SimuScale: Optimizing Parameters for Autoscaling of Serverless Edge Functions through Co-Simulation

Philipp Raith
Distributed Systems Group, TU Wien
Vienna, Austria
p.raith@dsg.tuwien.ac.at

Stefan Nastic
Distributed Systems Group, TU Wien
Vienna, Austria
s.nastic@dsg.tuwien.ac.at

Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
dustdar@dsg.tuwien.ac.at

Abstract—Serverless Edge Computing is growing in popularity, and while commercial providers are starting to offer edge-oriented products, much research is still being done on orchestrating functions (e.g., autoscaling). These approaches range from threshold- to AI-based strategies and support various Service Level Objectives (SLOs), such as Round-Trip-Time (RTT) and resource usage. However, the Quality of Service (QoS) continuously deteriorates due to the dynamic edge-cloud continuum and static parameterization of orchestration strategy parameters. Platforms must adapt the orchestration parameters during runtime to counteract this drift that causes SLO violations. To this end, we introduce the Orchestration Parameter Optimization Problem (OPOP), which aims to find parameters for orchestration strategies to minimize SLO violations. We propose a novel self-adaptive Simulation-based Scaling (*SimuScale*) approach that uses co-simulation to solve OPOP for autoscalers during runtime. *SimuScale* uses live monitoring data to feed the simulation and perform parameter optimization. Our Proof of Concept is integrated with Kubernetes and evaluated on a real-world edge-cloud testbed. While this work focuses on a threshold-based autoscaler, it can be extended to optimize other orchestration components (e.g., schedulers). Our experimental results show that *SimuScale* finds parameters that decrease RTT SLO violations between 15% and 40%. *SimuScale* also can reduce resource usage by 29.87% while maintaining the target 95*th* RTT percentile. Moreover, it can reduce variance caused by different request patterns, making orchestration strategies more resilient in realistic scenarios.

Index Terms—Serverless Edge Computing, Co-Simulation, Self-Adaptive System, Autoscaler, Edge-Cloud Continuum

I. Introduction

Serverless Computing promises to prevent customers from manually managing function deployments by autonomously orchestrating function instances in response to incoming requests. Serverless Edge Computing [1], [2] can be a key enabler for emerging application paradigms, such as Edge Intelligence [3], that require all resources in the edge-cloud continuum to work together [4], [5]. Serverless platforms build on orchestration strategies to autoscale function instances, schedule them, and route requests from clients. Orchestration strategies adapt functions to satisfy operational goals, such as service level objectives (SLOs), which include user- and platform-oriented ones. For example, users want low latency response times to reduce cost and satisfy function requirements. Platform providers try to satisfy customer SLOs and avoid penalties. At the same time, they aim to achieve high resource efficiency, making it challenging to balance between customer and platform provider goals.

Research in orchestrating deployments has been extensively discussed and evaluated in the literature. Autoscaling has evolved over the years, from grid computing to cloud computing, and nowadays moves towards the edge [6], [7]. However, an issue that remains is finding appropriate parameters and drifting orchestration strategies. Proper parameterization of orchestration strategies, in general, is challenging. Autoscaling strategies can use various optimizations, ranging from heuristic- , threshold-, to AI-based, etc. [7]. Which can focus on different aspects, such as reducing data movement [8], guaranteeing response times [9], or reducing resource usage [10]. However, the optimal performance of those strategies is bound to the proper parameterization, which depends on the environment (e.g., infrastructure, deployed functions, client request patterns, etc.) [8], [11].

Straesser et al. [12] present several challenges for autoscalers in production, and one is finding optimal parameters, which can become increasingly complex with different approaches. Al et al. [13] show the impact of the varying CPU threshold for autoscalers on cost and performance. Calzarossa et al. [14] conclude that the performance of autoscalers depends on the configurations, which in turn is affected by the request patterns, giving reason to adapt configurations during runtime. Especially in Serverless Edge platforms spanning multiple geo-distributed clusters, request patterns can vary from one to another. For example, the Shanghai Telecom dataset [15]–[17], which contains real-world mobile user request patterns, shows a high variance of concurrent users over time, causing *Orchestration Strategy Drift*. Therefore, an open challenge is putting orchestration strategies into production and finding parameters to satisfy operational requirements during runtime to avoid *Orchestration Strategy Drift*.

These drifts are analogous to the concept of *Concept Drift*, established in Machine learning operations (MLOps) [18]. Concept Drifts occur when the input changes over time and AI models need to be retrained. A common indicator is a decreasing model performance. In the same way, we observe increasing SLO violations due to outdated orchestration parameters as request patterns and environments change. To find appropriate parameters and prevent this drift, we introduce the *Orchestration Parameter Optimization Problem* (*OPOP*),

which tries to minimize SLO violations by updating orchestration parameters during runtime.

To this end, we introduce a self-adaptive Simulation-based Scaling approach (*SimuScale*) that can efficiently tune orchestration strategy parameters during runtime. It uses a trace-driven open-source simulator *faas-sim* [19] to run the optimization process of finding autoscaling parameters and solving *OPOP*. *SimuScale* is implemented as Proof-of-Concept (PoC) that works in tandem with Kubernetes. The Co-Simulation allows us to evaluate new orchestration strategy parameters efficiently, thus enabling parameter optimization. It uses live monitoring data to replicate the state of the real-world system as closely as possible and can then use optimization techniques to find appropriate parameters. We present two approaches: a random parameter search and a gradient descent-based one.

The contributions of this paper are as follows:

1) Based on the issue of *Orchestration Strategy Drifts*, we introduce our model to solve *OPOP*. This aims to minimize SLO violations by selecting optimal parameters for orchestration strategies.

2) We propose *SimuScale*[1], a self-adaptive approach to automatically adapting threshold-based autoscalers during runtime by using co-simulation. *SimuScale* works in tandem with Kubernetes, a container orchestration platform commonly serving as a base for open-source Serverless platforms. Because we use a simulation, we evaluate the impact of the time horizon we simulate ahead. Results show that a short time horizon (*i.e.,* 30 seconds), results in lower SLO violations for high-intensity request patterns.

3) A random and a gradient descent approach are used to solve *OPOP* and integrated into *SimuScale*. We evaluate our approach on a real-world testbed that demonstrates *SimuScale's* ability to reduce SLO violations by tuning the autoscaler's parameters. Experiments show that *SimuScale* can reduce SLO violations between 15% to 40% while overcoming the challenge of dynamic request patterns.

Our evaluation on a real-world testbed shows that the co-simulation approach can facilitate self-adaptive platforms. The results indicate that our platform prototype can reduce SLO violations and resource usage effectively.

