SLO-ML: A Language for Service Level Objective Modelling in Multi-cloud Applications

Abdessalam Elhabbash
a.elhabbash@lancaster.ac.uk
School of Computing and Communications
Lancaster University, UK

Assylbek Jumagaliyev
asyl.jumagaliyev@zuhlke.com
Zühlke Engineering Ltd., UK

Gordon S. Blair
School of Computing and Communications
Lancaster University, UK

Yehia Elkhatib
School of Computing and Communications
Lancaster University, UK

ABSTRACT

Cloud modelling languages (CMLs) are designed to assist customers in tackling the diversity of services in the cloud market. While many CMLs have been proposed in the literature, they lack practical support for automating the selection of services based on the specific service level objectives of a customer's application. We put forward `SLO-ML`, a novel and generative CML to capture service level requirements and, subsequently, to select the services to honour customer requirements and generate the deployment code appropriate to these services. We present the architectural design of `SLO-ML` and the associated broker that realises the deployment operations. We rigorously evaluate `SLO-ML` using a mixed methods approach. First, we exploit an experimental case study with a group of researchers and developers using a real-world cloud application. We also assess overheads through an exhaustive set of empirical scalability tests. Through expressing the levels of gained productivity and experienced usability, we highlight `SLO-ML`'s profound potential in enabling user-centric cloud brokers. We also discuss limitations as application requirements grow.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Domain specific languages**.

KEYWORDS

Cloud Computing, Cloud Modelling Languages, Domain Specific Language, Service Level Agreements, Service Level Objectives

ACM Reference Format:
Abdessalam Elhabbash, Assylbek Jumagaliyev, Gordon S. Blair, and Yehia Elkhatib. 2019. SLO-ML: A Language for Service Level Objective Modelling in Multi-cloud Applications. In *Proceedings of the IEEE/ACM 12th International Conference on Utility and Cloud Computing (UCC '19), December 2–5, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3344341.3368805

UCC '19, December 2–5, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6894-0/19/12…$15.00
https://doi.org/10.1145/3344341.3368805

1 INTRODUCTION

The growth of the cloud market poses a challenge to its customers who are already overwhelmed with a wide choice of services [8]. The scale as well as heterogeneity of the range of offerings and their real time performance variation are adding more complexity to the decision of cloud service selection [22, 26], particularly in multi-cloud applications [27].

Firstly, the **scale** of cloud services is rapidly growing as more services are offered in the market. A survey of the number of the main cloud providers showed that 198 instance types were offered in 2017 compared to 134 in 2015 [13]. The number of instance types on offer from Microsoft Azure alone increased more than three times between 2015 and 2017.

Secondly, providers adopt **heterogeneous** ways to describe instance specifications, pricing, and service level objectives (SLOs). For instance, Microsoft Azure Cosmos DB and AWS DynamoDB are largely equivalent NoSQL cloud services. They both express the availability SLO[1] in terms of the error rate, i.e. the percentage of failed requests during a billing month. However, Cosmos DB error rate is calculated in one-hour intervals whereas DynamoDB measures it in five minute intervals.

Thirdly, **unexpected performance** may result in substantial financial losses. Recent analysis of some cloud instances shows that performance levels are inconsistent with the promised offerings [3, 15, 19, 24]. For example, as reported in [27], the performance of a standard workload on an AWS `c4.xlarge` instance is quite the same as that of `c4.large` although the former is twice both in specification and cost of the latter.

In view of the above challenges, the process of manually selecting the optimal service can overwhelm a human decision maker. In order to make it easier for customers to select services and deploy applications, cloud modelling languages (CMLs) were proposed (e.g. [6], [23], [17],[25],[20], [28]). They provide means for composing a high level description of a cloud application topology, then automate their deployment accordingly. Such description, also known as Infrastructure as a Code (IaC), declaratively represents the application architecture, interactions, and the types of required cloud services. An orchestrator can then utilise the IaC model to deploy the application on the cloud, as illustrated in Fig. 1.

[1] *Monthly Availability Percentage* for Cosmos DB, and *Monthly Uptime Percentage* for DynamoDB.

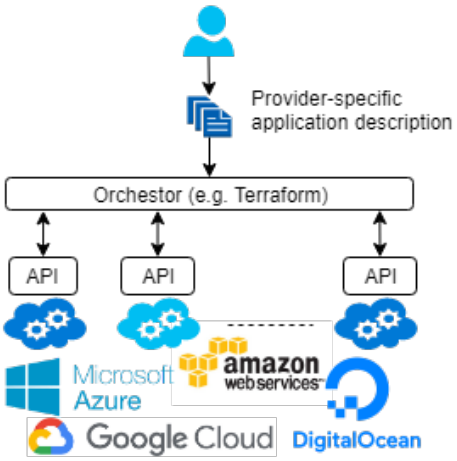

Figure 1: The general architecture of an IaC-based system for semi-automated cloud application deployment.

There are, however, two main drawbacks with current CMLs. First, they lack the support for automated cloud service selection. Customers first need to identify the service(s) they need, which is challenging due to scale and heterogeneity as discussed. Second, there is a lack of support for modelling SLOs of cloud applications. Customers need to manually compare cloud provides service level agreements (SLAs) in order to select a service based on the required SLOs. A few of the current CMLs support such modelling but through standards that are designed primarily for web service providers to specify their services levels, which is unsuitable for use by cloud customers.

In this paper, we aim to address the aforementioned challenges of scale and heterogeneity in addition to the SLO modelling gap. Our aim is to assist cloud customers in selecting cloud services by achieving interoperability between the provider SLAs and the CMLs. Our approach is to base the selection on provider guarantees regarding service performance. That is, we aim to make selection decisions based on a set of SLOs that are part of the SLAs. However, this requires a customer-oriented language for SLO specification and an engine that realises a SLO-driven selection of cloud services.

Therefore, we propose a design of a new language for SLO modelling, *SLO-ML*, that provides a comprehensive syntax for capturing service level requirements, supporting all SLOs currently used by IaaS providers and those specified in industry standards. Through the SLO-ML approach, we aim to raise the level of abstraction provided to cloud customers. We adopt a generative language approach whereby customers specify SLOs (i.e. develop SLO-ML script) for required cloud services regardless of the low level details of those services. Then, the SLO-ML script will be translated into deployment code that is utilised by the orchestrator to deploy.

In addition, we present the architecture of a cloud brokerage system (CBS) that realises the SLO-ML approach [12]. The customer will provide an SLO model to the CBS. The CBS will then parse the models, select the cloud services, and generate the deployment model. The broker can also deploy the application on the selected cloud services.

