses and evaluates SLAs for the service specification and requirements, and sends the outcome to the Service Manager, which is specifically designed to guarantee the correct deployment and execution of services, and to manage the cloud infrastructure. The Panoptes Monitoring System (Monitor component) is automatically configured by the Service Manager and provides monitoring data to the Knowledge Manager and to the SLA Inspector. The Knowledge Manager measures the risk of the running services of not meeting the deadline specified in the SLA, and informs the Negotiator about these risks. We implemented this violation risk analysis by relying on the Supervised Random Forest technique [28], which assess these risks based on monitoring information and on the characteristics of the SLA. The Negotiator proposes modifications in the SLA; and, when it receives a service modification proposal, it decides whether to accept it.

Each service is processed according to the workflow depicted in Figure 8 (based on [13]). A service is evaluated regularly, and the penalties are added up when the service is completed.

Static SLAs services are executed by employing the SLA defined at design time. The SLA is verified every

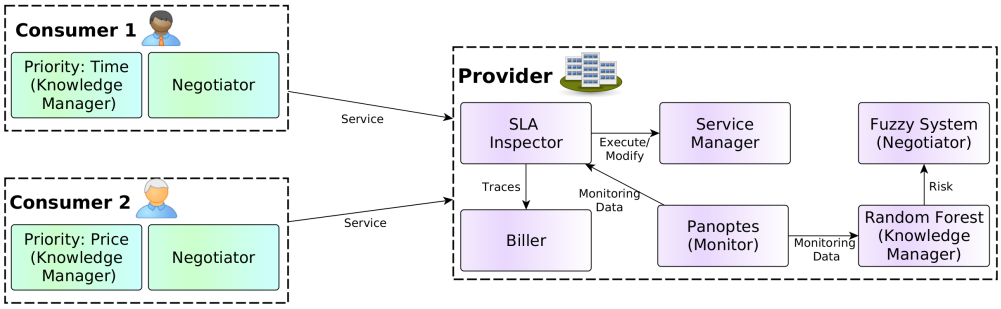

Figure 7: Components of the use case implementation.

minute and, when the service ends, the revenue (price paid) and penalties are computed. In the *Renegotiation* approach, for the sake of simplicity, the risk of SLA violation is measured only once during its execution lifetime, at a random time between the starting and the deadline of the service. If it is not higher than given thresholds, the service is provided normally according to the SLA defined at design time. Otherwise, the provider sends a SLA proposal to the consumer, who analyses it according to its priorities and takes a decision by relying on a Fuzzy Decision System (see below). If the proposal is accepted, the service continues and, after the change, it is evaluated considering the new SLA, otherwise the initially defined SLA remains valid till the end of the service.

The *Dynamic* approach is similar to the Renegotiation one, the only difference is that the involved parties do not have any active role. Indeed, in case of high violation risk, the SLA is modified automatically since the changes are pre-defined in the SLA. In both cases, to motivate or compensate a party for the changes, a bonus (defined in the SLA) is given to the other party when a change is performed during the service execution. Although the bonus a priori is usually much smaller than the bonus required for renegotiating the SLA during the execution, which would improve the results of the Dynamic approach and, consequently, of SLAC, we opt to use the same range of values of the renegotiation approach for the sake of simplicity.

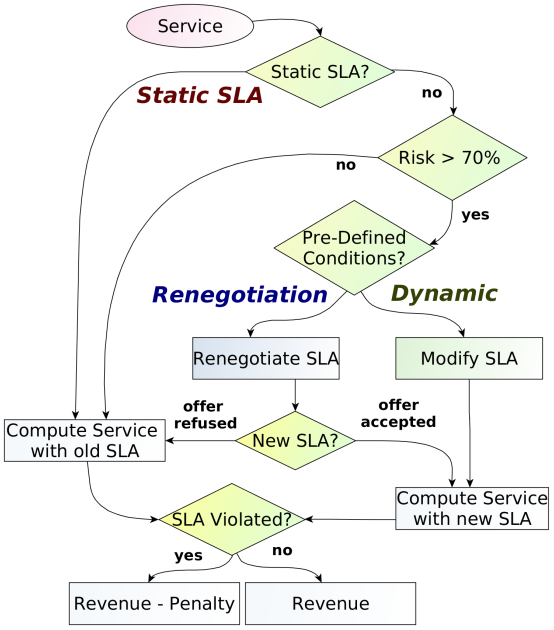

Figure 8: Flow diagram of the service processing for the Static, Renegotiation and Dynamic approaches.

