SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs

Thomas Pusztai
Andrea Morichetta
Víctor Casamayor Pujol
Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

Stefan Nastic
Reinvent Labs GmbH
Vienna, Austria
snastic@reinvent-group.at

Xiaoning Ding
Deepak Vij
Ying Xiong
Futurewei Technologies, Inc.
Santa Clara, CA, USA
firstname.lastname@futurewei.com

Abstract—Service Level Objectives (SLOs) allow defining expected performance of cloud services, such that cloud service providers know what they guarantee and service consumers know what to expect. Most approaches focus on low-level SLOs, closely related to resources, e.g., average CPU or memory usage, and are usually bound to specific elasticity controllers. We present SLO Script, a language and accompanying framework, motivated by real-world, industrial needs to allow service providers to define complex, high-level SLOs in an orchestrator-independent manner. The main features of SLO Script include: i) novel abstractions (*StronglyTypedSLO*) with type safety features, ensuring compatibility between SLOs and elasticity strategies, ii) abstractions that enable decoupling of SLOs from elasticity strategies, iii) a strongly typed metrics API, and iv) an orchestrator-independent object model that enables language extensibility. We present a case study about a real-world, cloud-native application and evaluate our language while implementing a realistic Cost Efficiency SLO.

Index Terms—cloud computing, SLO, elasticity, metrics, orchestrator independence

I. INTRODUCTION

In cloud computing, it is common for providers and consumers to agree on Service Level Agreements (SLAs) to define bounds within which a certain cloud service has to operate [1]. An SLA consists of one or more *Service Level Objectives (SLOs)*, where an SLO is a "commitment to maintain a particular state of the service in a given period" [2]. Often, SLOs provide directly measurable capacity guarantees, such as available memory. However, service consumers usually prefer to get performance guarantees, which can be related to business-relevant Key Performance Indicators (KPIs). The vast majority of today's cloud providers offer only rudimentary support for SLOs, which means that customers, who want to have a high-level SLO, need to manually map it to directly measurable low-level metrics, such as CPU or memory [3].

Elasticity is a flagship property of cloud computing. Herbst et al. [4] define it as: "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point

in time the available resources match the current demand as closely as possible." This definition already suggests that today's cloud offerings usually deal with resource elasticity, i.e., adding resources, such as CPU, memory, or additional service instances, when the demand is high and removing resources when the demand is low. However, elasticity is not limited to the resource dimension, instead it has three dimensions – the other two being cost elasticity, i.e., the amount of money a consumer is willing to pay for a service, and quality elasticity, e.g., the desired precision of the output data of a machine learning system [5]. We define an *elasticity strategy* as a sequence of altering actions that adjust the amount of resources provisioned for a workload, their type, or both. Additionally, it can change the workload configuration, i.e., alter quality parameters, as well. Thus, an elasticity strategy is capable of affecting all three elasticity dimensions.

From a business perspective, it is important to be able to map business goals to measurable KPIs, which, again, must be translated into SLOs. With a low-level average CPU usage SLO this is not easily possible. A high-level SLO that combines multiple elasticity dimensions, e.g., by combining resource usage with the total cost of the system, is better suited for this purpose.

In this paper we continue our work envisioned in [3], which we refer to as Polaris SLO Cloud (Polaris) project, and present *SLO Script*[1], a language and accompanying framework, which permits service providers to define complex SLOs on their services and service consumers to configure and apply them to their workloads. Our main contributions with SLO Script include:

1) Novel abstractions (*StronglyTypedSLO*) with type safety features that ensure compatibility between workloads, SLOs, and elasticity strategies.
2) Language constructs: *ServiceLevelObjectives*, *ElastcityStrategies*, and *SloMappings* enable decoupling of SLOs from elasticity strategies, to promote reuse and increase the number of possible SLO/elasticity strategy combinations. Details are provided in Sections III-A and III-B.

[1]SLO Script is referred to as "SLO Elasticity Policy Language" in [3].

1

3) Strongly typed metrics API that boosts productivity when writing queries, presented in Section III-C.
4) Orchestrator-independent object model that promotes extensibility, as detailed in Section III-D.

The remainder of this paper is structured as follows: Section II introduces a motivating use case to explain why SLO Script is needed and lists the research challenges and requirements for our language, Section III portrays the design and main abstractions of SLO Script, Section IV describes the runtime mechanisms, Section V evaluates our SLO Script on the motivating use case on a Kubernetes implementation, Section VI discusses related work, and Section VII outlines future work and concludes the paper.

II. MOTIVATION

In the open source[2] Polaris project [3], we aim to make SLOs as the first class entities and bring multi-dimensional elasticity capabilities to the cloud computing environment. Polaris itself is part of Linux Foundation's Centaurus project[3], a novel open-source platform targeted towards building unified and highly scalable public or private distributed cloud infrastructure and edge systems.

A. Motivating Use Case

To motivate our approach, we present a real-world cloud use case, featuring a cloud service provider that wants to offer a Content Management System (CMS) in the form of Software-as-a-Service to its customers. Gentics Mesh[4] is an open source headless CMS, i.e., a CMS that is primarily used through its REST API, incorporated into a web application as a content source. The *service provider* offers customers the CMS-as-a-service for deployment on the cloud infrastructure. *Service consumers* are customers, who integrate the CMS-as-a-service into their applications. Fig. 1 shows an overview of the use case. The deployment consists of two major components: the CMS itself and an ElasticSearch[5] database. Both need to be managed transparently for the service consumers. Each service exposes one or more metrics, e.g., CPU usage, response time, or complex metrics like cost efficiency. The service provider defines a set of SLOs that are supported by the service.

The more requests a service should be able to handle per second, the more resources it needs, and thus, the more expensive it becomes. Different service consumers have different needs with regards to requests per second and are willing to pay different prices for these guarantees. However, for most of them it is difficult, if not impossible, to specify a low-level, resource-bound SLO that delivers the best performance within their budget. This is mainly due to a lack of detailed technical understanding of the services and because a resource-bound SLO only captures a single elasticity dimension. Instead, the service consumers would prefer simply specifying a high-level *cost efficiency* of the microservices. The cost efficiency
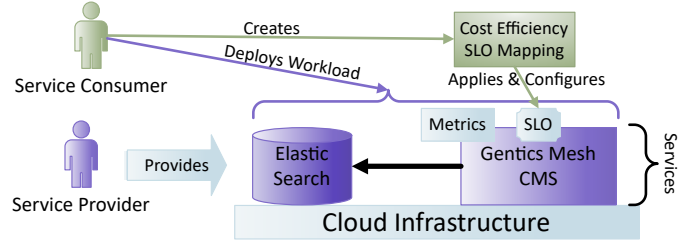
[2]https://polaris-slo-cloud.github.io

[3]https://www.centauruscloud.io

[4]https://getmesh.io

[5]https://www.elastic.co/elasticsearch/

Fig. 1: Gentics Mesh CMS cloud scenario overview.

is usually defined as the number of requests per second served faster than N milliseconds divided by the total cost of the microservice [6]. To achieve this with our approach, the service consumer only needs to perform a set of simple tasks. The service consumer deploys Gentics Mesh-as-a-service – we refer to this deployment as a *workload*. To apply the cost efficiency SLO to the workload, the service consumer creates an *SLO mapping*, which associates an SLO offered by the service provider with a workload of the service consumer. After creating the SLO mapping, the service consumer is finished, since the cloud will be responsible for automatically performing elasticity actions to ensure that the SLO is fulfilled.