The remaining work is structured as follows. Section II introduces Orchestration Drifts in more depth and shows preliminary results that underline the issue of static autoscaler parameters. Section III introduces our model to solve *OPOP* that builds on three components: a Serverless Edge Platform, the Parameter Optimization, and the Co-Simulation. Section IV presents details of our PoC implementation and the experiment setup. Section V shows and discusses the results obtained from our empirical evaluation of *SimuScale*. Section VI presents related work, and and Section VII introduces future work and concludes this work.

[1]https://github.com/edgerun/simu-scale

II. MOTIVATION

This section introduces the issue of drifting orchestration strategies and highlights its impact on functions based on results from preliminary experiments.

A. Orchestration Strategy Drift

The parameters of an orchestration strategy depend on the implementation. They are crucial as they dictate the strategy's behavior and must be appropriately set to perform well in real-world systems [8], [12]. For example, an autoscaler that tries to satisfy a certain round-trip-time (RTT) might neglect resource usage. Different request patterns make this even more difficult and render seemingly *obvious* parameters unsuitable, like setting the threshold to the target RTT. We showcase this behavior in Section II-B but focus for now on the overall problem of drifting orchestration strategies and the impact on platform performance.

For that, we quantify a platform's performance through Service Level Objectives (SLOs). SLOs are measurable targets of Key Performance Indicators (KPI) that quantify deployed functions' Quality of Service (QoS). KPIs include the state of the function (e.g., availability, latency, cost) or the state of the infrastructure (resource or network usage). For example, an SLO might specify that the function's RTT should have a 95*th* percentile of 0.6s. A violation occurs any time the RTT is higher than 0.6s, resulting in penalties for the platform provider. Customers usually impose them, but we also consider platform-oriented ones, such as resource usage. The platform's task is to balance these goals and avoid SLO violations.

However, serverless systems are highly dynamic and not static. For example, request patterns can differ significantly across compute clusters, even within a single city. The Shanghai Telecom dataset, based on real-world mobile user request patterns, shows this [15]–[17]. Therefore, setting orchestration strategy parameters statically is not enough as systems change over time, causing more SLO violations to happen [12], [13]. We call this issue *Orchestration Strategy Drift* and investigate its impact on our testbed by running request patterns from the Shanghai Telecom dataset using different autoscaler parameters. Specifically, we look into the negative impact static parameters can have under varying request patterns on performance and resource usage.

B. Drifts in Practice

In the following, we motivate and showcase the *Orchestration Strategy Drift* in practice on our testbed. We focus on the impact of different request patterns and extract three patterns from the Shanghai Telecom dataset [15]–[17]. The dataset contains mobile user connections of over 2000 base stations in Shanghai. We introduce three scenarios (*S1*, *S2*, *S3*) that combine request patterns of different base stations. *S1* being the one with the highest number of requests and *S3* the one with the lowest. We expect the RTT's 95*th* percentile of the deployed function to be around 0.6s. This includes network latency and any queuing delay in the system. Further details of

(a) 95*th* Percentile RTT in seconds

(b) SLO Violations in % (Target: $0.6s$)

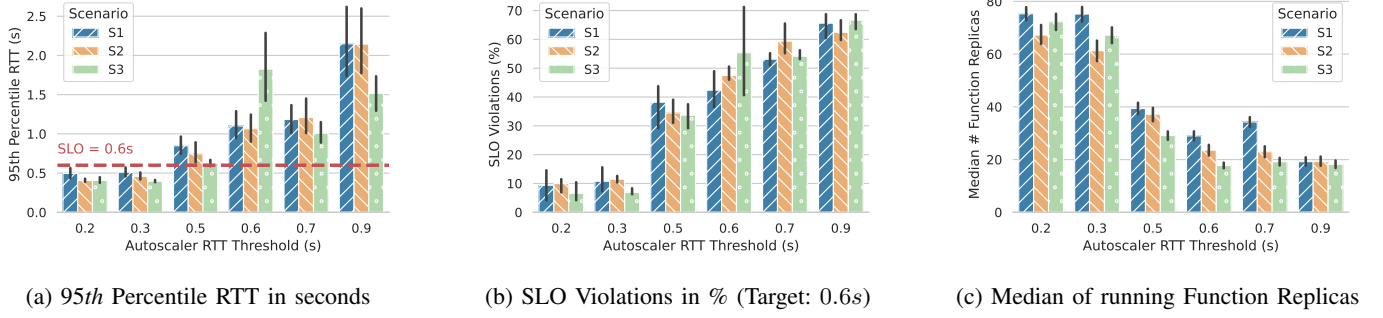(c) Median of running Function Replicas

Fig. 1: Autoscaler Drift across different Request Patterns from Shanghai Telecom Dataset [15]–[17].

the autoscaler and the experiment setup, including the testbed, can be found in Section IV.

As seen in Figure 1, we set the autoscaler's RTT threshold parameter in the range of 200ms to 900ms by using the 95*th* percentile. The results show that the SLO violations differ between scenarios, and in our case, no setting could have a steady RTT across all scenarios. We also observe that setting the RTT threshold to $0.6s$ (the SLO) has a significant variance between scenarios, making it challenging to find the right setting. Moreover, Figure 1c shows the resource usage (i.e., running replicas) per threshold setting. While lower RTT thresholds imply better performance, they also lead to higher resource usage. These experiments focus on a threshold-based autoscaler; other papers explore similar topics and indicate that parameterization is not easy and heavily depends on the environment [8], [12]. Based on those findings, we introduce our model to overcome *OPOP* and present *SimuScale*, a Simulation-based Scaling approach that we implement on a real-world system.

III. SimuScale - Orchestration Parameter Optimization

Serverless Computing platforms, responsible for avoiding SLO violations through function scaling, scheduling, and request routing, grapple with the complexity of maintaining optimal orchestration parameters over time. We previously identified this issue as *OPOP* and introduce our model to solve it, which is an unconstrained optimization framework. Its core objective is to find orchestration strategy parameters that minimize the SLO violations. The SLO violations measure the discrepancy between the observed KPIs and the desired SLO targets. Therefore, the goal is to minimize a function $f(\mathbf{x})$, $\mathbb{R}^n \to \mathbb{R}$, where the input $\mathbf{x}$ represents the orchestration parameters and f estimates the SLO violations resulting from those parameters for a given scenario. We leave the decision variable $\mathbf{x}$ unconstrained to emphasize the general applicability towards various orchestration strategies. f uses a function h to estimate the KPIs based on $\mathbf{x}$ which are put into a cost function v to calculate the SLO violations (i.e., the discrepancy between the target SLO and the observed KPIs).