This paper makes the following contributions:

(1) A novel SLO modelling language, SLO-ML, that supports a *comprehensive set* of SLOs for all types of cloud applications and covering all SLAs in the current IaaS market;
(2) An architecture of a brokerage system that utilises SLO-ML for cloud service selection; and
(3) A mixed-methods evaluation of the applicability of SLO-ML using a real commercial application. Specifically, we assess the added value through a case study experiment with a group of developers of different backgrounds, and we also quantitatively examine the overheads of SLO-ML.

The rest of the paper is organised as follows. §2 motivates the research through a real world application. §3 and §4 present the proposed approach. §5 evaluates SLO-ML through an experimental user study where real developers are asked to utilise SLO-ML for cloud service selection, while §6 evaluates the scalability of the broker architecture. §7 discusses the findings, limitations, and future work. §8 comments on related works and §9 draws conclusions.

2 MOTIVATING EXAMPLE

SolveEngine[2] is a cloud-based problem-solving service. It aggregates thousands of optimisation algorithms and uses artificial intelligence to choose the best ones to solve optimisation problems. Users format their problems in a supported input format and use the SolveEngine API to call the service. SolveEngine then applies the suitable solvers by using Machine Learning techniques, returning the results in JSON format. SolveEngine is a container-based application that consists of two main components, *Solver* and *Database* (see Fig. 2). The Solver processes user requests while the Database stores the processing results. Users can also query the Database to obtain certain data of interest.

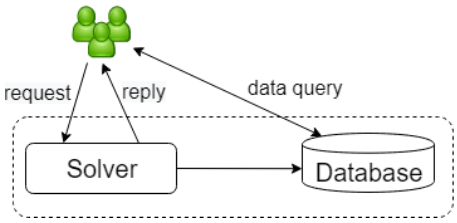

Figure 2: The architecture of the SolveEngine application.

To deploy such 2-component application in the cloud, the application operator (a cloud customer in this case) needs to manually look into different cloud provider SLAs and assess whether or not they satisfy the application's SLOs. This is a time-consuming and challenging task due to the scale and heterogeneity of service offerings as already highlighted. Consider for example an SLO of *Monthly Bandwidth Cost*, which specifies the customer's budget for data transfer between components. The cost calculation depends on several factors such as which provider to use, which service, and in which region. Taking also into account that the cost will be different for different permutations of services, the search space will make the selection decision very challenging for the customer.

3 SLO-ML DESIGN AND CONCEPTS

The key aim of SLO-ML is to provide a comprehensive syntax to capture all possible SLOs that customers may require to specify service

[2]SolverEngine is a commercial product developed by SATALIA, and is available at https://solve.satalia.com/

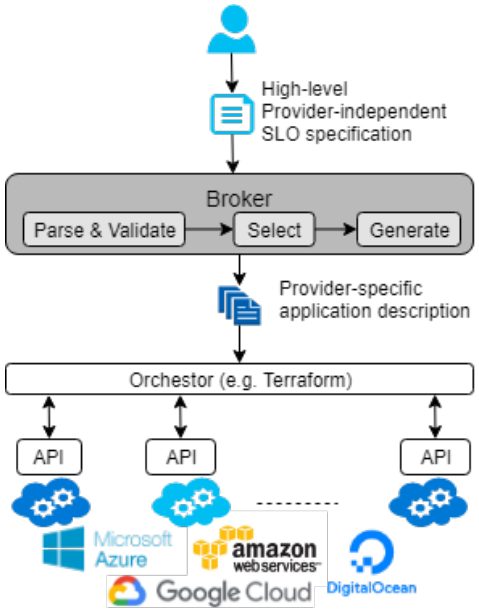

Figure 3: The architecture of using SLO-ML with the broker. Compared to the general IaC architecture that is common in industry (Fig. 1), the proposed architecture provides much more abstraction and transparency.

levels of their applications. For this purpose, SLO-ML enables customers to specify SLOs for each application component. Moreover, SLO-ML supports SLO specification on both single- and multi-cloud deployments.

3.1 Design principles

The design of SLO-ML is based on the following principles:

(1) **Customer-oriented.** SLO-ML is designed to enable customers to specify their high-level operational requirements in a simple declarative syntax. SLO-ML differentiates between two classes of SLOs, namely the service-level SLOs and the application-level SLOs. A service-level SLO represents a quantitative characteristic of the cloud service regardless of the hosted application. Some of the service-level SLOs are specified in the provider SLAs with penalties paid to the customer in case of violation. On the other hand, an application-level SLO represent a quantitative characteristic of the cloud application as perceived by the application client. This kind of SLO cannot be specified in the SLAs as it depends on many aspects of the application such as the application architecture, implementation choices, among other. This implies that the responsibility on satisfying the application-level SLOs is outside of the service provider. This requires an intelligent intermediary system that is able to capture knowledge about the performance of the application when hosted on a certain cloud service and utilise that knowledge to inform the service selection decision.

(2) **Independence**. In order to prevent vendor lock-in, SLO specification needs to be independent of cloud service specification. Furthermore, it needs to be independent of cloud application development technology and implementation details. This is to impose no restrictions on the customer choice of programming models, and to minimise SLO specification changes when adapting the application. In fact, the SLO specifications need to be adapted only when the architecture of the application changes, as the SLOs can be specified per application components.

(3) **Abstraction**. Customers should be able to specify SLOs regardless of the required type of cloud service, such as SaaS, PaaS, FaaS, etc.

(4) **Separation of concerns**. It should be possible to maintain and adapt isolated SLO specification at an application component level. For example, a load-balancing component's SLOs should be separate from those of a data storage element.

(5) **Mapping SLOs** A high-level SLOs which specified by users should be broken down to low-level ones, and then further mapped to the application component level. For example, the response time of a three-tier application consists of processing time for each layers.

(6) **Extensibility**. Extending capability should be simple. In other words, adding a new SLO concept should not require re-engineering of the CML but just adding a human- and machine-readable SLO name along with the appropriate unit and value type, if needed. Obviously, this requires slightly amending the engine that processes the specified model.

3.2 Key elements

At this stage of designing SLO-ML, we adopt textual syntax to represent SLOs. The main elements of the current syntax are: `name`, `type`, `unit`, `operator`, `application`, and `data_flow`.