Therefore, by allowing service consumers to specify a high-level SLO such as cost efficiency [7], our approach enables service consumers to specify a value that can be easily communicated to the non-technical, management layers of their companies, which is important for approving the budget and checking conformance with the business goals. The complex task of mapping this cost efficiency to low-level resources and performing complicated elasticity actions to achieve the SLO is left to the service provider, who knows the infrastructure and the requirements of the offered services. Using SLO Script, the service provider is able to efficiently use the know-how about the services to implement these complex SLOs.

B. Research Challenges

SLO Script addresses the following research challenges:

RC-1 *Enable complex elasticity strategies*: The majority of systems provide only simple elasticity strategies, with horizontal scaling being the most common [8]. For example, Kubernetes[6], which has the most production-level services among commonly used container orchestration systems [9], usually ships with the Horizontal Pod Autoscaler (HPA) [10]. However, some cloud providers have shown little or no further increase in application performance beyond certain instance counts [11]. Thus, a complex elasticity strategy, which, e.g., combines horizontal and vertical scaling, may achieve better results.

RC-2 *Enable high-level SLOs, based on complex metrics*: The majority of metrics used nowadays is directly measurable at the system or application level, such as CPU and memory utilization, or response time [8], [12], [13]. For example, HPA uses the average CPU utilization of all pods of a workload. We define a *composed metric* as a metric that can be obtained by aggregating and composing other

[6]https://kubernetes.io

2

metrics. In HPA they can be supplied through a custom metrics API or an external metrics API[7]. Both entail the registration of a custom API server, called an *adapter API server*, to which the Kubernetes API can proxy requests, thus, leading to additional development and maintenance effort. The custom metrics API [14] and the external metrics API [15] allow exposing arbitrary metrics (e.g., from the monitoring solution Prometheus[8]) as Kubernetes resources. However, apart from summing all values if an external metric matches multiple time series [16], the computation or aggregation of these metrics must be implemented by the adapter API server. HPA allows specifying multiple metrics for scaling, but it calculates a desired replica count for each of them separately and then scales to the highest value [17]. Yet, a high-level SLO is valuable, as shown in our motivating use case.

RC-3 *Decoupling of SLOs from elasticity strategies*: If systems provide only a single elasticity strategy, e.g., horizontal scaling in HPA, it usually means that the SLO is tied to that strategy, making the system rigid and inflexible. A tight coupling between SLO evaluation and elasticity strategy in the same controller, would require re-implementing every needed SLO in every elasticity controller, leading to duplicate code and difficult maintenance. Furthermore, a specific SLO may not yet have been implemented on a certain elasticity controller, albeit being needed by a consumer.

RC-4 *Unified API for multiple metrics sources*: Each major time series database has its own query language, e.g., Prometheus has PromQL, InfluxDB[9] has Flux, and Google Cloud has MQL[10]. Thus, an implementation in a particular language ties the SLO to a certain DB, because there is no common query language for time series databases, like SQL is for relational databases.

RC-5 *Cloud vendor independence*: Common autoscaling solutions are tied to a specific orchestrator or cloud provider. All major cloud vendors have their own specific configuration of autoscaling, e.g, AWS [18], Azure [19], and Google Cloud [20] all have their own, non-portable way of configuring an autoscaler – fostering vendor lock-in. HPA, although not being tied to a particular cloud provider, is still specific to Kubernetes.

C. Language Requirements Overview

SLO Script is at the heart of the Polaris project and will ultimately support the definition and implementation of metrics, SLOs, and elasticity strategies, each of which may be generic or specifically tailored to a particular service.

An *SLO* evaluates metrics to determine whether the system conforms to the expectations defined by the service consumer. When the SLO is violated (reactive triggering) or when it

is likely to be violated in the near future (proactive triggering), it may trigger an elasticity strategy. These can range from a simple horizontal scaling strategy, over more complex strategies that combine horizontal and vertical scaling, to application-specific elasticity strategies that combine scaling with adaptations of the service's configuration.

The goal is a language that presents a significant usability improvement over raw configurations that rely on YAML or JSON. To this end, the language must support higher-level abstractions than raw configuration files and provide type safety, which reduces errors and boosts productivity.

The requirements derived from our motivating use case and the core objectives of SLO Script are as follows:

1) Allow service consumers to configure and map an SLO to a workload.
2) Allow service consumers to choose any compatible elasticity strategy when configuring an SLO (loose coupling).
3) Allow SLOs to instantiate, configure, and trigger the elasticity strategy chosen by the service consumer.
4) Support the definition of composed metrics.
5) Support the definition of elasticity strategies.
6) Ensure compatibility between SLOs and elasticity strategies at the time of writing (i.e., type safety).
7) The SLO Script core has to be orchestrator-independent.
8) Plug into specific orchestrators using adapter libraries.
9) Service providers should be able to focus on the business logic of their metrics, SLOs, and elasticity strategies.
10) Present a DB-independent API for querying metrics.
11) Support packaging metrics, SLOs, and elasticity strategies into plugins.

SLO Script supports the use of any metrics source using adapters and elasticity strategies developed in any language, as long as their input data types match the output data types of the SLOs. This allows reusing an elasticity strategy written in a different language, e.g., because an orchestrator-specific API client may only be available in that language.

The next section will explain the design of the SLO Script language and how it achieves orchestrator-independence.

III. SLO SCRIPT LANGUAGE DESIGN & MAIN ABSTRACTIONS

In this section we describe how SLO Script provides the main contributions announced in the introduction section: 1) high-level StronglyTypedSLO abstractions with type safety features, 2) constructs to enable decoupling of SLOs from elasticity strategies, 3) a strongly typed metrics API, and 4) an orchestrator-independent object model that promotes extensibility. The first two contributions are treated incrementally by the subsections III-A and III-B, the third contribution is presented in subsection III-C, and the fourth contribution is discussed in subsection III-D.