Figure 2 depicts the individual components of our model to solve *OPOP* — *SimuScale*. The three high-level compo-

nents contain the real-world Serverless Edge Platform (Section III-A), the Parameter Optimization (Section III-B), and the Co-Simulation (Section III-C). The parameter optimization runs at intervals and uses co-simulation to optimize the parameters and minimize SLO violations. The Co-Simulation receives the latest platform state to mimic the real world as closely as possible. This includes the orchestration components, clients, and worker nodes on which function instances run. The Co-Simulation represents the implementation of the function h to estimate KPIs, while the Parameter Optimization uses different *modes* that describe the implementation of the cost function v. The Co-Simulation enables us to perform *what-if* scenarios considering the platform's state, the request pattern, and parameters. We take a snapshot of the current state, transform the simulation can use, and execute scenarios based on the known request pattern. The optimization uses Co-Simulation to find the suitable parameters, minimize the SLO violations, and send them to the platform, where the orchestration components are updated. Next, we lay out the details of each component by starting with the Serverless Edge Platform.

A. Serverless Edge Platform

The Serverless Edge platform is geo-distributed and consists of multiple clusters. It is based on a decentralized orchestration architecture [10] and includes worker nodes that host function replicas and clients that spawn requests. Orchestration strategies include schedulers, autoscalers, and load balancers that route client requests to the function replicas. It consists of a decentralized autoscaler implementation in which each cluster runs one instance. Autoscaler instances monitor their cluster and scale in or out by sending messages to the global scheduler. The global scheduler determines a cluster to start or remove function replicas. A local scheduler selects a worker node to start a new function replica. These build the fundamental components of our envisioned Serverless Edge Platform. However, we want to emphasize that *SimuScale* is not constrained to this particular architecture. For example, *SimuScale* is not concerned with the implementation details of schedulers, autoscalers, or load balancers. The implementation of h is responsible for capturing the platform's logic, including orchestration and compute clusters, to estimate the KPIs. To
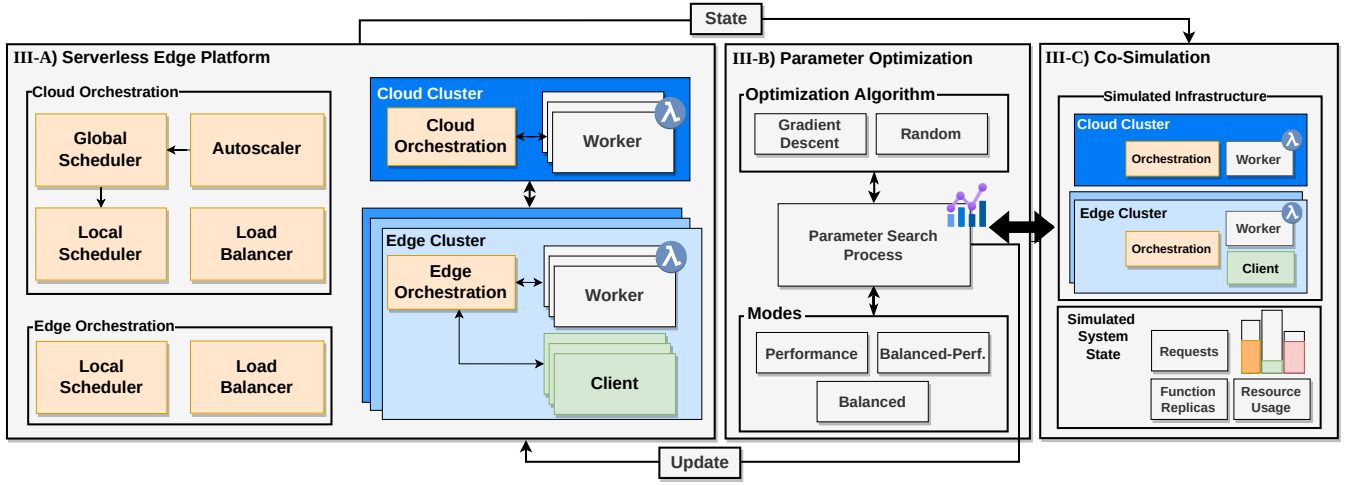
Fig. 2: *SimuScale* - A high-level system overview

this end, we present our Parameter Optimization approach that uses Co-Simulation to run an optimization algorithm and updates the real-world components afterward, as seen in Figure 2.
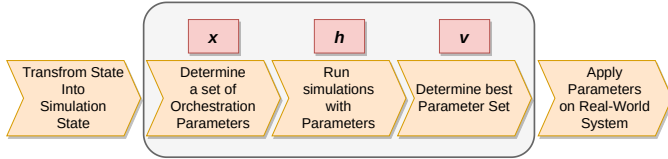

Fig. 3: Parameter Optimization using Co-Simulation

B. Parameter Optimization

We introduce the Parameter Optimization process to solve *OPOP*. As shown in Figure 2, the Parameter Optimization consists of three parts: the Optimization Algorithm, The Parameter Search Process, and the Modes. The Optimization Algorithm is used to find parameters minimizing the SLO violations. The Parameter Search Process takes the Optimization Algorithm and a Mode as input and uses the Co-Simulation to find parameters that reduce SLO violations, effectively solving *OPOP*. The Modes can be viewed as implementations of the cost function v and the Co-Simulation as implementation of h.

A basic approach to finding parameters is depicted in Figure 3. First, we transform the real-world state into the simulation state. Then, we generate parameters based on the current ones. Next, we run simulations for each parameter set, calculate the SLO violations for each setting, and choose the parameter set with the lowest violations. We apply these parameters to the real world. This builds the base implementation of our model to solve *OPOP*. The orchestration parameters represent the input, $\mathbf{x}$, the Co-Simulation represents the implementation of h, while the step of determining the optimal parameter set is the implementation of v. However, we need an optimization approach to find parameters that minimize SLO

violations, which consists of the optimization algorithm and the cost function.

We use two optimization algorithms: a random parameter search and one based on the Gradient Descent (*GD*) [20] algorithm. The *Random* approach creates n parameter sets and starts n simulation in parallel. Each parameter set is generated randomly by applying a percentage adjustment to each parameter. Specifically, we draw a random percentage number from a uniform distribution $[-15, 15]$ and apply the adjustment to the current orchestration parameter value. The *GD* approach approximates the gradient of the cost function by adding and subtracting an epsilon value e to each parameter $x \in \mathbf{x}$, assuming $\mathbf{x}$ is of length m. We approximate the gradient $\dfrac{\partial f}{\partial x_i}$ using the central approximation method. And then apply the update, with a learning rate of α:

$$x_i = x_i - \alpha \cdot \frac{\partial f}{\partial x_i}$$

In every iteration, we perform these steps for each $x_i \in \mathbf{x}$.