- `name`: A unique keyword is used to refer to each SLO. The keywords are self-explanatory, making it simple for developers to understand. For example, the keyword `Response_Time` is used to refer to the response time SLO.
- `value`: SLO-ML supports three types of the SLO values: scalar, interval, and categorical. The scalar type is used to specify a numerical value (e.g. availability = 0.99). The interval type is used to specify an upper- and lower-bound of SLO value (e.g. response time between 5ms and 10ms). Categorical types provide a higher level of abstraction for SLO value specification. It allows customers to specify a category (e.g. low, medium, high) instead of specific values or a predefined range, relieving customers from specifying an exact value in case they are not certain. For example, for memory-intensive application, a customer can specify the category high for the `Memory_Size` SLO.
- `unit`: SLO-ML uses a set of keywords that specify the units of measurement of each SLO. For example, the `Migration_Time` SLO is specified using the hours unit. In addition, SLO-ML contains rules for unit-to-unit conversion between units of the same kind.
- `operator`: SLO-ML defines a set of operators that are used to specify the SLO values. This set includes the operators: less than ($<$), less than or equal ($\leq$), greater than ($>$), greater than or equal ($\geq$), equal ($=$), and in (*in*). For instance, *in* can be used to indicate that *response_time* should be in the interval `[5ms,10ms]`.
- `application`: This is to specify the application-level SLOs.
- `data_flow`: This is to specify the directions of data transfer among the application components, which are used in the service selection phase to calculate the expected data transfer costs.

Listing 1 A SLO-ML file example

```
{
   "database_comp": { //component 1
      "SLOs": [
         //service-level SLO
         {   "unit": "",
            "name": "Monthly_uptime_percentage",
            "value": "0.9999",
            "operator": ">="
         },
         //service-level SLO
         {   "unit": "GB",
            "name": "Monthly_egress_bandwidth",
            "value": "2000",
            "operator": "<="
         }
      ],
      "config": {
         "type": "database"
      }
   },
   "solver_comp": {//component 2
      ...
   },
   "application": {
      "SLOs": [
         //application-level SLO
         {   "unit": "\$",
            "name": "Monthly_bandwidth_cost",
            "value": "20",
            "operator": "<"
         }
      ]
   },
   "data_flow": [{
      "from": "solver_comp",
      "to": "database_comp"
   }]
}
```

3.3 Grammar

We adopt JSON syntax [9] for structuring the SLO-ML file (.slo) that defines the required SLO. This definition is structure as a `Map<key,value>` where the `key` is an application component identifier that is defined in the IaC description, while `value` is an array of maps representing the SLOs required for that component. Each map is a `Map<key,value>` where the `key` is one of the elements described in §3.2 and `value` is the corresponding value. Listing 1 shows an example of the SLO specification of a cloud application that consists of two components, *database_comp* and *solver_comp*. The listing shows that *database_comp* requires two SLOs, `Monthly_uptime_percentage` and `Monthly_egress_bandwidth` at the service-level. The application also requires the `Monthly_bandwidth_cost` which specifies the budget for data transfer of the application. The `data_flow` part shows that data will be transferred from the *solver_comp* component to the *database_comp*. The use of an invalid element key, invalid element value, or invalid SLO unit will produce a parsing error.

4 BROKER ARCHITECTURE

We provide an architecture for a cloud broker that realises deployment based on user-provided SLO-ML descriptions.

4.1 Overall approach

Our approach views the cloud application as a set of components, each of which requires a set of SLOs to be specified. The approach builds on existing approaches of modelling cloud applications such as Terraform HCL[3], TOSCA[4], etc. We assume that the customer request consists of SLO model defined using SLO-ML and the broker will parse the model, select satisfying services, and then generate the CML deployment code (HCL, TOSCA, etc.). The broker will then execute the deployment code to deploy the application.

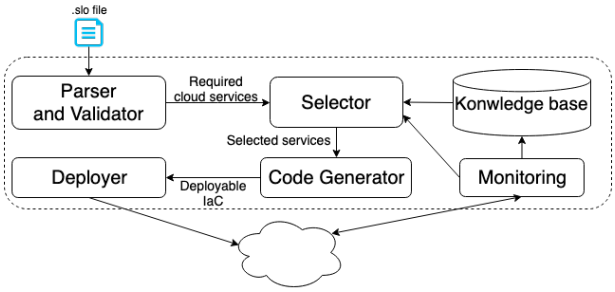

Figure 4: The design of the Realisation Engine.

4.2 Components

Our proposed broker architecture consists of the following main components, as illustrated in Fig. 4.

Parser and Validator parses both the SLO and IaC models to extract the required SLOs for each component. The validation intends to evaluate the SLO specification by checking the correctness of the (i) syntax, (ii) units, and (iii) consistency of the configuration. Syntax validation aims at inspecting the syntax for any errors in using SLO-ML keywords. Unit validation aims to check for any improper use of units. For instance, the unit *days* cannot be used with the `Bandwidth` SLO. Consistency validation ensures that component references in the SLO file correspond to the application components described in the IaC model.

Knowledge Base is a repository that stores information of the cloud instances such as their type, provider, and the service levels. The Knowledge Base also contains monitoring data that represent the real time performance of the cloud services.

Selector selects services that match the required SLOs for each component of the application. In its simplest implementation, the selection is based on provider SLAs. More sophisticated implementations may include intelligent selection using monitoring data and consequent predictions of performance. The selection approach adopted in this paper is founded on quantifying the extent to which each service SLO satisfies the required SLO by assigning a utility value to each SLO. These utilities are aggregated to calculate a utility for each cloud service. The utilities are then maximised to select the optimal service(s).

In order to assign utilities for each service SLO, we use the function shown in eq. (1), which are adapted from a utility model for quantifying volunteer services [11]. The function assigns a minimum utility of 0 to SLOs that satisfy the corresponding required SLO. The service SLOs that do not satisfy the required one receive a utility of -1. The SLOs utilities are then summed up using eq. (2) to calculate the service utility. For each combination of services, an application-level utility is calculated using eq. (3), where corresponding SLOs are aggregated using suitable aggregation functions (e.g. *sum* for

[3]https://www.terraform.io/docs/configuration/syntax.html
[4]https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

cost and *min* for availability). The application-level utilities are then maximised to select the optimal service(s).

$$U_i(S_j) = \begin{cases} 1 - e^{SLO_r - SLO_{ji}}, \text{if } SLO_{ji} \geq SLO_r \\ -1, \text{ otherwise} \end{cases} \quad (1)$$

where SLO_{ji} is the ith SLO of service j, SLO_r is the corresponding required SLO, and $U_i(S_j)$ is the utility of SLO_{ji}.

$$U(S_j) = \sum_{i=1}^{n} U_i(S_j) \quad (2)$$

where $U(S_j)$ is the utility of service j.

$$U_i(comb) = \begin{cases} 1 - e^{APP_r - AGG(SLO_i)}, \text{if } AGG(SLO_i) \geq APP_r \\ -1, \text{ otherwise} \end{cases} \quad (3)$$

where $AGG(SLO_i)$ is the aggregate of ith SLO of the services, APP_r is the corresponding required application-level SLO, and $U_i(comb)$ is the utility of the combination of services.

IaC Code Generator generates the deployment code of the application based on the selected instances. This deployment code is readily deployable with default settings of the selected cloud services, but customers can customise it as they wish.