A. SLO Script Overview & Language Meta-Model

SLO Script consists of high-level, domain-specific abstractions and restrictions, which constitute a language abstraction. It does not provide its own textual syntax, but uses TypeScript

---

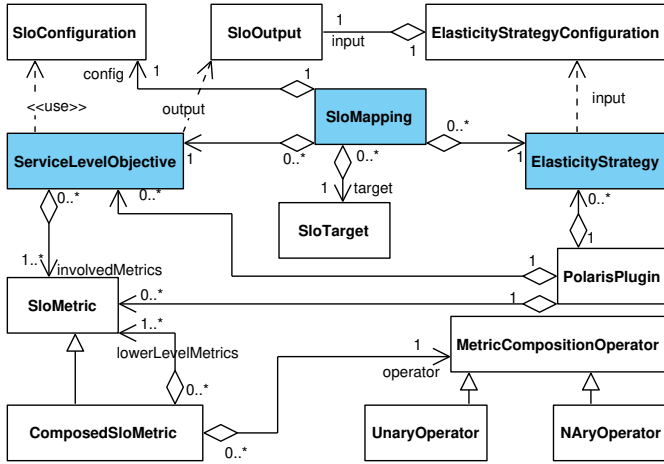[7]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-metrics-apis

[8]https://prometheus.io

[9]https://www.influxdata.com

[10]https://cloud.google.com/monitoring/mql/reference

Fig. 2: SLO Script meta-model (partial view).

Fig. 3: Type safety provided by StronglyTypedSLO.

as its base. Using a publicly available and well-supported language, increases the chances for SLO Script to be accepted by developers and reduces maintenance effort, because language and compiler maintenance is handled by the TypeScript authors. The requirements in the previous section result in the meta-model for SLO Script, depicted as a UML class diagram, in Fig. 2.

1) `ServiceLevelObjective` is one of the central constructs of the SLO Script language. An example instance is the `CostEfficiencySlo`, which implements the cost efficiency scenario described in Section II-A. An instance of the `ServiceLevelObjective` construct defines and implements the business logic of an SLO and is configured by the service consumer using an `SloConfiguration`. The `ServiceLevelObjective` uses instances of `SloMetric` to determine the current state of the system and compare it to the parameters specified by the service consumer in the `SloConfiguration`. The metrics are obtained using our strongly typed metrics API, which abstracts a monitoring system, such as Prometheus. The metrics may be low-level metrics, directly observable on the system or higher-level metrics (instances of `ComposedSloMetric`) or a combination of both. Every evaluation of the `ServiceLevelObjective` produces an `SloOutput`, which describes how much the SLO is currently fulfilled and is used as a part of the input to an `ElasticityStrategy`. Both, `ServiceLevelObjective` and `ElasticityStrategy`, define the type of `SloOutput` they produce or require respectively, which is one of the types needed for determining compatibility among them.

2) The `ElasticityStrategy` construct represents the implementation of an elasticity strategy. It executes a sequence of elasticity actions to ensure that a workload fulfills an SLO. Elasticity actions may include, e.g., provisioning or deprovisioning of resources, changing the types of resources used, or adapting the configuration of a service. The input to an `ElasticityStrategy` is a corresponding `Elasticity-StrategyConfiguration`, consisting of the `SloOutput` produced by the `ServiceLevelObjective` and static configuration provided by the consumer.
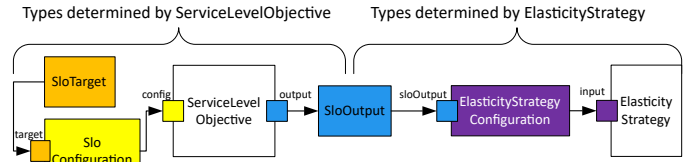
There is no direct connection between a `ServiceLevel-Objective` and an `ElasticityStrategy`, which clearly shows that these two constructs are decoupled from each other. A connection between them can only be established through additional constructs, i.e., `SloOutput` or `SloMapping`.

3) The `SloMapping` construct is used by the service consumer to establish the relationship between a `ServiceLevel-Objective`, an `ElasticityStrategy`, and an `SloTarget`, i.e., the workload to which the SLO applies. The `SloMapping` contains the `SloConfiguration`, which are the SLO-specific bounds that the consumer can define, the `SloTarget`, i.e., the workload to which the SLO is applied, and any static configuration for the chosen `ElasticityStrategy`.

B. StronglyTypedSLO

When defining a `ServiceLevelObjective` using SLO Script's StronglyTypedSLO mechanism, the service provider must first create an `SloConfiguration` data type that will be used by the service consumer to configure the `Service-LevelObjective` and an `SloOutput` data type to describe its output. While each `ServiceLevelObjective` will likely have its own `SloConfiguration` type, it is recommended to reuse an `SloOutput` data type for multiple `ServiceLevel-Objectives` to allow for loose coupling between `Service-LevelObjectives` and `ElasticityStrategies`.

To create the actual SLO, a service provider must instantiate the `ServiceLevelObjective` meta-model construct, represented by the `ServiceLevelObjective` TypeScript interface. It takes three generic parameters to enable type safety: `C` denotes the type of `SloConfiguration` object that will carry the parameters from an `SloMapping`, `O` is the type of `SloOutput`, which will be fed to the elasticity strategy, and `T` is used to define the type of target workload the SLO supports. An `ElasticityStrategy` uses the same mechanism to define the type of `SloOutput` that it expects as input.

Fig. 3 illustrates how the type safety feature of SLO Script works. There are two sets of types: those determined by the `ServiceLevelObjective` and those determined by the `ElasticityStrategy`. The `ServiceLevelObjective` defines that it needs a certain type of `SloConfiguration` (indicated by the yellow color) as configuration input. The `SloConfiguration` defines the type of `SloTarget` (orange), which may be used to scope the SLO to specific types of workloads. The `ElasticityStrategy` defines its type of `ElasticityStrategyConfiguration` (purple), which, in turn, specifies the type of `SloOutput` (blue) that is required by the `ElasticityStrategy`.

Thus, the bridge between these two sets is the `SloOutput` type. Once the service consumer has chosen a particular

`ServiceLevelObjective` type, the possible `SloTarget` types are fixed because of the `SloConfiguration`. Since the `ServiceLevelObjective` defines an `SloOutput` type, the set of compatible elasticity strategies is composed of exactly those `ElasticityStrategies` that have defined an `ElasticityStrategyConfiguration` with the same `Slo-Output` type as input.

Type checking is especially useful in enterprise scenarios, where hundreds of SLOs need to be managed. Using YAML or JSON files for this purpose provides no way of verifying that the used SLOs, workloads, and elasticity strategies are compatible, while SLO Script provides this feature. Furthermore, using a type safe language yields significant time savings when a set of SLOs and their mappings need to be refactored.

The SLO Script runtime invokes the SLO instance at configurable intervals to check if the SLO is currently fulfilled or if the elasticity strategy needs to take corrective actions. It may simply check if the metrics currently match the requirements of the SLO or it can use predictions and machine learning to determine if the SLO is likely to be violated in the near future and thus take proactive actions through the elasticity strategy. The result of this operation is an instance of the defined `SloOutput` type, which is returned asynchronously.