6.2.2. Fuzzy Decision System

To decide whether to accept or refuse a new SLA in the Renegotiation approach, the consumer needs to know the difference between the original SLA and the SLA proposed at renegotiation time. In our use case, to simulate this process, which is typically carried out by a human or by an autonomous decision system, we designed a fuzzy logic decision support system inspired by the approach presented in [29].

Table 4: Fuzzy rules of the consumer decision system.

Rule	Evaluation
P_e increases	favourable
P_r or D increases	unfavourable
P_r and D increase	very unfavourable
P_e increase $< 5\%$	neutral

The decision system takes as input the consumer's priorities and the changes proposed by the provider, e.g., the increment in price, to decides whether to accept the SLA proposed. In our use case, the considered SLA parameters are: the deadline for the service (D), the price to be paid for the service (P_r) and the penalty in case of violation (P_e). Table 4 exemplifies some rules used by the decision system. Moreover, consumers specify their priorities among these parameters and the system gives more weight to the changes related to the higher priorities parameters. For example, if the proposed SLA is neutral with respect to the penalty, very favourable on the service price (the provider reduced it considerably), and very unfavourable in the deadline (time to finish the service increased significantly), this proposal is only accepted if the service price has an higher priority for the consumer than the deadline. For a complete account of the fuzzy rules and the framework used in the experiments, we refer to the SLAC project's website [23].

6.2.3. Evaluation

The experiments were conducted in a cloud with 2 physical machines, providing 12 heterogeneous VMs. Services are created by taking into account the distribution of a trace of a real-world cloud environment, the Google's cloud dataset [30], and the same services are executed using all described approaches. Each service has an associated SLA, which is created along with the service, according to an estimation of the resources necessary to finish the service within the completion time. The considered characteristics of the services used to train the Supervised Random Forest are: CPU, RAM, platform requirements (e.g., operating system and architecture), disk space, completion time and network bandwidth. Different types of services are used in the experiments, such as web crawling, word count, number generation and format conversion, which are similar to real-world applications [31]. Service's penalties and prices are based on the service execution time and on a randomly defined number. Penalties are always higher than the price, since the price is paid even if a service is violated.

First we create a training set before every experiment round by executing 1000 services. Then, the traces of these services are used to train the Random Forest algorithm (Knowledge Manager). The algorithm, then, defines the probability of each service to be in the *violated* and *not violated* classes.

In the actual experiments, new random services are generated and the same services are re-executed for each approach. We run 9 rounds of experiments with the number of services from 100 to 500 (with an increase of 50 services every round). We assume that the services' arrival is a Poisson process, i.e., the time between consecutive arrivals has an exponential distribution and that a service arrives, on average, every 0.7 seconds.

In the case of the Renegotiation approach, the decision system (Negotiator) only accepts proposals which are beneficial for the party that received the offer. Therefore, the provider offers compensations to the consumer; for example, if the violation risk is high, the provider requests more time to finish the service but offers a discount on the price and a higher penalty. The definition of the exact parameters of the considered metrics of the proposal, which are used by the Renegotiation and the Dynamic approaches are randomly generated within a predefined range.

