A Novel Middleware for Efficiently Implementing Complex Cloud-Native 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) guide the elasticity of cloud applications, e.g., by deciding when and how much the resources provisioned to an application should be changed. Evaluating SLOs requires metrics, which can be directly measured on the application or system, or, more elaborately, be composed from multiple low-level metrics. The implementation of such metrics and SLOs, the triggering of elasticity strategies, and allowing configurability by the user deploying an application, requires a flexible middleware. In this paper, we present a middleware that provides an orchestrator-independent SLO controller for periodically evaluating SLOs and triggering elasticity strategies, while decoupling SLOs from the elasticity strategies to increase flexibility, and provider-independent services for obtaining low-level metrics and composing them into higher-level metrics. We evaluate our middleware by implementing a motivating use case, featuring a cost efficiency SLO for an application deployed on Kubernetes.**

***Index Terms*—Cloud, Service Level Objectives, Elasticity, Orchestrator-independent, Middleware**

I. Introduction

A *Service Level Objective (SLO)* is a "commitment to maintain a particular state of the service in a given period" [1]. As such, SLOs are fundamental parts of Service Level Agreements (SLAs), which, in cloud computing, are agreements between cloud providers and cloud consumers to define limits, within which the rented services need to operate [2]. SLOs need to be measurable, based on one or more system or application metrics, e.g., CPU usage or the application's response time, i.e., the time it takes the application to handle and send a response to a network request to its API.

When an SLO is violated, e.g., the application's response time exceeds an upper threshold due to high demand, the cloud can provision more resources for the application, allowing it to meet the response time SLO again. Once the demand reduces and the response time goes below a lower threshold, the cloud can deprovision resources to reduce costs. This autonomous adaptability to the current demand is called *elasticity* [3], which may apply not only to resources, but also to costs, and quality (e.g., accuracy of a weather prediction) [4]. We define an *elasticity strategy* as a sequence of actions that adjust the resources provisioned to a cloud application and/or adjust its configuration to correct an SLO violation.

To efficiently adjust the elasticity of a deployed cloud application, which we refer to as a *workload*, based on its SLOs, a Monitor Analyze Plan Execute (MAPE) loop [5] can be implemented: i) the *monitoring* of system and workload metrics can be handled by tools, such as Prometheus[1], ii) the *analysis* of the metrics to evaluate whether the defined goals are met, is the task of an SLO, iii) the *planning* of actions to correct a violated SLO needs to be done by the elasticity strategy, and iv) the *execution* of the planned actions is carried out by the cloud orchestrator, e.g., Kubernetes[2]. As part of the Polaris SLO Cloud (Polaris) project [6], we are working to bring complex SLOs, which can be combined with multiple elaborate elasticity strategies, to the cloud.

In this paper, we focus on the realization of SLOs, i.e., the *analysis* step of the control loop. In the analysis step, an SLO must obtain one or more metrics from the monitoring step and pass its evaluation result on to the planning step, i.e., an elasticity strategy – this is not trivial. A common approach is to implement the SLO as a control loop itself. The variety of monitoring solutions and databases (DBs) makes obtaining metrics difficult without tying the implementation to a particular vendor. Once the metrics have been obtained, they may need to be aggregated to gain deeper insights. When the current status of the SLO has been determined, the outcome needs to be conveyed to an elasticity strategy; ideally multiple elasticity strategies should be supported.

To facilitate the implementation of complex SLOs, we present the Polaris Middleware. Its implementation is published as open source, as part of the Polaris project[3]. Our main contributions with the Polaris middleware include:

1) An *orchestrator-independent SLO controller* periodically evaluates SLOs and triggers elasticity strategies, while ensuring that SLOs and elasticity strategies remain de-

This work is supported by Futurewei's Cloud Lab. as part of the overall open source initiative.

[1]https://prometheus.io
[2]https://kubernetes.io
[3]https://polaris-slo-cloud.github.io

2159-6190/21/$31.00 ©2021 IEEE
DOI 10.1109/CLOUD53861.2021.00055

coupled to increase the number of possible SLO/elasticity strategy combinations.
2) A *provider-independent SLO metrics collection and processing mechanism* allows querying raw time series metrics, as well as, composing multiple metrics into reusable higher-level metrics.
3) A *Command-Line Interface (CLI) Tool* creates and manages projects that rely on the Polaris middleware.

Additionally, we provide platform connectors for Kubernetes, which has been found to have the most capabilities for production-level services among commonly used container orchestrators [7], and Prometheus, which is a popular choice for a time series DB.

The remainder of this paper is structured as follows: Section II provides further motivation for our work using a real-world use case, Section III presents a high-level overview of the Polaris middleware, Section IV describes the central mechanisms, and Section V their implementation. In Section VI we evaluate the Polaris middleware by implementing the real-world use case, in Section VII we present related work, and in Section VIII we conclude this paper.

II. Motivation

One of the aims of the Polaris project is to make SLOs first class entities in cloud computing and facilitate their realization through its runtime. Polaris is part of Linux Foundation's Centaurus project[4], a novel open-source platform targeted towards building unified and highly scalable public or private distributed cloud infrastructure and edge systems.

A. Illustrative Scenario

To illustrate the need for the Polaris middleware, we present a real-world cloud use case, featuring a headless Content Management System (CMS) that can be provisioned in the form of Software-as-a-Service (SaaS). A headless CMS is primarily used through its REST API, as a content source integrated into another application. Gentics Mesh[5] is an open source headless CMS, consisting of two main components: the CMS itself and an ElasticSearch[6] DB. Each component exposes multiple low-level metrics, e.g., CPU usage, or network throughput. However, when deploying Gentics Mesh as SaaS, customers expect the cloud provider to offer SLOs that automatically scale the entire workload, i.e., the CMS and the DB. Such an SLO may be based on the average CPU usage of all workload components or the network throughput of the REST API. Mapping their business goals to such low-level SLOs, while keeping their budget, is difficult for most customers. Thus, a high-level SLO that combines performance and costs can allow customers to better define their objectives. One such high-level SLO is *cost efficiency*, which is often defined as the number of requests per second served faster than N milliseconds divided by the total cost of the workload [8], [9].