C. Strongly Typed Metrics API

The strongly typed metrics API provides two types of abstractions: i) *raw metrics queries* for querying time series databases independent of the query language they use natively and ii) *composed metrics* for creating higher-level metrics from aggregated and composed lower-level metrics obtained through raw metrics queries. Since our API is based on objects, rather than on a textual language, it also comes with type safety features. When using PromQL or Flux directly, developers often need to write queries as plain strings in their application code, thereby breaking the type safety of that code. Fig. 4 shows a class diagram with a simplified view of our strongly typed metrics API, whose raw metrics query abstractions were inspired by PromQL, with some influences from Flux, and MQL.

For raw metrics queries, the central model type is `Time-Series`, which describes a sequence of sampled values for a metric. In addition to the `metricName`, a `TimeSeries` has a map of `labels` that can be used to further describe its samples, e.g., a metric named `http_requests_per_sec` could have a label `service`, which identifies the particular service from which this metric was observed.

The base interface for querying time series is `TimeSeries-Query`. Like a relational DB query results in a set of one or more rows, a time series DB query results in a set of one or more time series, each with a distinct metric name and labels combination. A query for `http_requests_per_sec` could result in two distinct `TimeSeries`, one with the label `service = 'gentics_mesh'` and one with the label `service = 'elasticsearch'`. This is why the execution of a `TimeSeriesQuery` results in a `QueryResult`, which can contain multiple `TimeSeries` instances.
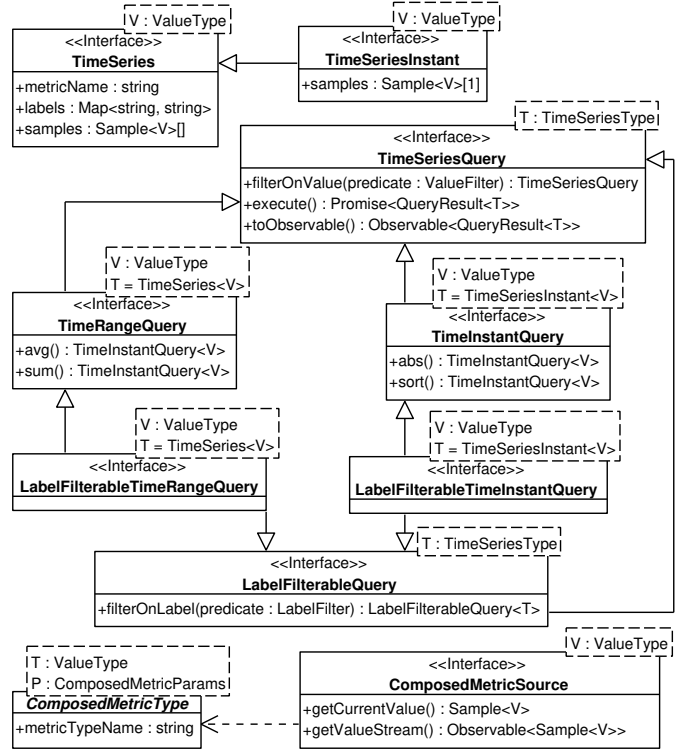

Fig. 4: Strongly Typed Metrics API (simplified view).

A time series DB not only allows retrieving a time series with particular properties, but also allows applying functions to the data, such as various types of aggregations or sorting. Certain functions, such as aggregations, require time series with multiple samples as input, while other functions, e.g., sorting, only work on time series with a single sample. For example, one may first query all time series for `http_requests_per_sec`, then compute the sum for each single time series, and finally sort the results to see which service gets the most requests. Prometheus will return an error when trying to sort time series with multiple samples, but it requires a developer to try to execute the query first.

The `TimeSeriesInstant` model type represents time series that are limited to a single sample. To support both time series types, the `TimeSeriesQuery` interface is extended by multiple subinterfaces: `TimeRangeQuery` for queries that result in a set of `TimeSeries` and `TimeInstantQuery` for queries that result in a set of `TimeSeriesInstants`. Each of these interfaces exposes only methods for DB functions that are applicable to the respective time series type. A function may also change the time series type, e.g., `sum()` is applied to a `TimeSeries`, but it returns a `TimeSeriesInstant`.

`LabelFilterableQuery` is another subinterface of `Time-SeriesQuery` that allows applying filters on the labels. Since our metrics query API needs to produce valid DB-specific queries, label filtering is a capability of a query that is lost after applying the first DB function, e.g., `sum()`, due to the structure of PromQL queries.

A composed metric is designated by a `ComposedMetric-Type`. It defines the name of the composed metric, the data

type used for its values, and which parameters are needed to obtain it (e.g., the name of the target workload). The metric values are supplied by a `ComposedMetricSource`, which may use raw metrics queries internally to obtain and aggregate multiple lower-level metrics, which are composed to form the higher-level composed metric.

For each `ComposedMetricType` there may be multiple `ComposedMetricSources`. This allows decoupling the type of a composed metric from the implementation that computes it and enables multiple implementations, which can be tailored to various types of workloads, such as REST APIs or databases, while delivering the same type of composed metric.

D. SLO Script Object Model

The SLO Script object model, a subset of which is shown in Fig. 5, is an instantiation of the language's meta-model in the framework. This abstract object model allows SLO Script to achieve orchestrator independence and promotes extensibility.

Every object that is submitted to the orchestrator must be of type `ApiObject` or a subclass of it. It contains an `objectKind` attribute that describes its type. The `ObjectKind.group` attribute denotes the API group of the type, which can be seen as a package in UML. The `version` attribute identifies the version of the API group and `kind` conveys the name of the type. `ApiObject` also has a `metadata` attribute, which contains additional information about the object, including the name of the instance. The `spec` attribute contains the actual "payload" content of the object. `ObjectReference` extends `ObjectKind` with a `name` attribute to be able to reference existing object instances in the orchestrator. This is needed to refer to the target workload of an SLO in an `SloTarget`, which derives from `ObjectReference`.

`ApiObject` is the root extension point for objects that need to be stored in the orchestrator. For example, to instantiate the `SloMapping` construct from the meta-model, a TypeScript class needs to be created that inherits from `SloMappingBase`, which derives from `ApiObject`. It contains the type information for the spec and sets up the correct `ObjectKind` for this `SloMapping`. The `SloConfiguration` construct can be represented by an arbitrary TypeScript interface or class, but needs to be wrapped in a class implementing `SloMapping-Spec`, which will store the configuration. A concrete example will be shown in Section V. The `SloMapping` represents a custom resource type that needs to be registered with the orchestrator. As part of future work, we will create a build system capable of automatically generating definitions for these resources.