The results of these experiments are illustrated in Figure 9, while Table 5 presents the overall results relative to the Renegotiation and the Dynamic approaches, expressed as percentages: in the case of penalties and revenue, the results correspond to a comparison with the Static approach, whilst for the other features they result from a comparison with the total number of services. Considering the parameters defined for the renegotiation approach and the benefit threshold used in the experiments, around 60% of the modification requests were accepted and carried out. Using the Dynamic approach, 21% of the services were modified, mainly due to high risk of

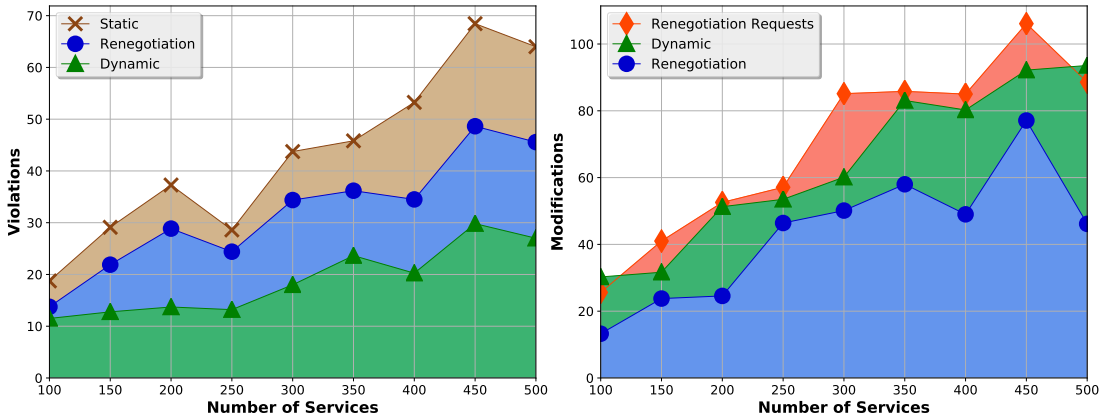

Figure 9: Performance analysis.

Table 5: Results in comparison to Static SLAs

	Renegotiation	**Dynamic**
Modification Requests	21%	0%
Modifications	12%	19%
Penalties	-32%	-66%
Revenue Increase	13%	24%

violation (more than 19%).

Overall, the flexibility provided by the Dynamic approach increased the revenue by 24% and reduced the penalties by 66%, while these measures were only 13% and 32% for the Renegotiation approach.

The benefits of the Renegotiation and Dynamic approaches heavily depend on the accuracy of the violation risk analyses. The results show that, although the penalties were reduced by 66%, the impact on the total revenue was an increase of around 24%. The main reasons for this difference are: (i) the limited impact of the penalties on the total revenue due to the low average number of violations; (ii) the compensation provided to the consumers when a modification is requested, which lowers the price paid for that service and sets higher penalties in case of violation; (iii) the number of SLAs that were violated even after modifying the service, since most of the modification requests increase the penalty in compensation for the higher service completion time. This suggests that performing an analysis to define the additional time required to avoid violations instead of generating a random number could improve significantly the total revenue.

Also, the parameters defined in the SLA modification proposal may have a considerable impact on the results. We adjusted these parameters to simulate a real-world situation, where every party defends his interest. Moreover, it can always be used together with the Renegotiation approach in case not all relevant modifications are included in the SLA.

7. Related works

Although we can find in the literature general discussions about the dynamism and flexibility of electronic contracts (see, e.g., [32, 33]), the dynamic aspects for SLAs have not been investigate before. In this section, we show how SLAC advances the state of the art with respect to dynamism and to the main features a language of the domain needs, and compare it to other languages defined not only for the cloud domain but also for related ones, such as service-oriented and grid computing. This analysis provides a broader view of the advantages and features of each of them and shows the advancements of SLAC also in relation to the state of the art in different areas.

Below we briefly describe the evaluation criteria. The results of our evaluations are summarised in Table 6 and show the advantage of SLAC over other

formalisms. Important characteristics of the cloud domain, like Dynamism, All Parties, and Broker, are supported only by SLAC. Its supporting software framework allows an easy deployment of SLAC SLAs in a wide-range of real-world scenarios.

For our assessment, we consider the following formalisms: WSLA [3], WS-Agreement [4], WSOL [5], RBSLA [6], Linked USDL (LUA for short) [7], SLALOM [8], SLAng [9], SLA* [10], CSLA[11], rSLA [34], ySLA [35][7] and, of course, SLAC. We evaluate them by first considering *General* features of the languages and then assessing their impact on the different phases of SLAs lifecycle. In the table, we use yes ✓ for indicating that the feature is fully supported, ✷ for indicating that the feature is partially supported, and ✗ for indicating that the feature is not supported. The categories and considered aspects are defined below.