Cost efficiency is a high-level metric that is not directly observable on the workload. Instead, it needs to be calculated by combining multiple low-level metrics. Doing this without tying the implementation to a specific time series DB is difficult, because each major time series DB has its own query language, e.g., Prometheus uses PromQL, InfluxDB[7] uses Flux, and Google Cloud Platform uses MQL[8]. To alleviate this problem, the Polaris middleware offers a DB-independent service for querying raw metrics. Once a high-level metric, e.g., the total cost of a workload, has been computed, it would be beneficial to reuse it in multiple SLOs, thus, we also provide a service for obtaining such high-level, composed metrics.

Before reading and evaluating metrics, an SLO needs to be configured by the customer and executed periodically to perform its evaluation. Once the SLO detects a violation, it has to be able to trigger an elasticity strategy to bring the workload back into a state, where the SLO is respected. Horizontal scaling is the most commonly used elasticity strategy today [10]. Nevertheless a customer should be able to choose from different elasticity strategies to trigger upon an SLO violation – yet, most SLOs today are tightly coupled with one specific elasticity strategy, e.g., the average CPU usage SLO in the Kubernetes Horizontal Pod Autoscaler (HPA) [11]. To support the aforementioned flexibility, the runtime's SLO control mechanism, which should be generic enough to be shared among all SLOs, must provide these features.

B. Research Challenges

RC-1 *Decoupling of SLOs from elasticity strategies*: Many SLOs are tightly coupled with the elasticity strategy they trigger. For example, HPA in Kubernetes provides an average CPU usage SLO, which can trigger only horizontal scaling. This rigid coupling reduces the flexibility of a system – re-implementing every useful elasticity strategy for every SLO controller is infeasible. Thus, a decoupling of SLOs from elasticity strategies is needed.

RC-2 *Enable realization of high-level SLOs, based on complex metrics*: Most metrics that guide cloud elasticity today are directly measurable at the system or application level [10], [12], [13]. While HPA supports custom metrics using the custom and external metrics APIs[9], both approaches require developers to write a custom API server, to which the Kubernetes API can proxy requests, thus, increasing development and maintenance effort. The external metrics API supports the specification of custom queries, but this feature must also be implemented by the custom API server. An SLO middleware must provide mechanisms for combining multiple low-level metrics into high-level metrics that are reusable in multiple SLOs.

RC-3 *Cloud platform and datastore independence*: The configuration of autoscaling solutions is commonly specific to a cloud vendor or orchestrator. Likewise, there is a distinct

[4]https://www.centauruscloud.io
[5]https://getmesh.io
[6]https://www.elastic.co/elasticsearch/
[7]https://www.influxdata.com
[8]https://cloud.google.com/monitoring/mql/reference
[9]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-metrics-apis

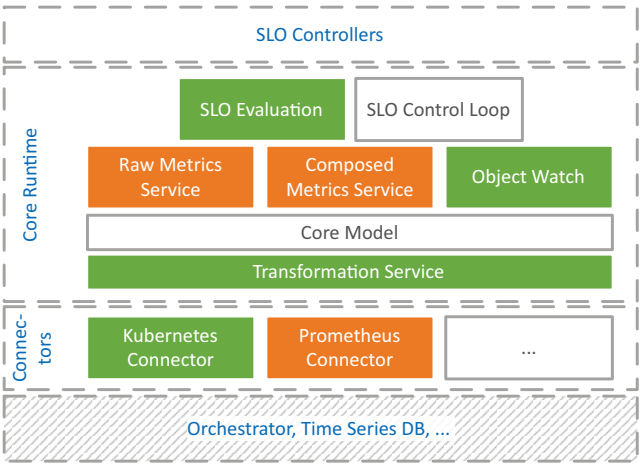

Fig. 1: Polaris Runtime Architecture (the colors indicate, which connector realizes interfaces from a particular component).

query language for each major time series DB. Portable SLOs require mechanisms to make them independent of particular vendors.

Our first contribution, the orchestrator-independent SLO controller, addresses all three research challenges, while our second contribution, the provider-independent SLO metrics collection and processing mechanism, focuses on RC-2 and RC-3. Our third contribution, the CLI Tool, is a supporting mechanism for leveraging the other two.

III. Framework Overview

In this Section we provide a high-level overview of the Polaris middleware's architecture and the Polaris CLI.

A. Architecture

The architecture of the Polaris middleware is divided into two major layers, as illustrated in Fig. 1. The *Core Runtime* layer contains orchestrator-independent abstractions and algorithms. The *Connectors* layer below it, contains orchestrator and DB-specific implementations of interfaces from the core runtime to allow connecting it to a specific orchestrator or time series DB, which are located underneath this layer. *SLO Controllers* are built on top of the core runtime, shielding them from orchestrator- and DB-specific APIs. We subsequently describe each of the runtime components in Fig. 1 briefly:

The *Core Model* contains abstractions for defining and implementing SLOs. The most important ones are `ServiceLevelObjective`, `SloTarget`, `SloMapping`, and `ElasticityStrategy`. `ServiceLevelObjective` defines the interface that the SLO implementation needs to realize to plug into the control loop provided by the runtime. `SloTarget` is an abstraction used identify the target workload that the SLO should be applied to. An SLO is configured through an `SloMapping`, which associates a particular SLO type with a target workload and an elasticity strategy, thus, establishing a loose coupling between them. SLO mappings are deployed to the orchestrator as custom resources. Each SLO mapping type entails the definition of a custom resource type in the orchestrator. The addition of a new SLO mapping resource instance, activates the respective SLO controller, which subsequently enforces the SLO. The `ElasticityStrategy` that is specified as part of an SLO mapping, identifies the strategy that should be used if the target violates the SLO, to bring it back into a state, where the SLO is adhered. Akin to an SLO mapping type, each `ElasticityStrategy` type is represented by a custom resource type in the orchestrator.

The *SLO control loop* is used in an SLO controller to watch the orchestrator for new or changed SLO mappings and to periodically evaluate the SLO. It relies solely on Polaris middleware abstractions and does not need be customized by an orchestrator connector or an SLO controller, albeit this is possible, if desired.

The *SLO Evaluation* facilities are used by the SLO control loop to perform the evaluation of the SLO and to trigger elasticity strategies on the orchestrator, if necessary. The evaluation of the SLO is handled by the core runtime, while the mechanisms for triggering an elasticity strategy, which are specific to each orchestrator, must be provided by the respective orchestrator connector.