These two approaches differ in complexity in finding suitable parameters. For example, Gradient Descent has two stopping criteria and simulates the system sequentially to determine the parameters. The random approach only iterates once and is, therefore, easy to parallelize, whereas we can only parallelize each iteration of the Gradient Descent. Besides the approaches to perform the optimizations, we also need to determine the *best* parameter settings. For example, how can we determine which configuration is the most suited? To answer this, we introduce *modes* that dictate how results are judged. Specifically, this addresses implementing the function v. v acts as an objective function and calculates the SLO violation cost based on the observed KPIs.

We introduce three modes that vary in the metrics they use and the goal they follow. The first one, *Performance*, only takes the RTT into account by using the 95*th* percentile to calculate the difference to the configured SLO target (SLO_{RTT}). This *mode* focuses on performance and, therefore, will have

increased resource usage while keeping performance SLO violations down. The other two modes use, in addition, a second objective function that takes the resource usage into account by calculating the difference to a target resource usage (SLO_{CPU}). Resource usage is the average number of CPU cores used, and the goal is a percentage of the target resource usage. Based on that, we introduce a cost function that combines performance resource usage by building the weighted sum of the deviation from the target:

$$v_{\text{combined}} = \frac{RTT_{P95}}{SLO_{RTT}} * w_p + \frac{CPU_{mean}}{SLO_{CPU}} * w_r$$

This allows us to balance the importance of each error and introduce the remaining two modes: *Balanced* and *Balanced-Perf.*. *Balanced* sets $w_r = 1$ and $w_p = 0.5$, while *Balanced-Perf.* reverses the weight settings.

C. Co-Simulation

The Co-Simulation estimates the platform's behavior (i.e., KPIs) based on the orchestration parameters. For this, we transform the state of the real-world platform in the Co-Simulation and run different simulation scenarios. It allows us to simulate different scenarios ahead of time and use the KPIs to calculate the SLO violations, as shown in Section III-B. In the following, we describe the transformation of the real-world into the simulation state. The mapping includes orchestration strategies, worker nodes, clients, and network connections between clusters and individual nodes, as shown in Figure 2. The simulated system state contains the latest information about function replicas, requests (i.e., completed function invocations), and resource usage. The information about finished requests contains the start and end timestamps from the client and the actual function execution (i.e., the duration without any queuing), which load balancers were used, and the function replica that processed the request. Function replica state includes the worker node they were placed on and the state (e.g., running, deleted, etc.). It is also important to transform the function replicas currently being scheduled (i.e., pending) and the resource usage, such as CPU and memory usage. The simulated infrastructure and system state can be extended to incorporate other aspects. *SimuScale* constantly observes the latest state to perform the transformation every time the Parameter Optimization is executed.

D. Challenges

An issue prevalent across many orchestration implementations is the challenge of setting the initial parameters. The impact of those can be crucial to the overall performance of functions and the resulting resource usage. A challenge for *SimuScale* is the ability to self-adapt from seemingly wrong settings, which can be seen as convergence to the optimal value and is important for *SimuScale's* ability to handle worst-case scenarios. Therefore, validating whether *SimuScale* can handle this is essential.

Another challenge is to set the right *Simulation Time Horizon*. That is, how much time should each scenario simulate?

For example, if we set the *Simulation Time Horizon* too short, we might miss critical changes in the request pattern. On the other hand, each simulated scenario might take longer with an increasing *Simulation Time Horizon*, thus resulting in fewer updates. Another aspect to consider is when putting *SimuScale* into production; tools must be used to estimate future request patterns, which probably worsen with longer time horizons. Therefore, finding the right *Simulation Time Horizon* setting is critical.

Moreover, we investigate what impact *SimuScale* has on SLO violations and resource usage compared to the static deployment of orchestration strategies. Especially when considering different request patterns and if the modes are deterministic regarding their goal. We are addressing all these questions and challenges and our real-world implementation of *SimuScale* in the following.

IV. IMPLEMENTATION & EVALUATION

We evaluate *SimuScale*[2] using the two optimization approaches: Gradient Descent (*GD*) and *Random* on a real-world testbed. We compare it to the *static* case, where parameters are only set once initially. This serves as a baseline and replicates the behavior expected nowadays on serverless platforms. The following outlines our implementation of *SimuScale* and details of the experiment setup.

A. Serverless Edge Platform

Our PoC Serverless Edge platform is built on Kubernetes, a container orchestration service, and is split into multiple clusters - from cloud to edge. Users can send requests to the load balancers, which forward them to running function replica instances, which the autoscalers and schedulers manage. *telemd* pushes monitoring data via Redis' publish/subscribe feature. The orchestration components subscribe to these metrics and are all implemented in Python, each using an in-memory cache for the system state. The system state includes resource usage, information about requests, running application instances, etc. The orchestration components run as individual containers in the cluster for which they are responsible.

1) Orchestration Strategies: To comply with our model presented in Section III-A, we must implement an autoscaler, global and local schedulers, and load balancers for the decentralized orchestration architecture. The autoscaler implementation mimics the official Kubernetes Horizontal Pod Autoscaler[3], which is commonly used in open-source serverless platforms [21]. We denote this strategy as *HPA*. It is based on the round-trip-time (*RTT*) of requests per cluster. Based on the current metric value, the *HPA* estimates how many replicas are needed to satisfy the threshold value. The threshold value is a parameter that we denote as thr_{RTT}. It uses the following formula to calculate the desired number of function replicas:

$$desiredReplicas = currentReplicas \times \frac{RTT_{P95}}{thr_{RTT}}$$

RTT_{P95} is the 95*th* percentile of the observed RTT. Other parameters include *reconcile interval*, *lookback*, and a *percentile*. The *reconcile* interval determines the time the autoscaler waits before executing again, the *lookback* determines how far we look into the past when calculating the current value, *i.e.,* RTT_{P95}. The threshold (thr_{RTT}) is the parameter we optimize in each experiment. HPA and open source platforms that re-use it default to a *reconcile interval* of 15 seconds [22], [23]. We set the *percentile* to 95 and the *lookback* to 10 seconds across all experiments.

If the autoscaler decides the platform should create or remove function replicas, it sends a message to the global scheduler. The global scheduler implements a simple network-aware strategy by first prioritizing clusters from which the scale action stems. If the origin cluster is out of resources, it chooses the nearest cluster based on network latency. The local scheduler is implemented like the default Kubernetes scheduler, filling up all worker nodes while balancing out used resources. Both schedulers are kept simple on purpose to focus on the autoscaler and the optimization of its parameters.