General

Cloud Domain considers whether a SLA language has been specifically designed for cloud computing.

Service Models evaluates whether all service models (including specific vocabularies) are directly supported or, in some cases, extensions are needed.

Formalisation refers to the level of formality in the definition of syntax and semantics of the language.

Dynamism considers the capacity to express possible changes of the SLA terms at runtime.

Confidence or Fuzziness is the capacity to deal with QoS uncertainty, with confidence defining the percentage of compliance of clauses, and fuzziness referring to an acceptable interval around the threshold of a metric.

Reusability refers to the possibility of reusing constructs defined in a template or in a SLA across different SLAs. *Reusability Scopes* refers to the possibility of specifying scopes for the definition of terms and monitoring constructs.

Composability is the ability to express composite SLAs.

[7]In this work, the authors focus only on reusability, which hinders our analysis. Since they write that the language is based on rSLA, we use the results of the rSLA analysis for those categories for which we do not have information.

Extensibility evaluates whether the language terms and metrics can be extended.

All parties considers whether it is possible to describe all parties involved in the service provision and in all actions (monitoring, verification, provision, etc.).

Price Model is the level of coverage of price schemes and of computation models.

Consistency check considers whether consistency of SLAs is verified; we write ✷ if only syntactic checks are offered, ✓ if also semantic aspects are considered, and ✗ if no check is performed.

Definition, Discovery and Negotiation.

Editor for writing SLAs, they can be generic (✷), domain-specific(✓) or absent (✗).

Broker may have different levels of support when taking decisions.

Metric Definition refers to the possibility offered for defining quality metrics.

Alternatives evaluates the ability to specify alternative levels of service.

Soft Constraints is concerned with the use of soft-constraints to address over-constrained requirements.

Matchmaking Metric enables the specification of how to match equivalent metrics or metric units.

Negotiability is the ability to indicate how, and to which extent, quality terms are negotiable.

Deployment and Monitoring.

Metric Schedule indicates how often the SLA terms are measured.

Metric Provider indicates the possibility of specifying the party responsible for monitoring each SLA term.

Automatic Deployer refers to the provision of tools for automatic deployment of the service.

Integrated Monitoring is concerned with the capacity to automatically configure the monitoring system relying on the SLA specification.

Billing and Penalty Enforcement.

Penalties and *Rewards* to be enforced under specified conditions. In this case, ✓ stands for fine-grained support (at the level of service level objectives), ✷ for coarse-grained, and ✗ for no support.

Actions triggered by SLA violations.

Conditions Evaluator refers to the possibility of specifying the party in charge of auditing each term.

Assessment Scheduler specifies when each term is assessed.

Termination

Automatic Undeployment of the used resources.

We close the section by comparing the version of SLAC proposed in this paper with respect to the earlier proposals it builds upon by significantly revising and extending them. In this work we: (i) appropriately integrate language extensions into the core language; (ii) simplify the resulting linguistic constructs, and add new ones for specifying, e.g., migration, monitoring information and visibility control for terms; (iii) separate the SLA definition and the negotiation languages; (iv) describe new experiments; (v) discuss the impact of dynamic SLAs in different domains; (vi) define a new service lifecycle; and (vii) describe the design and implementation of a software framework to support the automation of the whole new lifecycle.

8. Concluding Remarks

We have introduced the SLAC language for defining dynamic service levels; proposed a new lifecycle for the service provision in clouds; discussed the main challenges that our new language poses; provided several example scenarios using the language; proposed a software framework covering the whole SLA lifecycle; presented experimental results showing that both providers and consumers can benefit from SLAC; discussed non-measurable benefits for providers, consumers and brokers.

The SLAC language contributes to the automation and specification of services and covers a wide range of use cases that are not covered by static SLAs and corresponding languages. The dynamic part of SLAC is particularly useful for brokers and medium/big companies that require guarantees and plan for long term, but can also be used by small consumers, e.g. to ensure vertical and horizontal scalability. The language is intuitive and extensible and covers the most important aspects of the cloud domain.