The *Transformation Service* allows transforming orchestrator-independent Polaris middleware objects into orchestrator-specific objects for a particular target platform and vice-versa. The runtime provides a transformation mechanism that allows orchestrator connectors to register a type transformer for every object type that needs to be customized for the target orchestrator.

The *Object Watch* facilities allow observing a set of resource instances of a specific type in the orchestrator for additions, changes, and removals of instances. This is used, e.g., by the SLO control loop to monitor additions of or changes to SLO mappings. Orchestrator connectors must implement these facilities for their respective platforms.

The *Raw Metrics Service* enables DB-independent access to time series data to obtain metrics. A DB connector must transform the generic queries produced by this service into queries for its particular DB.

The *Composed Metrics Service* provides access to higher-level metrics, called *composed metrics*, which allow combining multiple lower level metrics into a reusable high-level metric. To make it accessible through the Composed Metrics Service, a composed metric may be packaged into a library that can be included in an SLO controller or it can be exposed as a service or stored in a shared DB, promoting loose coupling between the metrics providers and the SLO controllers. The implementation of the sharing mechanism may be provided by either the orchestrator or the DB connector.

The connectors create the bridge between the core runtime and the underlying orchestrator and DBs.

The *Kubernetes Connector* library provides Kubernetes-specific realizations of the three runtime facilities that are highlighted in green in Fig. 1. Kubernetes-specific transformers plug into the Transformation Service to enable the transformation of objects from the core model to Kubernetes-specific objects. The library also implements the object watch

facilities for the Kubernetes orchestrator, which allow the SLO control loop in an SLO controller to watch a particular SLO mapping Custom Resource Definition (CRD) for additions of new resource instances or changes to existing ones. The SLO evaluation realization for Kubernetes augments the generic evaluation facility from the core runtime by allowing it to trigger an elasticity strategy using Kubernetes CRD instances.

The *Prometheus Connector* implements the generic Raw Metrics Service using queries specific to the Prometheus time-series database. It also supplies a mechanism for reading composed metrics from Prometheus.

B. Polaris CLI

The Polaris Command-Line Interface (CLI) provides a mechanism for project creation, building, and deployment for developers, who want to use the Polaris middleware to create custom SLOs and controllers. Its aim is to provide a convenient user interface to developers, as well as a starting point for integrating Polaris middleware projects in Continuous Integration (CI) pipelines. The major commands are the following:

`polaris-cli generate <componentType> <name>` adds a component of the specified type to the project. The `componentType` may currently be one of three types:

- `mapping-type` creates a new SLO mapping type that can be used by consumers to apply and configure an SLO.
- `slo-controller` creates an SLO controller for an SLO mapping type and deployment configuration files.
- `mapping` creates a new mapping instance for an existing SLO mapping type. This is intended to be used by consumers, who want to configure and apply a particular SLO to their workload.

`polaris-cli (docker-)build <name>` executes the build process for the specified component to produce deployable artifacts. For an SLO mapping type, this is a library package that can be published for use by customers. For SLO controllers, a container image with the executable controller for deployment on the orchestrator is produced. For an SLO mapping, the output is a configuration file, representing an instance of the corresponding SLO type CRD.

`polaris-cli deploy <name> [destination]` deploys the build artifact of the specified component to the specified destination orchestrator.

The Polaris CLI provides a default implementation for all commands, but allows developers to override these defaults in the project file, enabling, for example, the use of a different tool for deployment of the artifacts.

IV. Mechanisms

In this Section, we describe the two main mechanisms provided by the Polaris middleware, i.e., the orchestrator-independent SLO controller and the provider-independent SLO metrics collection and processing mechanism.

A. Orchestrator-Independent SLO Controller

The central mechanism in an SLO controller is the SLO control loop – it monitors and enforces an SLO configured by a user. The SLO control loop itself is orchestrator-independent and merely requires some supporting services to be implemented by the orchestrator connector. The SLO controller can then use the control loop without further adaptation.

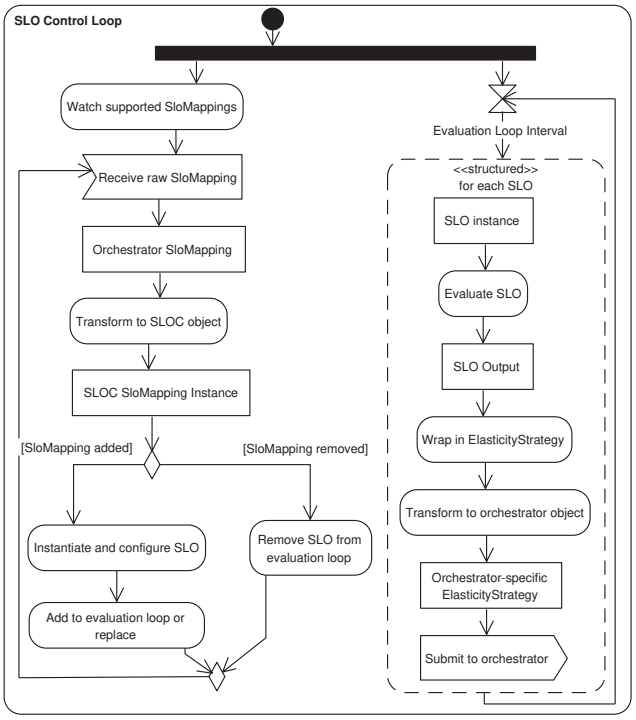

Fig. 2: SLO Control Loop.

The SLO control loop consists of two sub-loops, as shown in Fig. 2. The *watch loop* on the left side is concerned with observing additions, changes, or deletions of SLO mappings in the orchestrator using the object watch facilities and maintaining the list of SLOs managed by the control loop. The *evaluation loop* on the right side periodically evaluates each SLO and triggers the configured elasticity strategy using the SLO evaluation facilities.

1) Watch Loop: The watch loop begins by observing the SLO mapping custom resource types that the SLO controller supports. To this end, it uses the object watch facilities, which emit an event whenever an object of the watched types (i.e., the supported SLO mappings) is added, changed, or removed from the orchestrator. This functionality must be implemented by the orchestrator connector. Each watch event entails receiving the raw SLO mapping object that has been added, changed, or removed. Since this object is specific to the underlying orchestrator, it is transformed using the Transformation Service into an orchestrator-independent object.