B. Co-Simulation

We use the open-source trace-driven Serverless Edge Computing platform simulator, *faas-sim* [19], to replicate the real-world in the cluster. However, to limit the scope of this work, we make the following assumptions. (1) we know the workload pattern, (2) we know the network configuration between nodes and clusters, (3) we do not simulate any node failure that can happen in the real world, and (4) we assume that all function images are stored already on each node. These assumptions would otherwise require solutions outside this work's scope, as we want to focus on the efficacy of the co- simulation's ability to adapt to real-world orchestration components. Specifically, we need a simulator that can replicate the behavior of our components in the simulation. We use a Python API to implement the core logic of all orchestration components once and re-use it in the simulation and the real-world[4], the orchestration strategy implementations can be found on Github[5]. Besides avoiding code duplication and guaranteeing parity in terms of logic, the simulator uses real-world traces to feed the simulation. In contrast to the popular CloudSim [24], it does not rely on the knowledge of how many instructions an application takes but rather on real-world measurements. In our case, we conduct profiling experiments to determine the execution time of the deployed function and fit a logarithmic distribution from which we sample [19], [25]. The profiling experiments invoke the function, we use for the evaluation, on each host and allow us to fit a logarithmic distribution from which we sample in the simulation. The real advantage of this approach lies in the support for future applications that can be observed in a black-box manner without counting executed instructions. A challenge of such a system is to communicate the state efficiently. The Co-Simulation runs in a central location and always keeps the global state in-memory. Therefore, powerful machines can be used to execute the optimization.

C. Parameter optimization

We implement the Parameter Optimization through a *Random* and *GD* approach to minimize the SLO violations of f. The input $\mathbf{x}$ of f contains the target duration (thr_{RTT}) of each autoscaler instance. The *Random* approach generates n new autoscaler parameters that each is being evaluated through a simulation. In contrast to the simulations in *GD*, we can execute all n simulations at the same time. Specifically, we generate *5* parameter sets, and each autoscaler parameter is randomly set in the range of $\pm15\%$ of the original value. The *GD* settings were chosen based on a set of preliminary experiments. As mentioned, we must avoid long-running optimizations to avoid updating the autoscaler based on outdated data. Therefore, we set the number of iterations to *3*, the learning rate to *1e-1*, and the epsilon value to *0.2*. These settings must be adjusted to the underlying simulation and have proven feasible for our setup. Besides the optimization approaches, we also implement three *modes*, already introduced in Section III-B: *Performance*, *Balanced* and *Balanced-Perf*. *Performance* only considers performance SLO violations, *i.e.,* using the RTT, while the other two incorporate performance and resource aspects. To this end, we set weights for *Balanced* and *Balanced-Perf.* that determine their focus. The former favors resource usage (i.e., $w_r = 1$, $w_p = 0.5$) and the latter reverses them (i.e., $w_r = 0.5$, $w_p = 1$). The weights are set according to their goal. For all modes $SLO_{RTT} = 0.6$ and $SLO_{CPU} = 50\%$. That means all modes have 0.6s as 95*th* percentile target and should not use more than 50% of available CPU cores. We evaluate all approaches and modes, resulting in six combinations.

D. Experiment Setup

Besides implementing *SimuScale*, we also highlight details of the remaining experiment setup. This ranges from the testbed specification to the function we use to test *SimuScale* and the request patterns synthesized from a real-world dataset.

1) Testbed: The experiments run on a real-world testbed that consists of three compute clusters using an end-to-end evaluation framework for edge-cloud resource management [25]. We denote the edge clusters as *EC1* and *EC2*, while the one mimicking a cloud is called *CC*. Our testbed contains 15 VMs with different sizes to mimic a heterogeneous environment, similar to related works [26], [27]. Each VM has between 2 to 8 cores at 2.1GHz and 3 to 16GB of memory — we set the platform to spawn at most 90 function instances. We use the Linux network traffic shaping tool *tc* to emulate network latency between the clusters. Between cloud and edge clusters, we inject a network latency of 60ms, and between edge clusters, 30ms. The clients connect to the Edge Clusters and send the requests to the respective load balancer instances in each cluster.
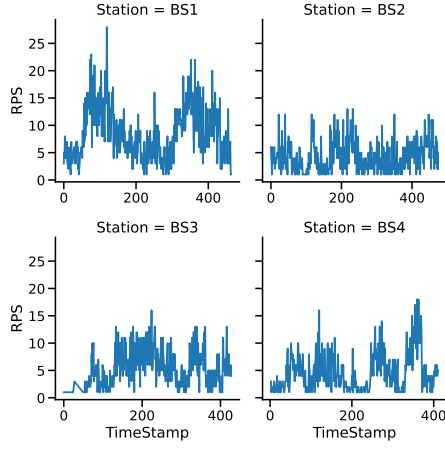
---

[4]https://github.com/edgerun/faas

[5]https://github.com/edgerun/faas-optimizations/

Fig. 4: Base station request patterns

Scenario	EC1	EC2
S1	BS1	BS2
S2	BS3	BS4
S3	BS2	BS4

TABLE I: Request pattern to Edge Cluster mapping.

2) Function: The deployed function is CPU-bound and calculates a specified number of digits of π. This enables us to mimic short-running CPU-focused functions and vary the execution duration. Due to system delay, we expect the 95*th* percentile to be around 600ms. As we do not focus on the most optimal function placement in this work, we expect the cloud cluster to host function replicas and incur network overhead. The function is chosen to mimic a low-latency AI inference service, such as is typical for Edge Intelligence [10].

3) Request Patterns: An important part of our experiments is the use of request patterns that exhibit different degrees of requests over time. The request patterns are based on the Shanghai Telecom dataset [15]–[17], and we extract them from four base stations. We chose the base stations based on the number of connected users and their patterns. Specifically, we have taken the profiles of the following base stations and assigned each an ID from which on we use to refer to them: *31.218201/121.487151* (BS1), *35.379598/116.072359* (BS2), *31.129955/121.336848* (BS3), *31.395915/121.363062* (BS4). We synthesize request patterns for our experiments by extracting each base station's general pattern over 48 hours and trimming it to 10 minutes of actual requests. The patterns per base station are shown in Figure 4. Base station A has the highest requests (~ 4000), while the others have each around ~ 2000. The requests per second are based on the number of users connected at each point, each sending three requests. The interarrival times are drawn from a Poisson distribution. Based on these profiles, we create three scenarios that assign a request pattern to an edge cluster in our testbed. The scenarios and request pattern mappings are shown in Table I.
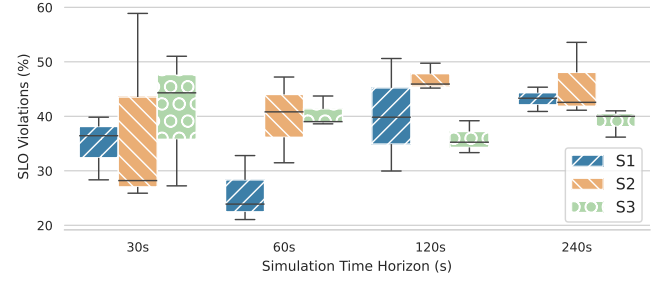

Fig. 5: SLO Violations under different Sim Time Horizons

V. RESULTS

A. Key Performance Indicators

In the following, we focus on three KPIs: SLO Violations in %, 95*th* percentile of the Round Trip Time in seconds, and the median number of running replicas over the experiment. The SLO violations are calculated by grouping the requests into windows (*i.e.,* 1 second) and calculating the 95*th* percentile. A violation occurs if the value exceeds the SLO (*i.e.,* 0.6s). Based on that, we count how often an SLO violation occurred relative to the complete experiment.