8.1. Discussion

Dynamic SLAs impact significantly on cloud management and may add complexity to this task. Below, we present the main challenges and opportunities of using this paradigm according to each phase of the SLA lifecycle and hint at possible solutions for these challenges.

8.1.1. Complexity and challenges of dynamic SLAs

Planning. From the provider's perspective, the planning and discovering phases with dynamic SLAs are similar to the corresponding phases of static SLAs. Providers need to define the conditional changes to be executed at runtime for all services, but this extra effort is necessary only when setting up new services or updating distribution policies.

Consumers can take a more general approach and consider only service scalability in dynamic SLA. However, to fully benefit of this approach, the services need to be meticulously planned and future needs carefully considered. Big companies, multi-clouds and brokers are naturally the main beneficiaries of this approach, since the cost of this planning process is normally just a small fraction of the benefits that flexibility and business certainty can bring.

From the broker perspective, there are several possible scenarios depending on the type of brokerage [16]. The broker might use dynamic SLAs on both sides, or only with providers or only with consumers. Since the profit margins of brokers is usually small, a careful planning of the dynamic part of SLAs, mainly with providers, is needed. Dynamism would permit brokers to precisely determine the costs of different scenarios and offer competitive services.

SLA dynamism, however, requires the anticipation of possible scenarios from all participants and some of its advantages depend on the quality of this anticipation, since unforeseen scenarios might require renegotiation. The typical, and straightforward, strategy to foresee these scenarios is by means of human evaluation of the considered use cases. However, many different solutions may be applied to automatise the whole process. Possible solutions range from the use of simple flexibility rules for the main terms (e.g., having a margin of 20% more VMs than

Table 6: Evaluation results of SLA languages

	Criteria	WSLA	WS-A	WSOL	RBSLA	LUA	SLALOM	SLAng	SLA*	CSLA	rSLA	ySLA	**SLAC**
General	Cloud Domain	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
	Service Models	✷	✷	✷	✷	✷	✷	✷	✷	✓	✷	✷	✷
	Formalisation	✗	✗	✗	✷	✷	✷	✓	✓	✷	✷	✷	✓
	Dynamism	✗	✗	✗	✷	✗	✗	✗	✗	✗	✗	✗	✓
	Confidence or Fuzziness	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
	Reusability	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓
	Reusability Scopes	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Composability	✗	✷	✗	✗	✗	✗	✓	✓	✷	✗	✗	✓
	Extensibility	✓	✓	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓
	All Parties	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	Price Model	✗	✗	✗	✗	✓	✗	✷	✓	✷	✗	✗	✓
	Consistency Check	✷	✷	✷	✓	✓	✷	✓	✓	✗	✗	✗	✓
Definition Disc./Nego.	Editor	✷	✷	✷	✷	✷	✗	✗	✓	✓	✓	✓	✓
	Broker	✗	✷	✗	✗	✗	✗	✗	✗	✗			✓
	Metric Definition	✓	✗	✗	✓	✗	✗	✗	✓	✗	✷	✷	✓
	Alternatives	✷	✷	✷	✷	✗	✗	✗	✗	✗	✗	✗	✓
	Soft Constraints	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	Matchmaking Metric	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
	Negotiability	✗	✷	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Deploy. Mon.	Metric Schedule	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
	Metric Provider	✓	✗	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓
	Automatic Deployer	✓	✗	✗	✓	✓	✗	✗	✷	✗	✗	✗	✓
	Integrate Monitoring	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓
Billing and Penalty Enfor.	Penalties	✗	✓	✷	✷	✓	✓	✓	✓	✓	✓	✓	✓
	Rewards	✗	✓	✗	✷	✓	✗	✗	✗	✗	✗	✗	✓
	Actions	✓	✗	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓
	Condition Evaluator	✓	✗	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓
	Assessment Schedule	✷	✷	✷	✷	✗	✗	✓	✓	✗	✓	✓	✓
Ter.	Automatic Undeployment	✗	✗	✗	✗	✗	✗	✗	✷	✗	✗	✗	✓

the estimated), to more advanced solutions based on machine learning models that, to determine the required flexibility, take into account the modifications required in previous services and their costs.