The watch loop then acts according to the type of watch event. If a new SLO mapping has been added or changed, the appropriate SLO object that is capable of evaluating the SLO is instantiated, configured, and added to or replaced in the list of SLOs for the evaluation loop. If an existing SLO mapping has been removed, the corresponding SLO object is removed from the evaluation loop. Subsequently the watch loop goes

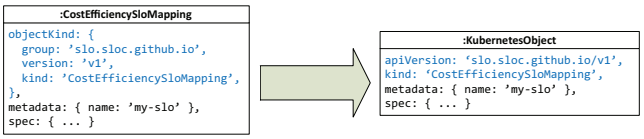

Fig. 3: Cost efficiency SLO mapping before and after transformation.

back to waiting for the next event.

2) Evaluation Loop: The evaluation loop executes at an interval that is configurable by the SLO controller. Whenever it is triggered, the evaluation loop iterates through the list of all its SLOs. For each SLO, the current status is evaluated using the SLO evaluation facilities. The exact evaluation process depends on the implementation of the particular SLO that is built on top of the Polaris middleware. Generally, SLO evaluation entails the retrieval of all relevant input metrics using the Raw Metrics Service and the Composed Metrics Service. These metrics may be further processed and combined and are subsequently compared to the ideal values configured by the user in the SLO mapping. This results in an *SLO output* that indicates if the SLO is currently fulfilled, violated, or outperformed (i.e., fulfilled by a large margin, such that e.g., a resource reduction is possible) and any additional information necessary to get it back into a fulfilled state, if necessary.

This output is wrapped in an elasticity strategy object of the type specified by the SLO configuration. The elasticity strategy object is subsequently transformed to an orchestrator-specific object using the Transformation Service and submitted to the orchestrator, where it will trigger the respective controller for the elasticity strategy. This submission to the orchestrator is the part of the SLO evaluation facilities that must be implemented by the orchestrator connector – the remainder of the SLO evaluation is orchestrator-independent.

The SLO control loop is designed to handle errors during the evaluation of an SLO gracefully, such that a problematic SLO does not cause the entire controller to fail.

The SLO control loop relies on the Transformation Service to convert between orchestrator-independent and orchestrator-specific objects. All objects that are received from or submitted to the orchestrator pass through this service. Orchestrator connector libraries can register transformers for object types, whose orchestrator-specific data structure does not match that of the corresponding type in the Polaris middleware. The Transformation Service is responsible for transforming the structure of objects, while serialization and deserialization (e.g., to/from JSON) are handled by the object watch and SLO evaluation facilities. To transform an object's structure, the Transformation Service recursively iterates through all attributes of an input object. If a transformer has been registered for an attribute's type, it is executed on the attribute's value according to the direction of the current transformation operation (i.e., from orchestrator-independent to orchestrator-specific or from orchestrator-specific to orchestrator-independent). If no transformer is registered for a particular type, the value is copied and the recursive iteration continues on the value's attributes. Fig. 3 exemplifies how an SLO mapping object for a cost efficiency SLO is transformed to a Kubernetes resource object (observe that the `objectKind` attribute of the Polaris object is transformed into two attributes, `apiVersion` and `kind`, on the Kuberenetes object).

Another essential mechanism in an SLO controller is the decoupling of SLOs and Elasticity Strategies. The goal of this is two-fold: i) allow an SLO to trigger a user-configurable elasticity strategy that is unknown at the time the SLO controller is built (i.e., the SLO controller cannot have a hardcoded set of elasticity strategy options) and ii) allow an elasticity strategy to be used by multiple SLOs to avoid having to reimplement the same set of elasticity strategies for every SLO.

To achieve both goals, we have defined a common structure for elasticity strategy resources, consisting of three parts: a reference to the target workload, the output data from the SLO evaluation, and static configuration parameters supplied by the user. The target workload reference and the static configuration parameters are copied from the SLO mapping by the Polaris middleware. The configuration parameters are specific to the elasticity strategy that the user has chosen, which does not limit the generality of the mechanism, because they are statically specified together with the identifier of the elasticity strategy that the user has chosen and are not modified by the SLO. Conversely, the SLO evaluation output data are entirely produced by the SLO controller. The structure of the SLO output determines which elasticity strategies can be combined with that SLO, i.e., if an elasticity strategy supports the SLO's output data type as input, the two are compatible. The Polaris middleware only needs to copy the SLO output data to the elasticity strategy resource.

Using a generic data structure that is supported by multiple SLOs and elasticity strategies as an SLO's output data type, increases the number of possible SLO/elasticity strategy combinations. Any suitable data structure can be used for this purpose. The Polaris middleware includes the generic `SloCompliance` data type, which captures the compliance to an SLO as a percentage: a compliance value of 100% indicates that the SLO is exactly met, a higher value means that the SLO is violated and that, e.g., an increase in resources is needed, while a lower value indicates that the SLO is being outperformed and that resources can be reduced to save costs. To avoid too frequent scaling, `SloCompliance` includes the possibility for specifying a tolerance value, within which no elasticity action should be performed.

B. Provider-Independent SLO Metrics Collection And Processing Mechanism

The metrics required for evaluating an SLO can be obtained through two mechanisms: i) the Raw Metrics Service and ii) the Composed Metrics Service. The former is intended for low-level metrics that are directly measurable on a workload, e.g., CPU usage or network throughput, while the latter allows obtaining higher-level metrics that are aggregations of several lower-level metrics or predictions of metrics.

1) Raw Metrics Service: The Raw Metrics Service enables the DB-independent construction of queries for time series data. To this end, it allows specifying the metric name and

the target workload, for which it should be obtained, as well as the time range and filter criteria. Furthermore, it provides arithmetic and logical operators and aggregation functions to operate on the metrics. Upon execution, a query is transformed into the native query language of the used time series DB. The result of a query is an ordered sequence or a set of ordered sequences of primarily simple (i.e., numeric or Boolean) raw or low-level metric values.

```
rawMetricsService.getTimeSeriesSource()
  .select('my_workload',
    'request_duration_seconds_count',
    TimeRange.fromDuration(
      Duration.fromMinutes(1)))
  .filterOnLabel(LabelFilters.regex(
    'http_controller', 'my_workload.*'))
  .sumByGroup(LabelGrouping.by('path'))
  .execute();
```

Listing 1: Raw Metrics Service query for total duration of all HTTP requests in the last minute, grouped by paths.