To identify an `ElasticityStrategy` when configuring an `SloMapping`, the `ObjectKind` subclass `Elasticity-StrategyKind` is used. For each `ElasticityStrategy`, an `ElasticityStrategyKind` subclass has to be created and parameterized with the `SloOutput` and `SloTarget` types expected by the `ElasticityStrategy`. This in conjunction with the `SloOutput` type configured on a `ServiceLevel-Objective` and its corresponding `SloMapping`, enables the type checking discussed in the previous sections.
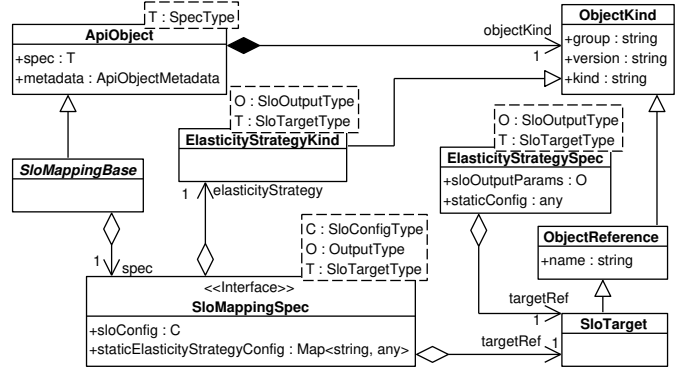

Fig. 5: Core SLO Script Object Model Types (partial view).

The `SloOutput` meta-model construct is instantiated by creating an arbitrary TypeScript class. To allow compatibility between as many SLOs and elasticity strategies as possible, generic `SloOutput` data types, which are supported by multiple `ServiceLevelObjectives` and `Elasticity-Strategies` are recommended. The `SloCompliance` class provided by the core object model conforms to this requirement. It expresses the current state of the SLO as a percentage of conformance. A value of 100% indicates that the SLO is precisely met. A higher value indicates that the SLO is violated and that additional resources, e.g., scaling out, are needed, whereas a value below 100% indicates that the SLO is being outperformed, i.e., a reduction of resources, e.g., scaling in, should be considered.

Every orchestrator has its own set of abstractions – thus, an independent framework must provide a mechanism for transforming objects between its own structure and the native structure of each supported orchestrator. To this end, SLO Script provides a transformation service that allows each orchestrator-specific connector library to register transformers for those object types that require transformation, while directly copying those objects that do not require any transformation. The transformation is not limited to the type of the root object, instead the appropriate transformer is applied to every nested object recursively.

The transformation service does not serialize to the data format required by the orchestrator (e.g., JSON or YAML). It transforms the instances of the orchestrator-independent SLO Script classes into plain JavaScript objects, which can be serialized by the orchestrator-specific connector library. The deserialization of objects received from the orchestrator is also left to the connector library. It needs to supply plain JavaScript objects to the transformation service, which transforms the objects and creates instances of SLO Script classes.

The SLO Script object model is heavily influenced by that of Kubernetes, but the two are not equal. For example, in Kubernetes there is no `objectKind` property on an object returned from the API. Instead, a Kubernetes API object contains an `apiVersion` and a `kind` property, with the former being a combination of the SLO Script `group` and `version` attributes of `ObjectKind` and the latter being equal to `ObjectKind.kind`.
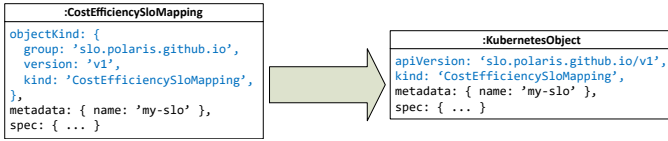
Fig. 6: Cost efficiency `SloMapping` before and after transformation.

To transform an SLO Script `ObjectKind` into its corresponding Kubernetes version, the SLO Script Kubernetes connector library registers a transformer for `ObjectKind`, which returns a plain JavaScript object with the `group` and `version` attributes combined into a single `apiVersion` attribute and a copy of the `kind` attribute. This alone is not enough because Kubernetes objects do not contain an `ObjectKind` property. Thus, the Polaris Kubernetes connector library also registers a transformer for the `ApiObject` class, which uses the transformation service to first transform the `ObjectKind` object and then embeds the contents of the result into a new object, which is going to become the final transformed `ApiObject`. The objects stored in the `metadata` and `spec` attributes are also first transformed and then stored in the result object. Fig. 6 shows an example of how a `CostEfficiencySloMapping` looks before and after being transformed to a Kubernetes object. The transformation techniques used for the SLO Script framework classes also apply to custom classes written by service providers for their SLOs.

The next Section will explore the runtime facilities, which are responsible for executing the defined SLOs.

IV. Runtime Mechanisms

Technically, the cluster component used for handling an SLO is a controller for the custom resource type defined by the `SloMapping` of that SLO. The controller watches the custom resource type instances in its deployment scope, creates and destroys SLO class instances accordingly, and evaluates them at a defined interval. Fig. 7 shows a UML activity diagram with the workflows within the controller.

To handle SLOs, the SLO Script runtime provides a control loop interface and a default implementation that maintains the set of active SLOs and evaluates them at a configurable interval. To add an SLO to the control loop, an `SloMapping`, which is received from the orchestrator, is needed, along with a key to uniquely identify that SLO. The key can be generated from the `metadata` of the `SloMapping` object. The `SloMapping` is used by the control loop to identify which SLO class to instantiate and to subsequently configure that instance, before adding it to its internal set.

The runtime aims to handle as many managerial tasks as possible to allow service providers to focus on their business logic. In the control loop in Fig. 7 only the actions highlighted in blue need to be implemented by the service provider.

The control loop is designed to work on all orchestrators. It needs to be configured with an `SloEvaluator`, which handles the execution of the SLO and the subsequent submission of its output to the orchestrator. Its `evaluateSlo(key, slo)` method, gets the SLO object's key and the object itself as parameters and has to asynchronously notify its caller when the
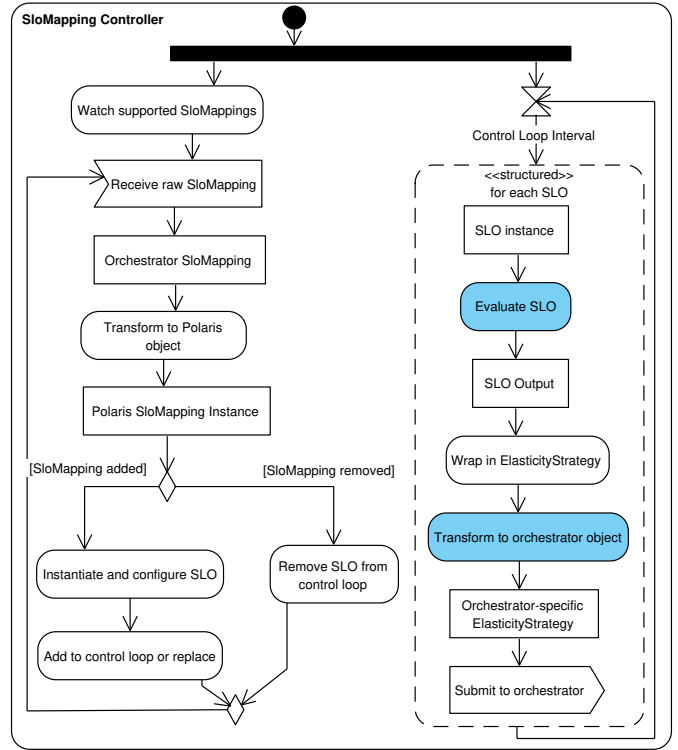

Fig. 7: `SloMappingController` workflow and SLOs lifecycle.