B. Simulation Time Horizons

First, we look into the impact the *Simulation Time Horizon* has on the results. For this, we run shortened experiments with the *Balanced-Perf.* mode using four duration settings that we simulate ahead: *30*, *60*, *120* and *240*. The results of these experiments decide which setting we use in the remaining experiments. Figure 5 shows the median SLO Violations in (%) across the duration settings and scenarios. We observe that shorter *Simulation Time Horizons* positively impact the SLO violations. Specifically, the two scenarios with more requests (*i.e., S1* and *S2*) had the lowest median SLO Violations with the duration set to *30* seconds. However, a positive trend exists when looking at *S3*, where a longer time horizon lowers the median number of SLO violations and reduces the variance. We believe these results confirm our assumption of shorter *Simulation Time Horizons* allowing the system to update more often, making it more resilient against bursty workloads. While longer *Simulation Time Horizons* allows the platform to find more suitable parameters in case the workload is relatively stable. We select the *30* seconds *Simulation Time Horizon* as our default for the remaining experiments.

C. Static vs. Co-Simulation

Next, we look into the results when setting the initial autoscaler RTT threshold to 0.6s (*i.e.,* $thr_{RTT} = 0.6$). The results determine whether *SimuScale* can reduce the SLO violations and its impact on resource usage. Figure 6 shows that *SimuScale* can reduce the SLO Violations by up to 15% on average across all scenarios when using the *Balanced-Perf.* mode and *GD* optimization in comparison to *Static*. Moreover, the minimal exhibited variance shows that *SimuScale* is
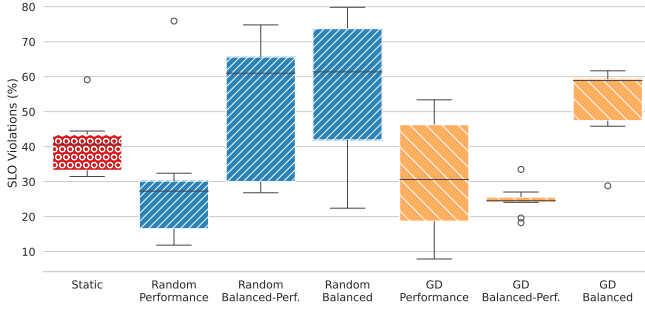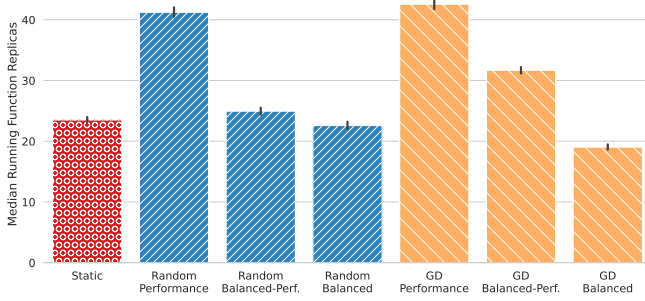
Fig. 6: SLO Violations - $thr_{RTT} = 0.6s$
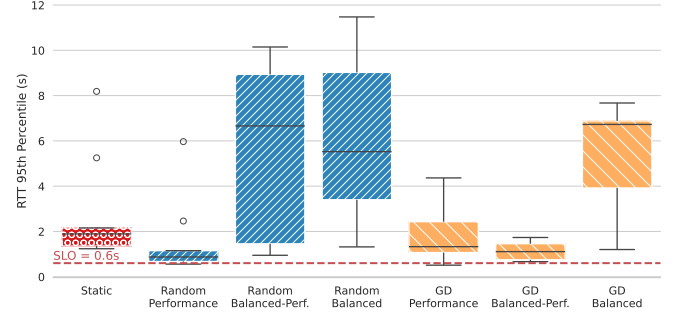

Fig. 8: RTT P95 - $thr_{RTT} = 0.6s$

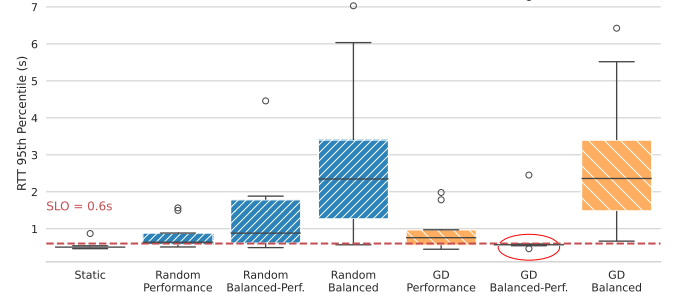Fig. 7: Resource Usage - $thr_{RTT} = 0.6s$

Fig. 9: RTT P95 - $thr_{RTT} = 0.3s$

effective across scenarios. However, an unexpected result is that the *GD* approach using the *Performance* mode was not able to reduce the SLO violations as much as the *Random*. The resource usage in Figure 7 shows that these two had the highest number of replicas. While this should imply a better performance, it might be less beneficial in edge-cloud systems to spawn that many function replicas due to network latency. We suspect that the *Performance* settings spawned many replicas at the wrong time. Therefore, overall RTT increased in these cases. However, we can also see in Figure 8 that the RTT for *Performance* and *GD - Balanced-Perf.* were lower than the *Static* setting. Based on that, aggressive modes, such as *Performance*, can harm the overall SLO compliance in contrast to those more reluctant to increase resource usage in favor of performance. The *Balanced* modes, while using the least amount of replicas, performed worse in terms of performance. However, looking at the SLO Violations, it becomes clear that using *GD* for optimization reduces variance. We conclude that *SimuScale* can effectively reduce SLO violations while balancing performance and resource usage, *i.e., Balanced-Perf.* only used on average 30% more replicas but keeping the resource usage below the SLO_{CPU} of 50% while reducing the SLO violations on average by 15%.