The Raw Metrics Service is designed as a fluent API [14], [15], which means that the code resulting from its use should be natural and easy to read. Specifically this entails chaining of method calls, supporting nested function calls, and relying on object scoping. Listing 1 shows a query for the sum of the durations of all HTTP requests that were made in the last minute, grouped by request paths.

2) Composed Metrics Service: The Composed Metrics Service is aimed at high-level metrics. These may be simple values (e.g., numbers or Booleans) or complex data structures. Unlike a raw (low-level) metric, a composed metric is not directly observable on a workload, but needs to be calculated, e.g., by aggregating several lower level metrics. A composed metric may also represent predictions of future values of a metric. Every composed metric has a *composed metric type* definition that specifies the data structure of its values and a unique name for identification.

The calculation of a composed metric requires an additional entity, termed a *composed metric source*, to perform this calculation. Each composed metric source supplies a metric of a specific composed metric type. A composed metric type is similar to an interface in object-oriented programming; it specifies the type of composed metric that is delivered and may be supplied by multiple composed metric sources.

Apart from its composed metric type, a composed metric source is also identified by the type of target workload it supports. This enables high-level metrics, such as cost efficiency, which need to be computed differently for various workload types. For example, for a REST service, cost efficiency relies on the response time of the incoming HTTP requests, a metric that is not available on a SQL database. There, the execution time of the queries could be used instead. This entails different composed metric sources, which can be registered to the respective workload types.

The Composed Metrics Service supports both, i) composed metric sources integrated into the SLO controller through libraries and ii) out-of-process composed metric sources that execute within their own metric controller. The former option computes the composed metric within the SLO controller and is simple to realize for developers, because it only requires the creation of a custom library that needs to be imported in the SLO controller and registered with the Composed Metrics Service. The latter option is more flexible and allows for decoupling of the implementation and maintenance of the SLO controller from that of the composed metric source.

Out-of-process composed metric sources may be implemented, e.g., as REST services or through the use of a shared DB. The latter allows the composed metric to be calculated once and reused by multiple SLO controllers. An out-of-process composed metric source can be leveraged to flexibly update or change the way a certain composed metric type is computed. For example, a `TotalCost` composed metric type is of interest to multiple SLOs. It may be supplied by a metric controller with a refresh rate of five minutes, i.e., the total cost of a workload is updated every five minutes. This metric controller can be replaced by a newer version, with a refresh rate of one minute, without having to recompile and redeploy the SLO controllers that depend on this composed metric.

V. Implementation

In this Section, we briefly describe the implementation of the mechanisms from Section IV in our core runtime and the connectors for Kubernetes and Prometheus.

The Polaris middleware and its CLI are realized in TypeScript and published as a set of npm library packages. An SLO controller is a Node.js application that uses these packages as dependencies to implement SLO checking and enforcement mechanisms. All middleware and CLI code, as well as example controllers, are available as open source[10].

A. Orchestrator-Independent SLO Controller

The orchestrator-independent SLO controller relies on the abstractions provided by the core model, as well as the object watch and SLO evaluation facilities. Fig. 4 shows the main components involved in the SLO control loop. In case the default control loop implementation does not suffice for a particular scenario, the runtime may be configured to use a custom implementation of the `SloControlLoop` interface.

The SLO control loop manages `ServiceLevelObjective` objects, which are implemented by the SLO controller. The `ObjectKindWatcher` is provided by the orchestrator connector library to enable observation of the supported SLO mapping types. The evaluation loop evaluates registered SLOs using the `SloEvaluator` provided by the orchestrator connector. The default implementation handles the evaluation of the SLO and the wrapping of its output in an elasticity strategy object – the connector library must only implement the submission to the orchestrator. The Kubernetes connector for the Polaris middleware relies on kubernetes-client[11], the officially supported JavaScript client library for Kubernetes. It is important to note that the decisions need to be made

[10] https://polaris-slo-cloud.github.io

[11] https://github.com/kubernetes-client/javascript

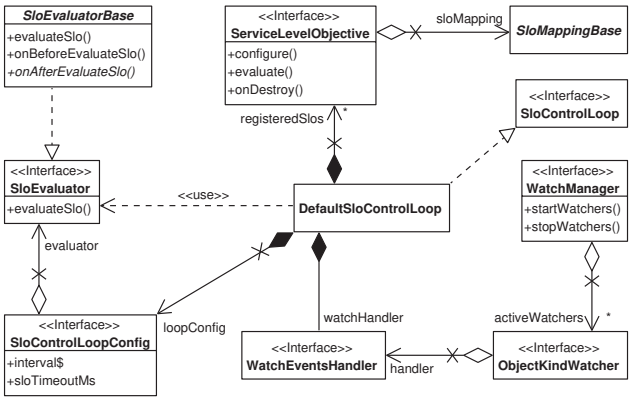

Fig. 4: SLO Control Loop components (simplified).

inside the SLO- and elasticity strategy-specific code in the respective controllers. The purpose of the Polaris middleware is to connect an SLO to any compatible elasticity strategy and to provide reusable facilities to reduce the effort of developing these types of controllers.

The Transformation Service is relies on the open-source library class-transformer[12] for executing the transformation process, but provides its own, more flexible, transformer registration mechanism. We assume that all raw orchestrator resources contain a metadata property that uniquely identifies their type. An orchestrator connector library is required to register a transformer that converts these metadata into an `ObjectKind` object, which is the Polaris abstraction used for identifying orchestrator resource types. The Transformation Service supports associating object kinds with Polaris classes to enable the transformation into the correct runtime objects.

The decoupling of SLOs and elasticity strategies relies on a common layout of elasticity strategy resources and the use of the same data type for the output of an SLO and the input of an elasticity strategy. The user selects an elasticity strategy for an SLO by specifying its object kind in the SLO mapping that configures the SLO. After evaluating an SLO, the Polaris middleware instantiates the elasticity strategy class associated with this object kind and copies the SLO output data to it. An SLO Mapping requires the user to choose exactly one elasticity strategy. An elasticity strategy is responsible for ensuring that its sub-actions do not conflict with each other, e.g., if it combines horizontal and vertical scaling. Unlike metrics, SLOs and elasticity strategies cannot be composed. However, it is possible to configure multiple SLOs for a single workload and, thus, also multiple elasticity strategies (one for each SLO). Since such combinations are highly use case specific, there is no generic conflict resolution mechanism. Instead, the user needs to ensure that there are no conflicts, which, however, does not limit the expressiveness of the solution.