SLO evaluation is finished and the results have been submitted to the orchestrator. The runtime provides an abstract class to handle the evaluation of the SLO, as well as the wrapping of its result into the configured `ElasticityStrategy` object. It provides hooks for the orchestrator specific connector to execute code before and after the evaluation to apply the SLO's results to the orchestrator. The default implementation of the control loop gracefully handles errors during SLO evaluation, to ensure that a faulty implementation of one SLO does not prevent other SLOs in the same controller from being evaluated. Apart from an `SloEvaluator`, the SLO control loop must be configured with an `Observable` to define the evaluation interval – it must emit whenever the control loop should execute an iteration. This may be used to not only trigger a loop iteration at regular intervals, but can include other triggers as well, e.g., a "force evaluation now" event.

To be able to operate, the SLO control loop has to be integrated into a controller for the respective `SloMapping(s)`. This controller part depends heavily on the target orchestrator and should be implemented in the corresponding SLO Script orchestrator connector library. We currently provide a connector library for Kubernetes, which relies on kubernetes-client[11], the officially supported JavaScript client library for Kubernetes. Our controller implementation uses the `watch`[12] functionality of the Kubernetes API to be efficiently notified whenever a resource of an observed type is added, removed, or changed, such that the SLO control loop can be adjusted.

[11]https://github.com/kubernetes-client/javascript
[12]https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes

For the strongly typed metrics API we currently provide a connector for PromQL.

The controller uses the transformation service to convert between orchestrator-independent and orchestrator specific objects. To this end, the transformation service wraps the open-source library class-transformer[13], which provides most of the facilities needed for transformation. The registration of transformers for specific classes uses custom SLO Script mechanisms. Unlike class-transformer, it allows registering a transformer not just for a single property, but globally for all instances of a class and optionally its subclasses, and use that transformer on all transformable properties of that type.

Similar to class-transformer, SLO Script utilizes a Type-Script decorator to designate the type of a class property for transformation. This is necessary because the information about the types of class properties is not available at runtime, so it needs to be attached to the constructor function object as custom ECMAScript metadata. The need for metadata that is available at runtime is also the reason why most SLO Script framework types are classes instead of TypeScript interfaces – interfaces do not exist at runtime and thus, cannot be used for carrying metadata. SLO Script's `@PolarisType` decorator sets this type metadata for a property of a class and registers a helper with class-transformer, responsible for looking up the registered SLO Script transformer for that type or using the default transformer.

Registering a transformer is also possible for specific `ObjectKind` configurations. This is used, e.g., to automatically instantiate the correct class when an object of kind `slo.polaris-slo-cloud.github.io/v1/Cost-EfficiencySloMapping` needs to be transformed.

To facilitate adoption of SLO Script, our project includes a Command-Line Interface (CLI) tool that can be used to generate skeletons for `SloMapping` classes and SLO controllers, as well as build and deploy them. More information about the CLI can be found in a demo video online[14].

Further details on the runtime are available in [21]. The next Section will examine the realization of our motivating use case to evaluate SLO Script.

V. Evaluation & Implementation

We have implemented the core library of SLO Script, its CLI, as well as the controllers and connectors for Kubernetes and Prometheus using TypeScript and Go. The motivating use case is realized using the CLI and these libraries. More details and the source code can be found in our code repositories[15].

To evaluate SLO Script we use an approach based on the guidelines defined in [22]. We use the real-world cost efficiency use case from Section II-A to illustrate that SLO Script fulfills its requirements by improving code *reusability*, *flexibility*, *ease of use*, and *expressiveness* and reducing *susceptibility to errors* and thus, increases productivity.

---

[13]https://github.com/typestack/class-transformer
[14]https://www.youtube.com/watch?v=3_z2koGTExw
[15]https://polaris-slo-cloud.github.io

The goal of the cost efficiency SLO is to trigger an elasticity strategy whenever the current cost efficiency deviates too far from the defined target value. Since SLO Script allows the use of an arbitrary elasticity strategy, as long its input parameter type matches the SLO's output, we will use the term *increase resources* to refer to any sequence of elasticity actions that enlarge the resources allocated to a service, e.g., scaling up or scaling out, and *decrease resources* to any sequence of elasticity actions that reduce the resources allocated to a service, such as scaling down or scaling in.

Contrary to a simple CPU utilization SLO, it is not possible to derive whether an increase or a decrease in resources is needed by examining only the current and target values of the cost efficiency. For example, a low cost efficiency is ambiguous – it may indicate that either

- the system cannot handle the current high demand in time and that an increase in resources is needed or
- all requests are handled in time, but too many resources are provisioned compared to the few incoming requests, such that a decrease in resources is needed.

An *expressive* language is needed to distinguish these two cases. In SLO Script, we define the type `CostEfficiency-SloConfig` as shown in Listing 1. To handle the ambiguity problem we just described, we add an additional parameter to this configuration type: the minimum percentile of requests that should be handled within the time threshold. If the number of requests per second faster than the threshold is below that percentile, the service does not have enough resources to handle the load, whereas if it is above that percentile, the service has too many resources.

```
export interface CostEfficiencySloConfig {
  responseTimeThresholdMs: number;
  targetCostEfficiency: number;
  minRequestsPercentile?: number; }
```

Listing 1: Cost efficiency SLO configuration.

Listing 2 shows the `SloMappingSpec` and `SloMapping` classes. The spec class defines in the generic parameters for its superclass that the configuration type for this SLO will be `CostEfficiencySloConfig`, the output type will be `SloCompliance`, and the target workload must be of type `RestServiceTarget`. This short definition ensures that i) the SLO can only be applied to workloads of the correct type, i.e., workloads that expose the required metrics, and that ii) only an elasticity strategy that supports the SLO's output data can be used, because each `ElasticityStrategyKind` needs to specify the compatible input types in an analogous way. This greatly reduces the possibility for deploy-time or runtime errors, because SLO Script enforces that only compatible workloads and elasticity strategies are used.

The constructor of the `CostEfficiencySloMapping` class initializes the `objectKind` property to ensure that the correct API group and kind are configured and uses the `@Polaris-Type` decorator to set the appropriate class for the transformation of the `spec` property. At the moment, the Kubernetes Custom Resource Definition (CRD) for registering the SLO

mapping type with the orchestrator must either be written manually or be generated from an equivalent data structure written in Go – this will be addressed in a future version of the CLI, which will support automatic generation of CRDs.

```
export class CostEfficiencySloMappingSpec extends
    SloMappingSpecBase<CostEfficiencySloConfig,
    SloCompliance, RestServiceTarget> { }
export class CostEfficiencySloMapping extends
    SloMappingBase<CostEfficiencySloMappingSpec> {
  constructor(initData?:
      SloMappingInitData<CostEfficiencySloMapping>){
    super(initData);
    this.objectKind = new ObjectKind({
      group: 'slo.polaris-slo-cloud.github.io',
      version: 'v1',
      kind: 'CostEfficiencySloMapping' });
    initSelf(this, initData);
  }
  @PolarisType(() => CostEfficiencySloMappingSpec)
  spec: CostEfficiencySloMappingSpec; }
```

Listing 2: Cost efficiency SLO mapping.