Discovery. Manual discovery of services becomes more difficult when conditions and actions change the valid terms of the SLA. With dynamic SLAs, a large number of states have to be analysed; thus, verifying whether there exists a state that is not compatible with the parties' specification is costly and implies low matching rates. Therefore, automated techniques that consider the problem of state explosion are necessary. Such techniques could compute the similarity between offers and requests, and select just the most compatible ones.

Negotiation. Negotiation is another significant challenge since there is a large number of possible solutions or requirements from the parties. In [24], we proposed an open source negotiation framework for matching offers and requests and for facilitating the agreement between the parties in case no immediate matching is possible. The framework provides adaptation, consistency check, verification of SLA properties, suggesting the changes necessary for reaching an agreement. In this work, the preferences of the parties are defined by a utility function, which is a first step to automatise the whole negotiation process and reduce the need for human intervention.

Scheduling. Scheduling and service admission need to handle service modification requests that may not require consent from the provider. Although most of the existing methodologies employ statistical methods to predict systems' load and possible variations, the agreement about pre-defined changes is a valuable source of information as it contains the explicit definitions of the changes which are more likely to happen. This information might be used to define the best way to deploy a new service (e.g., considering also the resources that would guarantee possible scalability) and to decide whether to admit a new service (e.g., considering the resources that should be reserved and that are likely to be used in the future). Machine learning algorithms are good candidate for the development of schedulers in this area, because such algorithms can learn patterns based on the consumers profiles, monitoring data and SLA characteristics to predict changes (temporal, arbitrary and conditional) and use this information to schedule new services or optimise their placement.

Service Management. Management solutions can also use the conditions in the SLAs to estimate and adjust systems' load and improve revenues. The immediate advantage is the possibility of activating new terms to avoid violations of the SLA agreed with a given consumer or violation of other SLAs established with other consumers. For example, in case an important consumer requests a large number of new VMs, and the provider does not have enough VMs available, it can fulfil the request by reducing the number of VMs from another consumer leveraging on the dynamism of the SLA with the other consumer, while providing a discount to him. Obviously, taking advantage of this mechanism is a complex process that requires considering multi-objective management problems to find optimal (or better) solutions.

Even if all these challenges are important to the consolidation of dynamic SLAs, we believe that the most urgent ones to leverage its adoption are: the need for automatic solutions for planning and foreseeing the possible SLA scenarios, and new approaches to negotiate them.

8.2. Advantages of dynamic SLAs and Future Works

In our experiments, we showed the possible economic advantages of using dynamic SLAs for providers and consumers. Besides these benefits, there are several other advantages of our approach, for example:

- SLAC extends the coverage of SLA definition languages to new scenarios by supporting dynamism and other concepts already in use in contracts defined in natural language.
- With SLAC, vertical and horizontal scalability can be formally defined.
- Business can be planned meticulously considering future scenarios, conditions and penalties in case of violations.
- Explicitly supporting brokers, which can combine the benefits for consumers and providers.

We plan to extend the language to support Edge Computing and we are currently analysing the use of

SLAC in the context of blockchain and *smart contracts*, whose terms, such as payment, confidentiality and quality, are automatically enforced by relying on a previously agreed protocol. In [36] we survey blockchain-based cloud solutions and discuss the role of SLAs in this field, while in [37] we describe an architecture to utilise SLAC SLAs as smart contracts in these contexts. A particular characteristic of smart contracts is that they can consistently be executed by a network of mutually distrusting nodes, without the arbitration of a trusted authority [38]. Smart contracts are prominently associated to platforms based on distributed ledger technologies, such as Ethereum; are general purpose, covering different areas, from finance to cloud services; and many smart contracts definition languages are Turing-complete.