Deployer receives the deployment code and automates the deployment of the application on the selected cloud instances.

Monitoring records the low level performance metrics of the selected cloud services. The collected data are stored in the Knowledge Base. The metrics are then mapped to the high level SLOs. If the mapping results in violation of an SLO, the violation is reported to the selector to re-select new instances and adapt the application accordingly. It is worth mentioning that the details of monitoring and adaptation are out of this paper's scope as we focus on presenting the modelling language and the realisation architecture.

5 QUALITATIVE EVALUATION

In this section, we evaluate `SLO-ML` using an experimental case study. We first present the experiment setup and the selected case study, then comment on the results.

5.1 Experiment design

Objectives. The experiment aims at evaluating users productivity, in terms of the time required to select cloud services, and accuracy, in terms of the optimally of SLO offerings of the selected service.

Strategy. We compare `SLO-ML` approach against the manual selection of cloud services where users need to manually inspect and compare the service specification and SLA offerings in order to select suitable services for the given use cases. We adopt a controlled experiment approach where participants are given three user cases. The use cases are designed to be simple so that they can conveniently doable by the participants within reasonable experimental time. This controlled experiment strategy design is leveraged to evaluate the interaction of the users with `SLO-ML`. The analysis of this interaction enables the identification of advantages and limits of `SLO-ML` in addition to improvements that can be introduced.

Procedure. The experiment procedure lasts for a maximum of an hour per participant. Each participant is assigned three use cases to select cloud services first manually then by developing and executing `SLO-ML` scripts for each case. All participants performed the same use cases and used the same powerful PC. At the end of the experiment, the participant fills a questionnaire about their experience in programming languages, cloud service selection, cloud application deployment and application modelling languages and tools. Then, each participant is asked to respond to a simple questionnaire using a 5-point Likert scale, to provide feedback about usability and productivity of `SLO-ML` and things to improve.

Task: Participants were given three simple architectures of cloud applications along with their SLO requirements. Each application consists of one or more components, where every component can be deployed on a cloud service that should satisfy the functional and non-functional requirements. Participants are given a list of services that functionally satisfy the components along with the services' SLAs. They need to find services that match the SLO requirements by:

i) following the current approach where provider SLAs are manually inspected to find matching services, and
ii) writing a `SLO-ML` script to be utilised for automated search.

Assistance: Before the experiment commences, participants are introduced to the relevant SLAs of the considered cloud providers, namely, Amazon Web Services, Microsoft Azure, Google cloud, and RackSpace. They are also introduced to `SLO-ML` with a brief quick-start guide (2-3 minutes) and a sample script. During the experiment, additional guidance is provided to any participant requiring assistance for interacting with either the service SLAs and offerings or with `SLO-ML`.

Recruitment: Participants were recruited from Computer Science researchers and students at Lancaster University, as well as from software developers at local startups and incubators. An incentive for participation was offered in the form of an online shopping voucher (value of £10). Overall, 20 participants with varying expertise levels in programming and cloud systems were recruited. These were broken down as 8 researchers, 8 graduate students, and 4 professional developers. Further, 11 of them self-reported high experience (above 5 of a scale from 1 to 7) in JSON, 4 with medium experience (3-4) and 5 with low experience (1-2). Regarding expertise in programming, 8 reported high programming experience (more than 7 years), 5 of medium experience (4-6 years), and 7 with low experience (3 or less years). Finally, 8 self-reported knowledge of cloud application deployment and/or cloud service selection, with AWS and Google Cloud being the most used providers.

5.2 The SolveEngine case study

We exploit the SolveEngine application (introduced in §2) as a real-world case study. We ask the participants to use it under the following three experimental use cases:

5.2.1 Single component: In this case, the SolveEngine application is to be deployed on a hybrid cloud where the Solver component is hosted locally whereas the database component is deployed on a cloud service. The customer needs to select a cloud database service to host the database component. Case 1 in table 1 lists the required SLOs of the database components.

5.2.2 Case 2. In this case both components need to be hosted on the cloud. The customer needs to select a cloud database service to host the *database* component and a compute service to host the *solver* component.

5.2.3 Case 3. This scenario is similar to the previous one. The difference is that the user has application-level SLOs. The participant needs to ensure the aggregate SLOs of the selected services satisfy the application level SLOs. Table 1 shows the bandwidth required between the components and required availability of the application. The customer needs to select a cloud service for each component taking into account the bandwidth budget constraint and needs to aggregate the `Monthly uptime` of the services.

Table 1: SLO requirements of SolveEngine

Component	SLOs
Case 1	
Database	monthly uptime percentage ≥ 0.99
	monthly consistency percentage ≥ 0.9999
	monthly latency attainment percentage ≥ 0.9999
	monthly throughput percentage ≥ 0.9999
Case 2	
Database	monthly uptime percentage ≥ 0.9999
Solver	monthly uptime percentage ≥ 0.9999
Case 3	
Database	monthly uptime percentage ≥ 0.9999
	monthly egress bandwidth ≤ 2 TB
Solver	monthly uptime percentage ≥ 0.9999
	monthly egress bandwidth ≤ 2 TB
Application	monthly uptime percentage ≥ 0.999
	monthly bandwidth cost ≤ $175

5.3 Accuracy results

The accuracy of the selection is evaluated by calculating the distance between the utility of the selected services and the optimal one. For this, the above utility functions (§4) are used to calculate the utility of the selected services.

Fig. 5 compares the accuracy of each participant's selection in both the SLO-ML and manual approaches. In all the three use cases, the results demonstrate that SLO-ML improves selection accuracy. In case 1, which is the simplest case, most of the participants manually selected the optimal service, Microsoft Azure Cosmos DB. This service is the only one with SLA support of the required SLOs (see table 1). Despite the simplicity of the case, three of the participants (P10, P15, and P18) selected wrong services, i.e. services that do not support the required SLOs (namely, `monthly consistency attainment percentage`, `monthly latency attainment percentage`, and `monthly throughput percentage`); hence, these 3 participants scored an accuracy level of 0 for this use case.

The improvement in accuracy is more notable as the complexity increases in case 2, and even more so in case 3. In order to highlight the improvement, we plot the average accuracy in the three cases in Fig. 7. This figure reveals that there has been a sharp decline in accuracy using the manual approach as the complexity of the case increases. This is in contrast to the SLO-ML approach where optimal accuracy is maintained throughout the use cases.

5.4 Productivity results

The productivity of participants is evaluated by calculating the time spent to make a decision of service selection. In the case of SLO-ML, productivity is calculated as the time spent to develop and execute a valid SLO-ML script whereas in the manual approach case it is calculated as the time from the beginning of inspecting the SLA information until deciding on a service.