The `CostEfficiencySlo` class implements the actual SLO. It uses the `MetricsSource` to retrieve the metrics for the target workload and uses them in conjunction with the configuration to compute an `SloCompliance` that indicates if the resources need to be increased or reduced.

To apply the SLO to a workload, service consumers need to instantiate the `SloMapping` as shown in Listing 3. Any TypeScript compatible IDE can provide code completion for the required properties, which greatly benefits the *ease of use*, and give immediate feedback if the chosen target workload or elasticity strategy are not compatible with the SLO, thus, *revealing errors at the time of writing*, which would have been discovered only at deploy-time or even at runtime, if plain JSON or YAML had been used for configuration.

```
export default new CostEfficiencySloMapping({
  metadata: new ApiObjectMetadata({ name:
      'data-service-cost-efficiency' }),
  spec: new CostEfficiencySloMappingSpec({
    targetRef: new RestServiceTarget({
      group: 'apps',
      version: 'v1',
      kind: 'Deployment',
      name: 'data-service' }),
    elasticityStrategy:
      new HorizontalElasticityStrategyKind(),
    sloConfig: {
      responseTimeThresholdMs: 400,
      targetCostEfficiency: 1000,
      minRequestsPercentile: 90 } }) });
```

Listing 3: Applying the cost efficiency SLO to a workload.

Since TypeScript is a superset of JavaScript, a developer can circumvent the type checking of SLO Script by writing plain JavaScript. The type safety can also be evaded by applying plain JSON or YAML configuration to the orchestrator. However, this is not an issue, because our aim is not to lock someone into a type safety system that cannot be circumvented in any way. The goal is to provide a language, consisting of domain-specific abstractions and restrictions, which, if used, increase productivity and provide type safety.

TABLE I: Lines of Code (excl. comments and blanks).

Component	Lines of Code	Generated	% of Total
SLO Mapping Type	53	50	2%
SLO Controller	224	99	8%
Runtime	2616	–	90%
Total	2893	149	100%

From the architectural perspective, the biggest benefit of SLO Script is the decoupling of SLOs from elasticity strategies, which increases code *reusability* and *flexibility*. The clear separation of SLO implementations from elasticity strategy implementations allows them to be reused in multiple combinations as long as their output/input types match. The input/output types can be seen like interfaces in object-oriented programming. If an elasticity strategy implements the interface required by the SLO, the two may be used in conjunction. This brings the flexibility of object-oriented programming to the management of SLOs in the cloud.

The SLO Script runtime eases the development of SLOs, because it lets service providers focus on their data types and business logic. The runtime's SLO control loop handles the integration with the orchestrator, as well as the management of the active SLOs. Out of the steps in the control loop, depicted in Fig. 7, only the "Evaluate SLO" step needed to be implemented for our cost efficiency use case. This is evident from Table I, which shows the line counts of the various components of our cost efficiency implementation. The SLO Script runtime makes up 90% of the total code. The `CostEfficiencySloMapping` class and its supporting types add up to 53 lines, however, 50 of these were generated by the CLI. The SLO controller and all its metrics queries take up 224 lines, 99 of which were generated.

The orchestrator independent object model of SLO Script eases the porting of SLOs and their mappings to other orchestration platforms, promoting flexibility, limiting the possibility for vendor lock-in for consumers, and fostering open source collaboration on SLOs for multiple platforms. Many SLOs may be implemented in a completely orchestrator-independent manner as well, allowing the creation of "standard SLO libraries" for instant reuse on other platforms.

VI. RELATED WORK

With elasticity being a flagship property of cloud computing, there has been a lot research in this area. We now explore some related work in the field of elasticity and SLOs.

All big commercial cloud providers support automated elasticity of some sort. However, the vast majority only provides simple SLOs that use a lower and upper bound or an average threshold for a metric that is directly measurable on the system. Some requirements can also be expressed in more high-level terms, e.g., AWS allows specifying the targeted availability of a service [23] or the durability of a DB in "nines" (e.g., "four nines" meaning 99.99% availability). Nevertheless, availability and durability are only simple SLOs that address a single elasticity dimension and "nines" cannot be considered a business metric. Custom metrics can be provided at some cloud providers through a query language.

However, specifying the metrics query in the SLO configuration reduces maintainability, e.g., our PromQL query to calculate cost efficiency was complex and specific to the components we used and would, thus, be cumbersome to maintain. Furthermore, each provider, such as AWS [18], Azure [19], and Google Cloud [20], uses its own mechanism for configuring the autoscaler, i.e., it is often not possible to write one configuration that works for all providers, fostering vendor lock-in The supported elasticity strategies are mostly horizontal and vertical scaling, with some exceptions, such as an elasticity strategy for the AWS DynamoDB [24] that allows increasing read and write capacities independently.

For Kubernetes, there are multiple autoscalers available – the most prominent being Horizontal Pod Autoscaler (HPA) [10], Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA) [25]. As already discussed in the motivation section, HPA allows a workload to scale out/in, but only supports simple SLOs, with complex custom metrics requiring an adapter API server to provide the custom metrics. Vertical Pod Autoscaler (VPA) can be used to scale up/down, albeit currently not in conjunction with HPA[16]. It allows configuration of the vertical elasticity strategy, but not of the SLO – the decision when to scale is taken automatically based on the current resource usage. The limits defined for the pod are respected though. Cluster Autoscaler (CA) does not scale single workloads, but the entire cluster by adding and removing nodes as needed. Its SLO is the time that is allowed to pass after a pod can no longer be scheduled on the cluster due to a lack of resources until CA resizes the cluster[17]. HPA, VPA, and CA all tie their SLOs tightly to the elasticity strategies. There is research that improves on VPA [26] and CA [27]. However, they focus on improving the performance of the elasticity strategy and the final result, but not on the possibilities for defining SLOs or decoupling the elasticity strategy from the SLO.

SLO-ML [28] is a language that allows service consumers to define SLOs in order for the language runtime to choose appropriate cloud services and SLAs for the deployment. While facilitating the initial deployment of a workload, it does not provide support for runtime elasticity.

OpenSLO[18] is a specification, in an early stage, that should allow the definition of and interaction with SLOs. However, it currently foresees the specification of metrics queries directly inside the definition/configuration of an SLO, which is hard to maintain for SLOs that depend on complex metrics and which should be reused many times.