B. Provider-Independent SLO Metrics Collection And Processing Mechanism

1) Raw Metrics Service: To create a raw metrics query, the Raw Metrics Service is used to obtain a `TimeSeriesSource`, which realizes a DB-independent interface for assembling time series queries for a particular target DB. The supported sources are registered with the Polaris middleware when the SLO controller starts. The `select()` method of `TimeSeriesSource` creates a new query by specifying the name of the metric and the target workload. Each method call on a query (see Listing 1) returns an immutable object that models the query up to this point. The query may be executed, using the `execute()` method, or extended by adding another query clause with an additional method call, which yields a new, immutable query object. This approach allows reusing a base query object, e.g., the time series of all HTTP request durations, for multiple queries without side effects, e.g., for the sum of all request durations and for the average duration of a request. When `execute()` is called on a query object q, the segments of the query chain, starting from the `select()` query object up to query object q, are passed to a `NativeQueryBuilder`. This builder needs to be provided by a DB connector library, e.g., the Prometheus connector.

2) Composed Metrics Service: To get a composed metric, a `ComposedMetricSource` is obtained from the Composed Metrics Service using a composed metric type and the target workload. Upon startup, the SLO controller registers all `ComposedMetricSource` realizations that are provided through libraries, together with their corresponding composed metric types and supported target workload types, in the Polaris middleware. These composed metric sources execute their metric computation logic inside the SLO controller, e.g., by using the Raw Metrics Service internally for retrieving and aggregating multiple raw metrics. If no `ComposedMetricSource` has been registered for a particular composed metric type, the Composed Metrics Service assumes that this is an out-of-process composed metric source, which is realized by a connector library. The Prometheus connector provides a `ComposedMetricSource` realization that relies on Prometheus as a shared DB, where standalone composed metric controllers store their computed metrics.

VI. Evaluation

We implement the motivating cost efficiency SLO use case from Section II to show the productivity benefits of using the Polaris middleware and run experiments in our cluster testbed to evaluate its performance.

A. Demo Application Setup

Fig. 5 provides an overview of the components of the cost efficiency SLO implementation and their relationships – blue components are implemented for the use case, white components are orchestrator and DB-independent parts of the Polaris middleware, and green and orange components are part of the Kubernetes and Prometheus connectors respectively. All code is available in the Polaris project's repository.

Our test cluster provides an elasticity strategy for horizontal scaling, which is also part of the Polaris project and accepts generic `SloCompliance` data as input.

[12]https://github.com/typestack/class-transformer

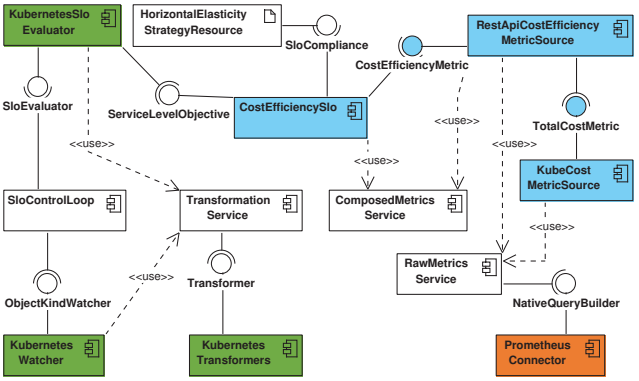

Fig. 5: Cost Efficiency SLO Implementation (blue), Kubernetes Connector (green), Prometheus Connector (orange).

First, we set up an SLO mapping type in an npm library to allow users to configure the cost efficiency SLO. To this end, the Polaris CLI can create a TypeScript class `CostEfficiencySloMapping`, which we extend with the configuration parameters. Currently, the YAML code for registering a Kubernetes CRD must be written manually or be generated from an equivalent data structure defined in Go – as part of future work, we will extend the Polaris CLI to support generating CRDs from the TypeScript SLO mapping classes.

To enable reuse of the cost efficiency metric, we implement it as a composed metric in a library. In our use case, cost efficiency is defined as the number of REST requests handled faster than N milliseconds, divided by the total cost of the workload. However, cost efficiency is not only useful for REST services, but can be applied to other types of services as well, e.g., a weather prediction service, albeit with a different raw metric as the numerator in the equation. To allow this, we define a generic cost efficiency composed metric type (composed metric types are shown as interfaces in Fig. 5) that can be implemented by multiple composed metric sources. Thus, we enable a cost efficiency SLO controller to support multiple workload types (e.g., REST services and prediction services) by either registering multiple cost efficiency composed metric sources from libraries or by relying on out-of-process composed metric services to provide the cost efficiency metric for the various workload types. In our use case we supply a cost efficiency implementation for REST services, but since the Composed Metrics Service differentiates between workload types when obtaining a composed metric source, this can be increased to an arbitrary number of implementations for various workload types.

Since total cost is an important metric in cloud computing, this part of the cost efficiency composed metric could be reused by other composed metrics or SLOs. To this end, we create a total cost composed metric type that may be supplied by multiple composed metric sources. We provide an implementation that relies on KubeCost[13], which we use to export the hourly resource costs to Prometheus. In the implementation of the `KubeCostMetricSource`, we use the Raw Metrics Service to obtain these costs and the recent CPU and memory usage of the involved workload components and multiply and sum them to obtain the total cost.

The `RestApiCostEfficiencyMetricSource` also relies on the Raw Metrics Service to read the HTTP request metrics from the time series DB. It uses the Composed Metrics Service to obtain the cost efficiency composed metric source for the current workload to calculate the cost efficiency. The modular approach of the composed metrics allows changing parts of the implementation (e.g., use a different cost provider) without affecting the rest of the composed metrics. Note that even though we use Prometheus in our use case, the implementation of both composed metrics is completely DB-independent – in fact, a DB connector must be initialized by the SLO controller (Prometheus connector in Fig. 5) to provide a `NativeQueryBuilder` for generating queries for a specific DB.

Next, the SLO controller needs to be created. Its bootstrapping code generated by the Polaris CLI initializes the Polaris middleware, the Kubernetes and Prometheus connectors, registers the cost efficiency SLO and its SLO mapping type with the SLO control loop, and starts the control loop. For the `CostEfficiencySlo` class, a skeleton is generated to realize the `ServiceLevelObjective` interface – it must be implemented by developers. Since the cost efficiency composed metric has been developed as a library, we need to call its initialization function during controller startup to register the cost efficiency metric with the Composed Metrics Service.