Fig. 6 compares the productivity of each participant in both approaches. In all three use cases, SLO-ML significantly reduces selection time. More importantly, the more complex the use case is the more significant the improvement is. To better demonstrate this, we plot the distributions of the time spent by participants in the three cases in Fig. 8. What can be clearly seen in this figure is the rapid growth of the completion time of the manual approach as the complexity of the case increases. On the other hand, the growth is much slower in the SLO-ML case, indicating its ability to assist developers in tackling cloud deployment scenarios of complex selection decisions without a high tax on their time.

5.5 Exit interview responses

After the completion of the three cases, the participants were interviewed in order to survey their experience of using SLO-ML. They were asked to answer seven question, four of which aimed to assess productivity and three to assess usability. Responses were collected using a 5-point Likert scale with anchors from 'Strongly disagree' to 'Strongly agree'.

Productivity. Fig. 9 shows the participant feedback on their productivity when using SLO-ML. All the participants agreed that less time would be required when using SLO-ML especially with complex cases of service selection. All of them also agreed that SLO-ML makes service selection easier by automating it as opposed to manual inspection of SLAs. Furthermore, 85% of the participants agree that SLO-ML reduces the possibility of selecting services that do not satisfy the SLO requirements or services that are less optimal.

Usability. Fig. 10 shows the participant feedback on the usability of SLO-ML. The majority of participants found SLO-ML and its concepts and notations easy to use and flexible. A few of the participants (15%) found SLO-ML lacks some features they might need, such as support for other SLOs. One participant (5%) found that SLO-ML restricts their freedom as a cloud customer as it does not leave the final decision of selection to them.

6 SCALABILITY EVALUATION

We now turn our attention to evaluating the feasibility and overheads of our approach. Specifically, we aim through experimental means to identify the factors that contribute to the end-to-end time of generating the deployment code. For this purpose, we evaluate the time required to parse the SLO-ML script, select services, and generate deployment code at different scales. From this, we extract conclusions about the ability of and the requirements for using the SLO-ML approach *at scale*. The used platform is an Intel Core i7 with 16GB RAM running Linux Ubuntu v16.04 and Java SE v1.8.0. Each experiment is repeated 100 times to obtain representative mean values.

6.1 Parsing time

This first experiment focuses on measuring the parsing time. This is the time required for analysing the application structure, in terms of the required components and the connections between them, and also the required component- and application-level SLOs. The main

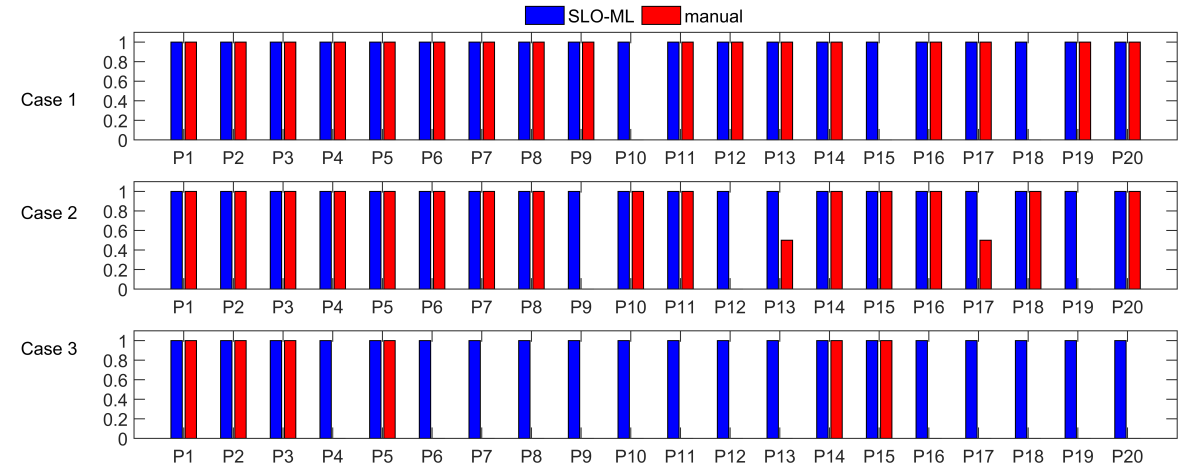

Figure 5: Accuracy of service selection of each case using manual and SLO-ML approaches.

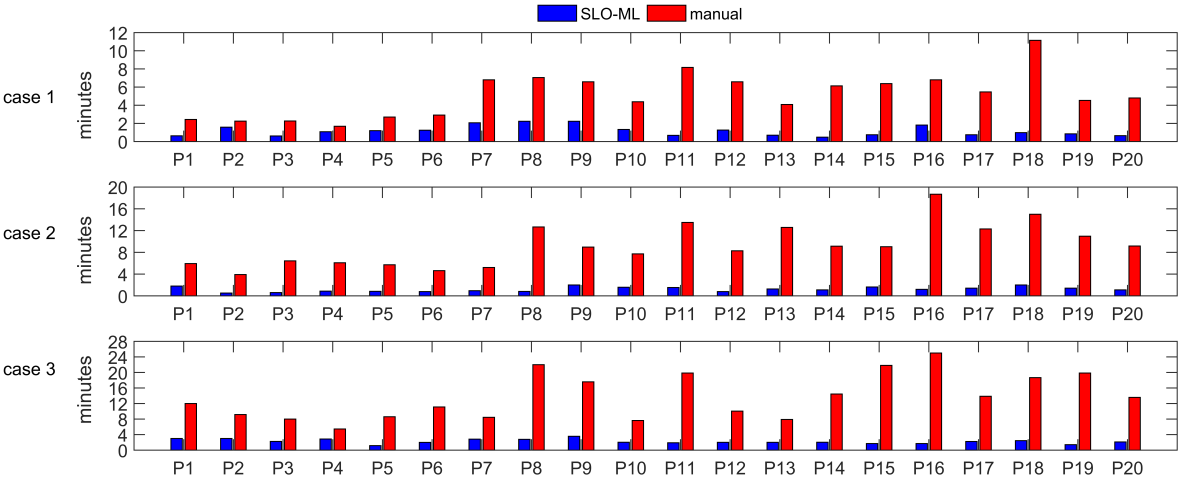

Figure 6: The time spent by participants to complete each case using either approaches.

dimensions affecting the scalability of parsing are the number of components, the number of SLOs and the degree of connectivity. Specifically, we inspect three degrees of application connectivity that correspond to varying application topologies [7]: 'low' represents applications such as Riak with a ring-like topology where a component only connects to one or two other components; 'mid' is analogous to hierarchical hub-and-spoke and other cliquey structures, e.g. MongoDB and Ceph; while 'high' embodies complex applications with highly connected components such as the microservice architecture of the Netflix or Facebook infrastructures.