Similarly to smart contracts, also SLAs formalise the terms of an agreement, specifically for the provision of a service, and can be automatically enforced according to a previously agreed protocol. Although our SLAs are defined in a domain specific language, which does not have the expression power of smart contract languages, SLAC covers the important features of the domain, including the dynamic aspects, monitoring and accounting/billing. Yet, SLAC was designed for environments with trusted authorities. Therefore, to actually deploy SLAC in distributed ledger environments, two main challenges need to be addressed: the trust problem, i.e., the enforcement of SLAs without trusted authorities; and privacy, since all participants can read and execute the smart contracts, which could reveal the identity of the involved parties and details about the service.

Lastly, notwithstanding the open gaps and the known resistance of the large market players and businesses to adopt new concepts and change processes - e.g., most of them still use SLAs in natural language - we believe that dynamic SLAs could be widely adopted in academia and industry because of:

- the large number of new scenarios they support;
- the gain in flexibility and business security;
- the fact that blockchain-based clouds could increase trust and eliminate vendor lock-in, and thus stimulate providers to look for new competitive advantages;
- the fact that dynamic SLAs can be easily converted to smart contracts, and thus make SLAC a good candidate as SLA definition language for the next generation clouds.

9. Acknowledgements

We would like to thank Kyriakos Kritikos, Ivona Brandic and all the anonymous reviewers for their detailed review and the very helpful comments.

References

[1] T. S. Dillon, C. Wu, E. Chang, Cloud computing: Issues and challenges, in: Proc. of AINA, IEEE Computer Society, 2010, pp. 27–33. doi:10.1109/AINA.2010.187.

[2] D. Kyriazis, Cloud computing service level agreements–exploitation of research results, European Commission Directorate General Communications Networks Content and Technology Unit, Tech. Rep 5 (2013) 29.

[3] A. Keller, H. Ludwig, The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services, JNSM 11 (1) (2003) 57–81. doi:10.1023/A:1022445108617.

[4] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu, Web Services Agreement Specification (WS-Agreement), Tech. rep., Open Grid Forum (2007).

[5] V. Tosic, B. Pagurek, K. Patel, WSOL - A Language for the Formal Specification of Classes of Service for Web Services, in: Proc. of ICWS, CSREA Press, 2003, pp. 375–381.

[6] A. Paschke, RBSLA A declarative Rule-based Service Level Agreement Language based on RuleML, Proc. of CIMCA-IAWTI 2 (2005) 308–314.

[7] C. Pedrinaci, J. Cardoso, T. Leidig, Linked USDL: A vocabulary for web-scale service trading, in: Proc. of ESWC, 2014, pp. 68–82.

[8] A. Correia, F. B. e Abreu, V. Amaral, SLALOM: a language for SLA specification and monitoring, CoRR abs/1109.6740.
URL http://arxiv.org/abs/1109.6740

[9] J. Skene, D. D. Lamanna, W. Emmerich, Precise service level agreements, in: Proc. of ICSE, 2004, pp. 179–188.

[10] K. T. Kearney, F. Torelli, C. Kotsokalis, Sla*: An abstract syntax for service level agreements, in: Proc. of the 11th IEEE/ACM GRID, 2010, pp. 217–224.

[11] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, P. Sens, Sla guarantees for cloud services, Future Generation Computer Systems 54 (2016) 233–246.

[12] R. B. Uriarte, F. Tiezzi, R. De Nicola, SLAC: A Formal Service-Level-Agreement Language for Cloud Computing, in: Proc. of UCC, 2014, pp. 419–426. doi:10.1109/UCC.2014.53.

[13] R. B. Uriarte, F. Tiezzi, R. De Nicola, Dynamic slas for clouds, in: Proc. of ESOCC, Springer, 2016, pp. 34–49.

[14] S. Farokhi, F. Jrad, I. Brandic, A. Streit, Hierarchical sla-based service selection for multi-cloud environments, in: Proc. of CLOSER, 2014, pp. 722–734.

[15] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, D. Leaf, Nist cloud computing reference architecture, NIST special publication 500 (2011) (2011) 292.

[16] P. Mell, T. Grance, The NIST Definition of Cloud Computing (2011). doi:10.1080/1047621920040218.