The SLO control loop monitors `CostEfficiencySloMapping` resources in the orchestrator through the object watch facilities. To this end, the Kubernetes connector provides an implementation of the `ObjectKindWatcher` interface, which relies on the Transformation Service to transform Kubernetes resources using the transformers supplied by the Kubernetes connector as well.

When a `CostEfficiencySloMapping` resource is received by the SLO control loop, the `CostEfficiencySlo` class is instantiated to handle its evaluation, when periodically triggered by the control loop through the SLO evaluation facilities. We use the Composed Metrics Service in the `CostEfficiencySlo` class to obtain the composed metric source for the cost efficiency metric. The current value of the metric is compared to the target value configured by the user and an SLO compliance value is calculated and returned to the SLO evaluation facilities, whose orchestrator-specific parts are realized by the Kubernetes connector. They use the elasticity strategy object kind configured in the SLO mapping instance to create a `HorizontalElasticityStrategy` resource to wrap the `SloCompliance` output and submit that to the orchestrator to trigger the elasticity strategy controller.

B. Qualitative Evaluation

Due to the use of the generic `SloCompliance` (depicted as an interface in Fig. 5) and the dynamic instantiation of the elasticity strategy resource, the cost efficiency SLO does not need to know about the specific elasticity strategy that will be used. Similarly, the horizontal elasticity strategy controller

[13]https://www.kubecost.com

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

Component	Lines of Code	% of Total
Composed Metrics	209	7%
SLO Controller	119	4%
Polaris Middleware	2594	89%
Total	2922	100%

does not require any information on the SLO that has created the elasticity strategy resource. The type of SLO output data is the only link that connects an SLO to an elasticity strategy; apart from having to share the same output/input data type, they are completely decoupled. For example, changing to a vertical elasticity strategy, only entails the user altering the SLO mapping instance, used to configure the SLO, to reference a vertical elasticity strategy object kind instead of a horizontal elasticity strategy object kind.

All orchestrator-specific actions used in the SLO control loop are encapsulated in the object watch and SLO evaluation facilities, as well as the transformers used by the Transformation Service, which, in this use case, are implemented by the Kubernetes connector library. Switching to a different orchestrator, e.g., OpenStack[14], only entails exchanging the Kubernetes connector library for an OpenStack connector library (i.e., importing a different library and changing one initialization function call), the rest of the cost efficiency SLO controller's implementation would remain unchanged. The same applies to changing the type of time series DB used as the source for the raw metrics needed to compute the cost efficiency composed metric: the Prometheus connector library could be exchanged, e.g., for an InfluxDB connector library, without altering the implementation of the cost efficiency composed metric source.

Table I summarizes the line counts of the involved components. The Polaris middleware has the largest part, with 89% of the total code. The reusable total cost and the cost efficiency metrics together add up to 209 lines or 7% of the code. The cost efficiency SLO controller is the smallest part with only 119 lines (4% of the total code), about half of which can be generated by the Polaris CLI. This shows that the usage of the Polaris middleware greatly increases productivity when developing complex SLOs, while keeping them portable to multiple orchestrators and DBs. To better illustrate the usage of the Polaris CLI, we have published a demo video online[15].

C. Performance Evaluation

Our testbed consists of a three-node Kubernetes cluster, with one control plane node and two worker nodes, all running MicroK8s[16] v1.20 (which is based on Kubernetes v1.20). The underlying virtual machines (VMs) are running Debian Linux 10 and have the following configurations:

- *Control plane & Worker1*: 4 vCPUs and 16 GB of RAM
- *Worker2*: 8 vCPUs and 32 GB of RAM

We use a synthetic workload for the performance tests, as this is the best practice for stress tests. To the best of our knowledge, there is no other middleware that offers the same features as Polaris. Out of the production-ready solutions, HPA offers the greatest similarity. However, the realization of composed metrics would require the addition of a custom Kubernetes API server to provide these metrics, which means that it could not compete with Polaris with respect to the lines of code. We conduct two experiments, where we create 100 cost efficiency SLO mappings and let the SLO controller evaluate them at an interval of 20 seconds.

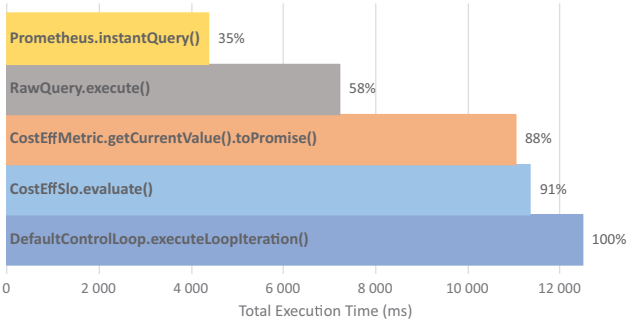

Fig. 6: Average total execution times of `executeControlLoop-Iteration()` and its children across all 300 seconds profiling sessions.

1) SLO Controller Resource Usage: First, we show that an SLO controller built with the Polaris middleware does not consume excessive resources, even when handling numerous SLOs. For this experiment, we deploy the cost efficiency SLO controller to our cluster in a pod with resource limits of 1 vCPU and 512 MiB RAM. We observe the resource usage over a period of 20 minutes using Grafana[17], which fetches metrics from Prometheus. While evaluating 100 SLOs every 20 seconds, the CPU usage stays between 0.2 and 0.25 vCPUs, while the memory usage is between 102 and 140 MiB. Thus, both, CPU and memory usage stay far below the pod's limits and constitute reasonable values for execution in the cloud.

2) Execution Performance of the Polaris Middleware: Next, we demonstrate that the Polaris middleware does not add significant overhead to an SLO controller. To this end, we execute the cost efficiency SLO controller on a development machine (Intel Core i7 Whiskey Lake-U with 4 CPU cores, clocked at 1.8 GHz and 16 GiB RAM) under the Visual Studio Code JavaScript debugger and profiler, while being connected to our cluster's control plane node through SSH. As for the previous experiment, we use 100 cost efficiency SLO mappings to generate load. We execute 3 profiling sessions, each with a length of 300 seconds (i.e., 5 minutes).