Fig. 11 plots the average parsing time in milliseconds. We notice that the parsing time increases with the increase of both the number of SLOs and the number of components. In both cases the increase exhibits a linear trend. We notice also that the parsing time increases with the increase in the connectivity degree between the components. However, in all cases the parsing time is practically acceptable, the maximum being ≈1s in a very-large scale deployment of 1000 components and 100 SLOs.

6.2 Deployment code generation time

The next experiment focuses on assessing the time required for the step that follows parsing, i.e. generating application deployment code. The generator is written as a Java program that receives information of the selected services and writes to a file CML-specific lines of code. In this paper, we generate code for Terraform deployer, i.e. the generated code is HCL code [5]. The only dimension affecting the scalability of code generation is the number of components in an application. We vary this dimension between 100 and 1,000 application components – see Fig. 13. We observe a linear trend between code generation time and the number of components. Nevertheless, as the figure depicts, the code generation time is quite insignificant

[5] https://www.terraform.io/docs/configuration/syntax.html

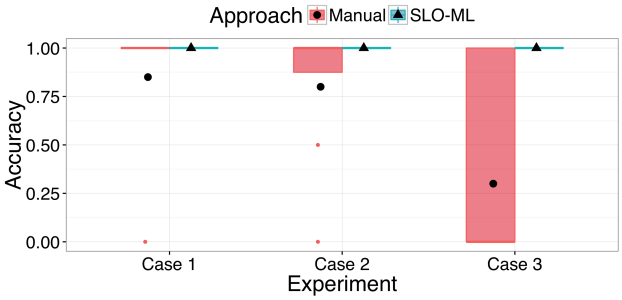

Figure 7: Box-plot and mean accuracy of service selection in each experimental case using either approaches. The traditional manual approach creates selection decisions that are further away from the optimum as application SLOs increase in complexity. Meanwhile, SLO-ML maintains optimal selection in *all* cases.

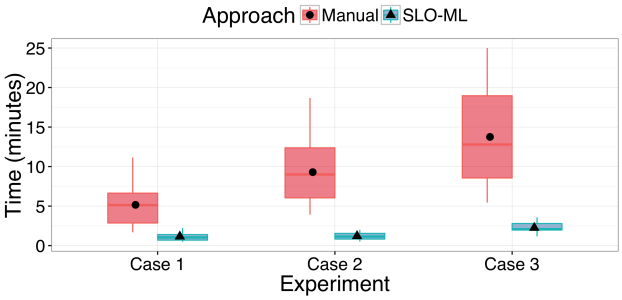

Figure 8: Box-plot and mean time spent by participants to complete each case using either approaches. Using the traditional manual approach, developers needed increasingly more time as application complexity grew. In contrast, SLO-ML allows them to focus only on SLO specification resulting in significantly reduced time, by a factor of 4.5–7.7**x.**

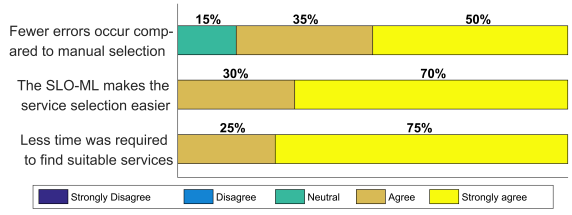

Figure 9: Participant feedback on productivity.

even in the case of a high number of components (e.g. ≈4ms for 1,000 components), underlining SLO-ML's practicality in this regard.

6.3 Selection time

The third experiment focuses on evaluating service selection time, defined as the time required to find a single or set of services that satisfy the application SLO requirements. The main dimensions affecting the scalability of selection are the number of components (i.e. required services), the number of candidate services for each component, and the number of required SLOs. We vary each of these dimensions, plotting the average selection time in milliseconds in Fig. 12. Selection time increases significantly with an increase in the

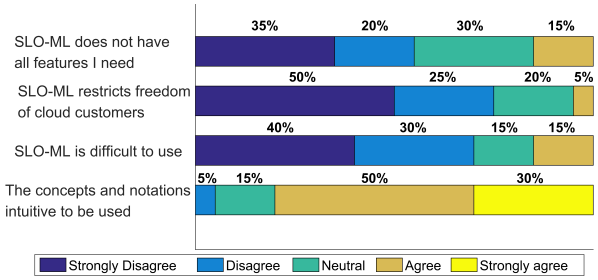

Figure 10: Participant feedback on usability.

number of components, and increases at an exponential rate with the number of candidate services. This implies that the service selection time is the bottleneck of the process especially in the case of a high number of components and/or services. This issue is discussed more in the following section.

7 DISCUSSION

Reflecting on our experiences in developing and evaluating SLO-ML, we draw the following observations and concerns.

7.1 Diversity-induced complexity

During the experimental case study with SolveEngine, many participants tend to ignore (intentionally or mistakenly) some of the options when selecting the services. For example, some participants made decisions based on a subset of the required SLOs, ignoring the effect of others. This led to the selection of services that either do not satisfy the requirements or do but are less optimal ones. This observation was obvious in case 3 where, perhaps due to its complexity, many participants ignored bandwidth price offerings. We observed that this oversight is due to two reasons. The first reason is the difficulty of finding the relevant SLA documents of the required services. Some participants tended to select services the SLAs of which are easily found. The second reason is the diversity in bandwidth pricing schemes, as several cloud providers charge differently based on region and availability zone. This seemed to confuse some participants as they were not able to determine which offering is best to use, while others did not want to spend time to inspect all of the offerings and opted for randomly selected one. Although complexity is clearly to blame for such behaviour, it clearly can lead to wrong or sub-optimal decision making.

Furthermore, some participants made wrong decisions (i.e. they selected services that do not satisfy the required SLOs) due to lack of knowledge. They either looked into irrelevant SLAs or they were confounded by the heterogeneous terminology adopted by different providers to express the same SLOs.