D. Impact of Thresholds

The following section covers the other two initial autoscaler RTT thresholds we set. Specifically, we set $thr_{RTT} = 0.3$ and $thr_{RTT} = 0.9$. The first setting allows us to observe the platform's performance if the threshold is set much lower than

the SLO, and the second is when the initial threshold is higher. To examine *SimuScale's* ability to adapt the threshold, we want to show it can reduce resource usage while keeping the RTT low and update the threshold setting to reduce SLO violations substantially.

First, we observe the results when setting thr_{RTT} to $0.3s$. Figure 9 shows the median 95*th* percentile of the observed RTT over all scenarios and experiments. Such a low threshold positively impacts the performance of the *Static* autoscaler. It achieved a 95*th* percentile of under *0.6s* without updating the threshold. At the same time, the *GD* approach using the *Balanced-Perf* mode achieved the same with some outliers. The other approaches diverged more from the SLO. Before looking more into the *Balanced-Perf* results, the *GD - Balanced* approach was able to have a similar SLO violation count in the *S1* scenario. In this case, we suspect the high amount of traffic caused the optimization to stay at a low threshold, while in other scenarios, it resulted in increasing thresholds.

Moreover, the *GD - Balanced-Perf.* approach was able to have only 8% SLO violations in *S3*, while the *Static* approach had an average of 6%. Therefore, *SimuScale* was also able to handle a lower threshold setting and, as Figure 10 shows, was able to reduce the resource usage significantly in comparison to the *Static* approach. Especially in the just described scenario, where *SimuScale* only had 2% more SLO violations, it could do so with a 29.87% resource usage reduction. The resource usage decreased overall in all approaches and modes, which explains the rising SLO violations and high RTT.

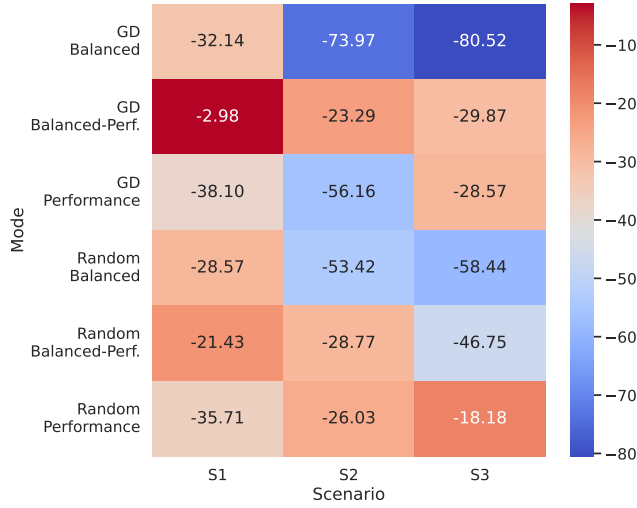While *SimuScale* can handle lower thresholds by having

Fig. 10: Resource Usage Decrease (%) to *Static* ($thr_{RTT} = 0.3s$)

Fig. 11: SLO Violations - $thr_{RTT} = 0.9s$

Fig. 12: RTT P95 Difference (%) to *Static* ($thr_{RTT} = 0.9s$)

a reduced resource and maintaining a similar performance, it also can adapt to threshold settings that are too high. We set the parameter $thr_{RTT} = 0.9s$ in this case. The *Static* variant will perform worse with many SLO violations. However, the focus of these results should emphasize the versatility of *SimuScale*. Figure 11 shows the SLO violations for these experiments. While *SimuScale's Random* approach and the *Balanced* mode failed to adapt quickly enough, it was able to significantly reduce SLO violations using the *GD* approach with *Performance* and *Balanced-Perf.* modes. Across all scenarios, the *Performance* mode reduced median SLO violations compared to the *Static* approach by around 40%. This indicates that the *Performance* mode was too aggressive for other initial threshold settings but appropriate for too high values. Besides reducing the SLO violations, Figure 12 shows the difference of the 95*th* RTT percentile relative to the *Static* experiments. Here we can see that *Performance* reduced the RTT by up to 84% in the case of *S1*.

Based on these results, we conclude that *SimuScale* can reduce resource usage and SLO violations. While the *GD* approach using *Balanced-Perf.* was able to perform better or similar than the *Static* deployment, the other variants require further fine-tuning. Especially the *Balanced* setting focused too much on resource usage, negatively impacting SLO violations and RTT. Also, the *Performance* mode behaved unexpectedly in some cases and was only able to outperform *Balanced-Perf.* when setting thr_{RTT} to 0.9s. Moreover, while the *Random* approach yielded good results, it overall had a higher variance than *GD*. This was especially visible in the last set of experiments, where we set the initial threshold to 0.9s.
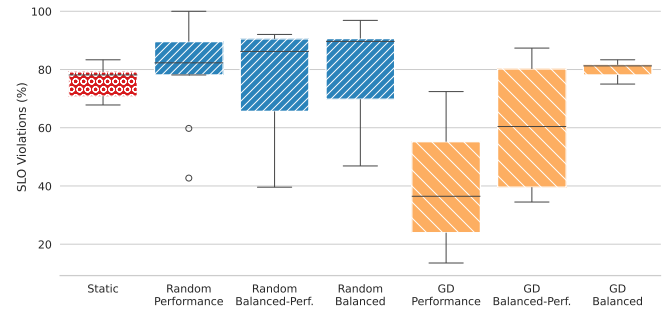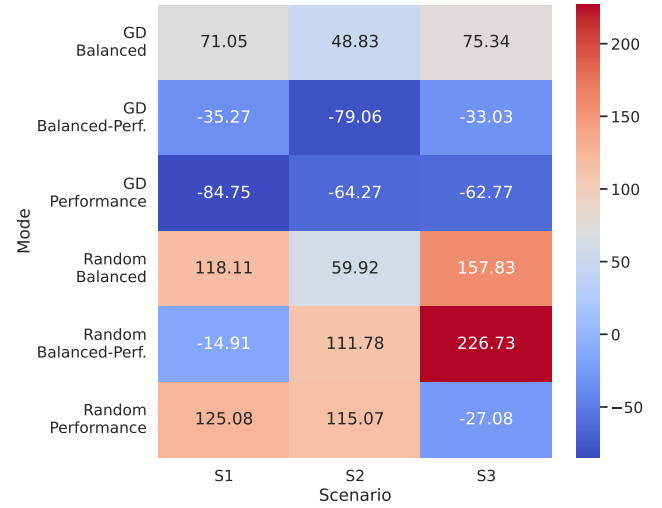
VI. RELATED WORK

A. Orchestration Strategy Parameters

Haidari et al. [28] investigated the impact of different thresholds for different request patterns. Their results indicate that the optimal threshold varies under different loads to minimize resource usage and response times. This underlines our work as we have shown a similar behavior in Section II. Similarly, Taherizadeh and Grobelnik [29] explore different settings for the HPA that they conduct under different request patterns. They investigate how the reconcile interval impacts autoscaling decisions, resource usage, and performance. Interestingly, they also looked into changing the policy when removing application instances by only removing one for a specific set of request patterns. Our work can complement this finding by extending the optimization process to tweak parameters and trying different policies that change the overall behavior. Rossi et al. [30] use Reinforcement Learning agents to update thresholds for autoscaling during runtime. The main differences are that their agent learns and interacts with the real world and that we base our scaling decisions on the RTT, whereas they explore a CPU-based autoscaler. Therefore,

our simulation-based platform can complement their work by using it to speed up the training of the agents.