[17] N. R. Herbst, S. Kounev, R. H. Reussner, Elasticity in cloud computing: What it is, and what it is not., in: ICAC, Vol. 13, 2013, pp. 23–27.

[18] L. Wu, R. Buyya, Service Level Agreement (SLA) in Utility Computing Systems, arXiv preprint arXiv:1010.2881.
URL http://arxiv.org/abs/1010.2881

[19] R. B. Uriarte, V. Scoca, F. Tiezzi, R. De Nicola, SLAC: Formal Definitions, Tech. rep., IMT, http://sysma.imtlucca.it/tools/slac/ (2017).

[20] K. Kritikos, R. B. Uriarte, Semantic sla for clouds: Combining slac and owl-q, in: Proc. of CLOSER, ScitePress, 2017, pp. 432–440. doi:10.5220/0006299804320440.

[21] K. Kritikos, D. Plexousakis, Owl-q for semantic qos-based web service description and discovery, in: Proc. of SMR2, CEUR-WS. org, 2007, pp. 114–128.

[22] G. Verfaillie, T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: Proc. of AAAI, Vol. 94, 1994, pp. 307–312.

[23] SLAC project website.
URL http://sysma.imtlucca.it/tools/slac/

[24] V. Scoca, R. B. Uriarte, R. De Nicola, Smart contract negotiation in cloud computing, in: Proc. of IEEE CLOUD, 2017, pp. 592–599.

[25] R. B. Uriarte, C. B. Westphall, Panoptes: A monitoring architecture and framework for supporting autonomic Clouds, in: Proc. of NOMS, IEEE, Poland, 2014, pp. 1–5.

[26] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proc. of TACAS, 2008, pp. 337–340.

[27] G. Urdaneta, G. Pierre, M. van Steen, Wikipedia workload analysis for decentralized hosting, Elsevier Computer Networks 53 (11) (2009) 1830–1845.

[28] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32. doi:10.1023/A:1010933404324.

[29] S. S. K. Djemame, Enabling service-level agreement renegotiation through extending WS-Agreement specification, SOCA (2015) 177–191doi:10.1007/s11761-014-0159-5.

[30] C. Reiss, J. Wilkes, J. L. Hellerstein, Google cluster-usage traces: format + schema, Technical report, Google Inc., Mountain View, CA, USA (Nov. 2011).
URL `https://github.com/google/cluster-data`

[31] R. Nanduri, N. Maheshwari, A. Reddyraja, V. Varma, Job Aware Scheduling Algorithm for MapReduce Framework, in: Proc. of CloudCom, 2011, pp. 724–729. doi:10.1109/CloudCom.2011.112.

[32] S. Cheung, D. K. W. Chiu, S. Till, A Three-Layer Framework for Cross-Organizational e-Contract Enactment, in: WES, Vol. 2512 of LNCS, Springer, 2002, pp. 78–92.

[33] M. P. Papazoglou, The world of e-business: Web-services, workflows, and business transactions, in: International Workshop on Web Services, E-Business, and the Semantic Web, Springer, 2002, pp. 153–173.

[34] H. Ludwig, K. Stamou, M. Mohamed, N. Mandagere, B. Langston, G. Alatorre, H. Nakamura, O. Anya, A. Keller, rsla: Monitoring slas in dynamic service environments, in: International Conference on Service-Oriented Computing, Springer, 2015, pp. 139–153.

[35] R. Engel, S. Rajamoni, B. Chen, H. Ludwig, A. Keller, ysla: Reusable and configurable slas for large-scale sla management, in: 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC), IEEE, 2018, pp. 317–325.

[36] R. B. Uriarte, R. De Nicola, Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards, IEEE Communications Standards Magazine 2 (3) (2018) 22–28.

[37] R. B. Uriarte, R. De Nicola, K. Kritikos, Towards distributed sla management with smart contracts and blockchain, in: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2018, pp. 266–271.

[38] M. Bartoletti, L. Pompianu, An empirical analysis of smart contracts: platforms, applications, and design patterns, in: International Conference on Financial Cryptography and Data Security, Springer, 2017, pp. 494–509.