7.2 Scalability

Service selection time is the major contributor to the end-to-end time of processing SLO-ML script. For the worst experimental case illustrated in Fig. 12 where an application has 7 different components and 100 SLOs, the selection time between 100 services is 45 milliseconds which is quite reasonable. Though, this overhead would increase quite rapidly as complexity grows. In this paper, we implemented a naïve exhaustive search to find the optimal service. However, in the case of applications of a higher scale, more scalable selection

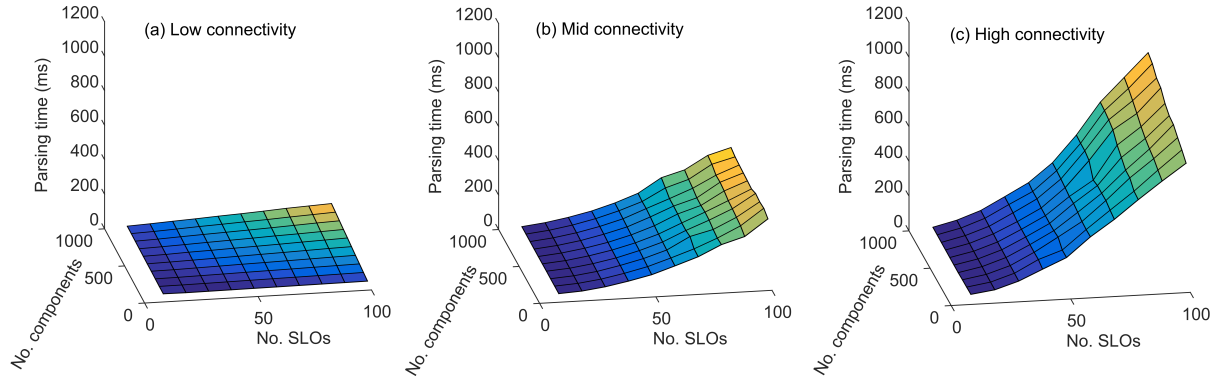

Figure 11: Parsing time of SLO-ML script with varied scales and connectivity degrees between components.

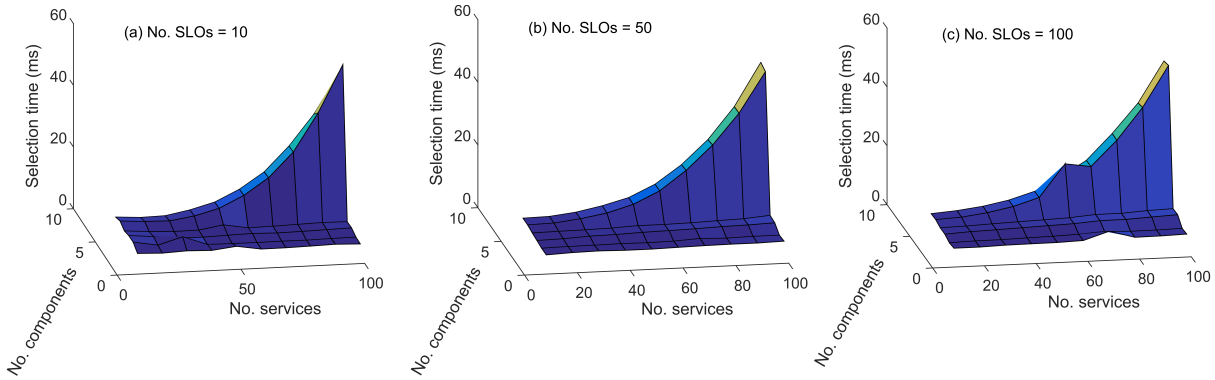

Figure 12: SLO-ML's selection time as the complexity of an application and the number of its SLOs grow.

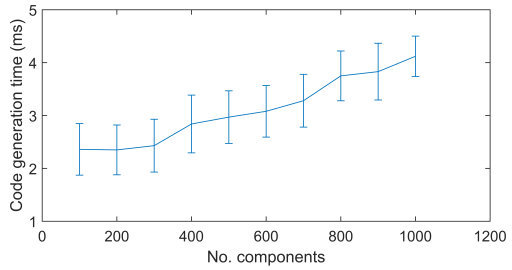

Figure 13: SLO-ML code generation time for applications with different numbers of components.

algorithms are required. This is beyond the scope of this paper, but luckily the web service selection literature is rich of such selection algorithms [10].

7.3 Experimental validity

As is common with experimental case study designs, external validity (i.e. the ability to generalise the results) is naturally impaired to an extent in order to attain higher internal validity (i.e. validating the cause-effect inference). However, by choosing a real-world application that is representative of a range of user-facing cloud applications, we are satisfied that our results are indicative of the significant value added by SLO-ML.

7.4 Future directions

Currently, SLO-ML only caters to SLOs that are supported by provider SLAs. For even more abstract support of application needs, this needs to be extended to include application-specific SLOs that cannot be guaranteed by the provider. An example is the `Completion time` SLO that specifies a deadline for a certain job. Such extension, however, requires run-time monitoring of the application and continuous assessment of the SLO's satisfaction. This also requires the development of adaptation techniques to adapt service selection in case the current selection fails to satisfy the SLOs. In turn, this also requires the accumulation of knowledge about services and application performance, and the utilisation of such knowledge to predict performance before making the selection decision. These issues are the focus of our ongoing work to extend the SLO-ML approach.

8 RELATED WORK

A CML uses modelling concepts to raise the level of abstraction, enabling customers to describe their specific application needs that could then be systematically matched against cloud service offerings. As such, CMLs have been used to design different aspects of cloud application engineering [4]. Many CMLs (e.g. Blueprint, CAML, CloudDSL, GENTL, CAMEL [25]) address the deployment of services and application components to a cloud environment by describing deployment configurations. Meanwhile, other CMLs (e.g. CloudMIG, StratusML, TOSCA) deal with the automation of

cloud resources provisioning, application migration to the cloud and re-configuration of provisioned cloud services.

A prominent example is the Topology and Orchestration Specification for Cloud Applications (TOSCA) [2], an OASIS standard for describing the structure of cloud applications (i.e. components and relationships) in XML format. Similar efforts include GENTL [1], CloudML-UFPE [16], and CloudML-SINTEF [5].

Blueprint [23] provides concepts for representing service-based applications to facilitate deployment and migration on cloud services. The provided concepts also allow for the representation of different cloud service offerings. MULTICLAPP [17] introduces a UML-based profile to model components that can be annotated with deployment information. StratusML [18] adopts a similar approach. CAMEL [25] can be viewed as a 'superset' CML as it integrates and extends existing DSLs. ARGON [28], addresses the issue of abstracting the complexity of using CMLs by enabling users to specify infrastructure resources then generating deployment code, similar to the SLO-ML approach. CadaML [21] is used to manage multi-tenant architecture evolution by transforming an abstract model into the appropriate code for different cloud data storage types.

There are two main shortcomings of the above and other CMLs. First, they require the customer to develop a service-specific IaC model, which means customers need to manually select the cloud services. Such IaC model can be complex to develop from scratch, especially for large-scale applications. Second, they provide limited support for modelling customer SLOs. Instead, they seem to have been designed with a simplistic representation of the provider's perspective not that of the customer. For instance, Blueprint assumes the presence of a marketplace where providers can publish their service descriptions as WS-Policy files. WS-Policy is intended to specify non-functional properties of unary web services, not the complex cloud services customers use today. Furthermore, such assumed marketplace does not exist [13, 14], so such CMLs are of little practical value in real-world deployments. SLO-ML addresses the above shortcomings and raises the level of abstraction provided to cloud customers.