Wang et al. [29] combine horizontal and vertical scaling to achieve an availability SLO and reduce costs. The SLO is however, limited to availability. It is achieved through horizontal scaling, while vertical scaling is utilized to reduce costs if the SLO is fulfilled. This approach provides a complex

elasticity strategy, but it falls short of supporting complex SLOs and multiple decoupled elasticity strategies.

There are some languages that allow defining SLOs using custom metrics and triggering elasticity strategies. For example, SYBL [30] is a language and runtime that allow defining complex constraints, i.e., SLOs, on cloud applications and their components. It supports the definition of custom metrics and can be extended e.g., with additional elasticity strategies. However, its implementation is tightly coupled to OpenStack and it lacks a plugin system, which would allow extensions without recompiling the entire runtime. rSLA [31] is an SLA definition language with runtime facilities that allows the definition of SLOs using raw and custom metrics and supports triggering arbitrary actions, e.g., scaling, upon SLO violations. Both, SYBL and rSLA support SLOs, custom metrics, and decouple them from elasticity strategies. However, they do not pass parameters that result from the SLO evaluation to the elasticity strategies, thus discarding possibly important information and allowing only generic actions, instead of parametrized elasticity strategies.

VII. Conclusion & Future Work

This paper has presented SLO Script, a language and accompanying framework for defining and implementing Service Level Objectives, based on TypeScript and being part of the open source Polaris project. We have motivated why SLO Script is needed using a real-world use case and enumerated the requirements for its design. We showed the language's meta-model and then described SLO Script's design and how it fulfills our main contributions of 1) high-level StronglyTypedSLO abstractions with type safety features, 2) decoupling of SLOs from elasticity strategies, 3) a strongly typed metrics API, and 4) an orchestrator-independent object model that promotes extensibility. Next, we explained how SLO Script's runtime mechanisms and the SLO control loop work. For evaluating our language and framework, we illustrated how to implement and configure the cost efficiency SLO for the motivating use case and highlighted the benefits of using SLO Script and its CLI for this purpose.

We identify several steps of future work with the goal of realizing the Polaris framework described in [3]. SLO Script will receive the ability to create elasticity strategies. The CLI will be extended with automatic generation of Kubernetes CRDs for SLO mapping types, as well as serialization and deployment of SLO mapping instances. A web user interface will allow service consumers to visualize the relationships between metrics, SLOs, SLO mappings, and elasticity strategies. Furthermore, the Polaris project will expand to bring SLOs to edge computing environments as well. The directions include: scheduling on edge resources, as well as executing elastic controls and governing SLO eligible resources under uncertainty [32].

References

[1] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level metrics to high level slas - lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments,"

[16]https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#known-limitations

[17]https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-are-the-service-level-objectives-for-cluster-autoscaler

[18]https://github.com/OpenSLO/OpenSLO

in *2010 International Conference on High Performance Computing & Simulation*. IEEE, 28.06.2010 - 02.07.2010, pp. 48–54.

[2] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[3] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, pp. 39–50, 2020.

[4] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, 2013, pp. 23–27.

[5] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of elastic processes," *Internet Computing, IEEE*, vol. 15, no. 5, pp. 66–71, 2011.

[6] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a catalogue of metrics for evaluating commercial cloud services," in *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, 20.09.2012 - 23.09.2012, pp. 164–173.

[7] T. A. Hjeltnes and B. Hansson, "Cost effectiveness and cost efficiency in e-learning," *QUIS-Quality, Interoperability and Standards in e-learning, Norway*, 2005.

[8] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, 2018.

[9] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE, 52019, pp. 1–6.

[10] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors (Basel, Switzerland)*, vol. 20, no. 16, 2020.

[11] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "The limit of horizontal scaling in public clouds," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 5, no. 1, 2020.

[12] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in cloud computing: a survey," *annals of telecommunications - annales des télécommunications*, vol. 70, no. 7-8, pp. 289–309, 2015.

[13] A. Ullah, J. Li, Y. Shen, and A. Hussain, "A control theoretical view of cloud elasticity: taxonomy, survey and challenges," *Cluster Computing*, vol. 21, no. 4, pp. 1735–1764, 2018.

[14] The Kubernetes Authors, "Custom metrics api - design proposal," 2018-01-22. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/custom-metrics-api.md

[15] ——, "External metrics api - design proposal," 2018-12-14. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/external-metrics-api.md

[16] ——, "Hpa v2 api extension proposal," 2018-02-14. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/hpa-external-metrics.md

[17] ——, "Horizontal pod autoscaler with arbitrary metrics - design proposal," 2018-11-19. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/hpa-v2.md

[18] Amazon Web Services, Inc., "Aws auto scaling features," 2020. [Online]. Available: https://aws.amazon.com/autoscaling/features/

[19] Microsoft, "Autoscaling," 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

[20] Google, LLC, "Autoscaling groups of instances," 2020. [Online]. Available: https://cloud.google.com/compute/docs/autoscaler

[21] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "A novel middleware for efficiently implementing complex cloud-native slos," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[22] P. Mohagheghi and Ø. Haugen, "Evaluating domain-specific modelling solutions," in *Advances in Conceptual Modeling – Applications and Challenges*, ser. Lecture Notes in Computer Science, J. Trujillo, G. Dobbie, H. Kangassalo, S. Hartmann, M. Kirchberg, M. Rossi, I. Reinhartz-Berger, E. Zimányi, and F. Frasincar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6413, pp. 212–221.

[23] Amazon Web Services, Inc., "5 9s (99.999%) or higher scenario with a recovery time under 1 minute," 2020. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/s-99.999-or-higher-scenario-with-a-recovery-time-under-1-minute.html

[24] ——, "Managing throughput capacity automatically with dynamodb auto scaling," 2020. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html

[25] The Kubernetes Authors, "Autoscaling components for kubernetes," 2020. [Online]. Available: https://github.com/kubernetes/autoscaler

[26] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 08.07.2019 - 13.07.2019, pp. 33–40.

[27] M. Wang, D. Zhang, and B. Wu, "A cluster autoscaler based on multiple node types in kubernetes," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 12.06.2020 - 14.06.2020, pp. 575–579.

[28] A. Elhabbash, A. Jumagaliyev, G. S. Blair, and Y. Elkhatib, "Slo-ml: A language for service level objective modelling in multi-cloud applications," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, K. Johnson, J. Spillner, D. Klusáček, and A. Anjum, Eds. New York, NY, USA: ACM, 12022019, pp. 241–250.

[29] W. Wang, H. Chen, and X. Chen, "An availability-aware virtual machine placement approach for dynamic scaling of cloud applications," in *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, 2012, pp. 509–516.

[30] G. Copil, D. Moldovan, H. Truong, and S. Dustdar, "Sybl: An extensible language for controlling elasticity in cloud applications," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 112–119.

[31] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig, "rsla: A service level agreement language for cloud services," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 27.06.2016 - 02.07.2016, pp. 415–422.

[32] S. Nastic, G. Copil, H.-L. Truong, and S. Dustdar, "Governing elastic iot cloud systems under uncertainty," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 131–138.