Fig. 6 shows a flame chart with the total execution times of all SLO control loop iterations and the major methods invoked by it. The numbers are the mean average values across all profiling sessions. The sum of the execution times of all SLO control loop iterations in a 300 second profiling session is on average 12,480 milliseconds (ms). The SLO control loop itself and the triggering of elasticity strategies using the results from the SLO evaluations only takes about 9% of that time, the remaining 91% are consumed by the

[14]https://www.openstack.org
[15]https://www.youtube.com/watch?v=3_z2koGTExw
[16]https://microk8s.io
[17]https://grafana.com

evaluation of the cost efficiency SLO. The SLO relies on the cost efficiency composed metric, which takes up most of the SLO's execution time. The composed metric sets up one raw metrics query itself for the HTTP request metrics and delegates the creation of the query for the costs to the total cost composed metric. The execution of both raw metrics queries amounts to about 58% of the total SLO control loop execution time. More than half of this (35% of the total) amounts to the query execution in the third-party Prometheus client library. This analysis demonstrates that the evaluation of SLOs using the Polaris middleware is performant and does not show any evidence of bottlenecks.

VII. Related Work

There is a multitude of SLO solutions for the cloud. However, most of them focus on providing one or a few specific SLOs instead of supplying a runtime for creating arbitrary SLOs or decoupling SLOs from elasticity strategies.

The vast majority of big commercial cloud providers, such as AWS [16], Azure [17], and Google Cloud Platform [18], offers simple SLOs, where users can specify a lower and upper bound or an average value for metrics that are directly observable on the system. Horizontal scaling is the most supported elasticity strategy, albeit some cloud providers have not shown additional increases in application performance above certain instance counts [19], suggesting that more elaborate elasticity strategies would be more suitable at that point. Some providers support higher-level SLO specifications, e.g., AWS uses "nines" for the availability of services [20] and the durability of DBs (e.g., "four nines" is equal to a service availability of 99.99%). Some providers allow specifying custom metrics through metrics query languages for existing SLOs, but nevertheless, most big cloud providers do not offer a runtime for implementing additional SLOs, which means that complex SLOs would have to be implemented entirely in the metrics query language.

For Kubernetes, Horizontal Pod Autoscaler (HPA) [11], Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA) [21] are commonly used solutions. However, they all tie a single SLO to a single elasticity strategy. HPA allows scaling out/in, based on an average CPU usage SLO, CA adds or removes nodes to/from a Kubernetes cluster, based on the time that is allowed to pass after a pod can no longer be scheduled, and VPA allows configurable up/down scaling, but does not allow explicit configuration of the SLO, i.e., the decision when to scale is made automatically. Some research tries to improve the performance of VPA [22] and CA [23], but so far, the aforementioned autoscalers remain limited in their extensibility. While HPA provides support for custom metrics, they require the implementation of a custom Kubernetes API server, which leads to additional effort. The experimental kube-metrics-adapter[18] allows expressing complex metrics queries in PromQL. However, this approach is difficult to maintain, especially for complicated metrics that require large queries.

[18]https://github.com/zalando-incubator/kube-metrics-adapter

There are some frameworks that support combining raw metrics into combined metrics, but they are normally not integrated with an SLO runtime. MELA [24] is designed for cloud elasticity monitoring and allows combining metrics using a metric composition language. StreamSight [25] is a language and framework for generating optimized queries for distributed processing of streaming analytics in edge computing, which may also be used to combine multiple metrics.

Some research systems provide flexible runtimes that allow defining custom SLOs, which are decoupled from elasticity strategies. The SYBL [26] language is designed for defining custom metrics and complex constraints, i.e., SLOs, on cloud applications and their components. While its runtime system can be extended with custom elasticity strategies, it is tied to OpenStack. The rSLA [27] language, supports, in conjunction with its runtime facilities, the definition of SLOs based on raw and custom metrics, as well as the triggering of actions upon SLO violations, i.e., elasticity strategies. Despite providing a runtime with support for custom metrics, SLOs, and to some degree elasticity strategies, SYBL and rSLA do not allow passing information resulting from the SLO's evaluation (e.g., the degree of SLO compliance) to the elasticity strategies, which may limit their effectiveness.

VIII. Conclusion & Future Work

In this paper, we presented the Polaris middleware, a flexible middleware system for implementing complex metrics and SLOs that trigger elasticity strategies in an orchestrator- and DB-independent manner. We have motivated the need for the Polaris middleware using a real-world use case and listed its architecture requirements. We presented the design and implementation of the mechanisms that enable our core contributions of 1) the orchestrator-independent SLO controller for periodically evaluating SLOs and triggering elasticity strategies, 2) the provider-independent SLO metrics collection and processing mechanism for obtaining raw, low-level metrics from a time series DB and composing them into reusable, higher-level composed metrics, and 3) a CLI Tool for creating and managing projects that rely on the Polaris middleware. Finally, we used the realization of the motivating use case using the Polaris middleware to evaluate the performance of our middleware and to show that it provides substantial benefits and flexibility when implementing SLOs.

To achieve all goals of the Polaris project [6], we identify the following steps of future work: To facilitate the creation of new SLO mapping types, we will extend the Polaris CLI with the ability to generate CRDs from the TypeScript class of an SLO mapping. We will introduce more complex elasticity strategies that go beyond scaling of workloads by allowing automatic transformations of the topology of a complex cloud workload. Furthermore, we will extend Polaris towards edge computing, by introducing edge-specific SLOs and elasticity strategies, such as for optimizing the usage of a cellular network.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] E. Manoel, M. J. Nielsen, A. Salahshour, S. Sampath K.V.L., and S. Sudarshanan, *Problem determination using self-managing autonomic technology*, 1st ed., ser. IBM redbooks. Austin Tex.: IBM International Technical Support Organization, 2005.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[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] Martin Fowler, "Fluentinterface," 2005. [Online]. Available: https://martinfowler.com/bliki/FluentInterface.html

[15] M. Fowler, *Domain-specific languages*. Upper Saddle River, NJ: Addison-Wesley, 2010.

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

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

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

[19] 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.

[20] 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

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

[22] 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.

[23] 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.

[24] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: elasticity analytics for cloud services," *International Journal of Big Data Intelligence*, vol. 2, no. 1, pp. 45–62, 2015.

[25] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Streamsight: A query-driven framework for streaming analytics in edge computing," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pp. 143–152.

[26] 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.

[27] 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.