9 CONCLUSION

We presented `SLO-ML`, the first cloud modelling language to automate the selection of cloud services to satisfy customer service level objectives (SLOs) and generate the appropriate deployment code. `SLO-ML` is specifically designed to capture a wide range of SLOs from customers. Our findings from an experimental case study suggest that the raised level of abstraction provided by `SLO-ML` results in significant improvements in both developer productivity and optimal service selection. We also identified the limitations of `SLO-ML`, which poses a number of research questions for future work in this area.

ACKNOWLEDGMENTS

This work was supported by the Adaptive Brokerage for the Cloud (ABC) project, UK EPSRC grant EP/R010889/1.

REFERENCES

[1] Vasilios Andrikopoulos, Anja Reuter, Santiago Gómez Sáez, and Frank Leymann. 2014. A GENTL Approach for Cloud Application Topologies. In *Service-Oriented and Cloud Computing*. 148–159.

[2] Ankita Atrey, Hendrik Moens, Gregory Van Seghbroeck, Bruno Volckaert, and Filip De Turck. 2015. *An overview of the OASIS TOSCA standard: Topology and Orchestration Specification for Cloud Applications.* Technical Report. IBCN-iMinds, Department of Information Technology.

[3] Matt Baughman, Ryan Chard, Logan Ward, Jason Pitt, Kyle Chard, and Ian Foster. 2018. Profiling and Predicting Application Performance on the Cloud. In *IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. 21–30. https://doi.org/10.1109/UCC.2018.00011

[4] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* 51, 1, Article 22 (Feb 2018), 38 pages. https://doi.org/10.1145/3150227

[5] Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. 2015. The Evolution of CloudML and its Applications. In *Workshop on MDE on and for the Cloud.*

[6] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2014. *TOSCA: Portable Automated Deployment and Management of Cloud Applications.* Springer, 527–549. https://doi.org/10.1007/978-1-4614-7535-4_22

[7] Simon Bouget, Yérom-David Bromberg, Adrien Luxey, and François Taiani. 2018. Pleiades: Distributed Structural Invariants at Scale. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 542–553. https://doi.org/10.1109/DSN.2018.00062

[8] Cloud Standards Coordination (CSC). 2016. *CSC Phase 2: Cloud Computing Users Needs - Analysis, conclusions and recommendations from a public survey.* Special Report 003 381 V2.1.1. The European Telecommunications Standards Institute (ETSI). 12–19 pages. http://csc.etsi.org/phase2/UserNeeds.html

[9] Douglas Crockford. 2006. *The application/json Media Type for JavaScript Object Notation (JSON).* Internet RFC 4627.

[10] Schahram Dustdar and Wolfgang Schreiner. 2005. A survey on web services composition. *International journal of web and grid services* 1, 1 (2005), 1–30.

[11] Abdessalam Elhabbash, Rami Bahsoon, Peter Tino, and Peter R. Lewis. 2014. A Utility Model for Volunteered Service Composition. In *International Conference on Utility and Cloud Computing (UCC)*. IEEE/ACM. https://doi.org/10.1109/UCC.2014.43

[12] Abdessalam Elhabbash, Yehia Elkhatib, Gordon Blair, Yuhui Lin, and Adam Barker. 2019. A Framework for SLO-driven Cloud Specification and Brokerage. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 666–667. https://doi.org/10.1109/CCGRID.2019.00085

[13] Abdessalam Elhabbash, Faiza Samreen, James Hadley, and Yehia Elkhatib. 2019. Cloud Brokerage: A Systematic Survey. *Computing Surveys* 51, 6, Article 119 (Jan 2019), 28 pages. https://doi.org/10.1145/3274657

[14] Yehia Elkhatib. 2016. Mapping Cross-Cloud Systems: Challenges and Opportunities. In *Conference on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, 77–83.

[15] Nadir Ghrada, Mohamed Faten Zhani, and Yehia Elkhatib. 2018. Price and Performance of Cloud-hosted Virtual Network Functions: Analysis and Future Challenges. In *PVE-SDN*.

[16] Glauco Estacio GonÃğalves, Patricia Endo, Marcelo Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mangs. 2011. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. *CloudCom* (2011).

[17] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. 2013. A UML Profile for Modeling Multicloud Applications. In *Service-Oriented and Cloud Computing*. 180–187.

[18] Mohammad Hamdaqa and Ladan Tahvildari. 2015. Stratus ML: A Layered Cloud Modeling Framework. In *IEEE International Conference on Cloud Engineering*.

[19] K. Hwang, X. Bai, Y. Shi, M. Li, W. G. Chen, and Y. Wu. 2016. Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies. *Trans. Parallel Distrib. Syst.* 27, 1 (2016), 130–143. https://doi.org/10.1109/TPDS.2015.2398438

[20] Assylbek Jumagaliyev and Yehia Elkhatib. 2019. CadaML: A Modeling Language for Multi-Tenant Cloud Application Data Architectures. In *IEEE International Conference on Cloud Computing (CLOUD).*

[21] Assylbek Jumagaliyev and Yehia Elkhatib. 2019. A Modelling Language to Support Evolution of Multi-Tenant Cloud Data Architectures. In *IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MODELS).*

[22] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *26th International Conference on World Wide Web*. 83–91. https://doi.org/10.1145/3038912.3052707

[23] Dinh Khoa Nguyen, Francesco Lelli, Yehia Taher, Michael Parkin, Mike P. Papazoglou, and Willem-Jan van den Heuvel. 2011. Blueprint Template Support for Engineering Cloud-Based Services. In *Towards a Service-Based Internet*. 26–37.

[24] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. 2012. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *Conference on Hot Topics in Cloud Computing (HotCloud)*. USENIX.

[25] Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, JÃűrg Domaschka, Frank Griesinger, Daniel Seybold, Daniel Romero, Michal Orzechowski, Georgia Kapitsaki, and Achilleas Achilleos. 2017. *The cloud application modelling and execution language (CAMEL).* Technical Report. Universität Ulm. https://doi.org/10.18725/oparu-4339

[26] Maria Salama, Amir Zeid, Ahmed Shawish, and Xiaohong Jiang. 2014. A novel QoS-based framework for cloud computing service provider selection. *International Journal of Cloud Applications and Computing* 4, 2 (2014), 48–72.

[27] Faiza Samreen, Yehia Elkhatib, Matthew Rowe, and Gordon S. Blair. 2016. Daleel: Simplifying cloud instance selection using machine learning. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 557–563. https://doi.org/10.1109/NOMS.2016.7502858

[28] Julio Sandobalin, Emilio Insfran, and Silvia Abrahao. 2017. An Infrastructure Modelling Tool for Cloud Provisioning. In *IEEE International Conference on Services Computing (SCC)*. 354–361. https://doi.org/10.1109/SCC.2017.52