B. Simulations for Resource Management

Simulations of real world systems allow decision makers to assess *what-if* scenarios that can guide the selection process [31]. In the context of orchestration systems (e.g., microservices, functions, containers, etc.) this can help providers efficiently adapt system components in dynamic environments [32]. Specifically, the adaptation of autoscaler or scheduler components profit from simulation-based adaptations in multiple ways.

Depending on the domain and use case, simulations unite multiple simulations together in a single cohesive simulation unit that can reproduce the behavior of a real complex system. These simulations are called coupled simulations (co-simulations) [33] and have been successfully used in improving container scheduling performance [32], [34]. Like co-simulations, symbiotic simulations use simulations to direct real-world systems [35], [36]. Onggo et al. [35] identified three symbiotic systems combining real systems and simulations and differ in the interactions between the two. For example, in the *Decision Support Symbiotic Simulation System*, decision makers (e.g., humans) use the simulation's output to direct the real system. In contrast, in the *Control Symbiotic Simulation System*, simulation directly influences the physical system.

In our work, we implement a PoC that integrates into Kubernetes and mixes the *Control Symbiotic Simulation System* approach and co-simulations to solve *OPOP*. The use of simulations for resource management during runtime has been explored in some other works. Frey et al. [37] use a simulation-based Genetic Algorithm to find optimal architecture configurations for cloud systems. This work complements our work, as we focus on the platform's parameters while they are searching for an optimal architecture. Simulations can be used similarly in that they mimic the real world, and their output is used to influence the orchestration components [32], [34], [36]. Tuli et al. [38] propose CILP, which uses the simulation as a teacher to generate the ground truth with which their AI models learn. This work showcases that simulations can be used to optimize orchestration strategies (i.e., AI-based). While they use it as a surrogate to generate data, our simulation interacts directly with the real-world system by sending autoscaler parameters. *SimuScale* differs from others due to its runtime being able to adapt the autoscaler's parameters in response to request pattern changes and has proven to trade-off resource usage and performance based on high-level parameters. Rausch et al. [8] investigate how to use simulations for tweaking scheduler parameters, but in contrast to our system, they do not evaluate in a real-world setting. However, it showcases that simulations are feasible for resource management and, specifically, tweaking parameters.

VII. Conclusion & Future Work

Serverless Edge Computing presents a promising platform paradigm for the heterogeneous edge-cloud continuum by autonomously managing functions under the SLOs' guarantee. Many research works explore different strategies to autoscale, schedule, or route requests, but in production systems, the Orchestration Strategy Drift arises, which renders initial orchestration parameters unsuitable, causing SLO violations.

Based on that, we propose *SimuScale*, a self-adaptive Simulation-based Scaling approach for Serverless Edge Computing. It uses a co-simulation to guide the optimization algorithm in finding optimal autoscaler parameters. Results show that shorter simulation time horizons favor bursty workloads, while longer horizons can perform better in scenarios with less variation. Our evaluation on a real-world testbed has shown that *SimuScale* can improve the results overall by reducing SLO violations and variance across different scenarios. Especially in instances where threshold parameters are set lower and higher than the SLO, it can reduce resource usage by up to 29.87% while maintaining the same RTT and SLO violations by up to 40% and the RTT by up to 84.75%, respectively. However, the results also show that more investigation is necessary to determine appropriate settings for the optimization process. For example, some optimization goals did not yield the expected results and, therefore, require a detailed look at defining these goals.

Future work aims at extending *SimuScale* to support other autoscaling strategies or update parameters of schedulers. An interesting avenue to make our work usable for AI-based models is using simulation-generated data for training [32]. This can be useful to extend datasets and enhance the rate at which AI models can be updated in new environments. Finally, we want to investigate the uncertainties that are inherently present in such environments [39] and how they can be integrated into the co-simulations to achieve more accurate predictions.

References

[1] S. Nastic, P. Raith, A. Furutanpey, T. Pusztai, and S. Dustdar, "A serverless computing fabric for edge cloud," in *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, 2022, pp. 1–12.

[2] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," in *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017)*. ACM, 2017, p. Article No. 28.

[3] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[4] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[5] P. Raith and S. Dustdar, "Edge intelligence as a service," in *2021 IEEE International Conference on Services Computing (SCC)*. IEEE, 2021, pp. 252–262.

[6] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[7] P. Raith, S. Nastic, and S. Dustdar, "Serverless edge computing—where we are and what lies ahead," *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, 2023.

[8] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[9] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 180–189.

[10] P. Raith, T. Rausch, S. Dustdar, F. Rossi, V. Cardellini, and R. Ranjan, "Mobility-aware serverless function adaptations across the edge-cloud continuum," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022, pp. 123–132.

[11] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *Proceedings of the 5th international workshop on serverless computing*, 2019, pp. 19–24.

[12] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? challenges for production-ready autoscaling," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 105–115.

[13] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2. IEEE, 2013, pp. 256–261.

[14] M. C. Calzarossa, L. Massari, and D. Tessera, "Evaluation of cloud autoscaling strategies under different incoming workload patterns," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, p. e5667, 2020.

[15] Y. Guo, S. Wang, A. Zhou, J. Xu, J. Yuan, and C.-H. Hsu, "User allocation-aware edge cloud placement in mobile edge computing," *Software: Practice and Experience*, vol. 50, no. 5, pp. 489–502, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2685

[16] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.

[17] Y. Li, A. Zhou, X. Ma, and S. Wang, "Profit-aware edge server placement," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 55–67, 2022.

[18] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (mlops): Overview, definition, and architecture," *IEEE Access*, 2023.

[19] P. Raith, T. Rausch, A. Furutanpey, and S. Dustdar, "faas-sim: A trace-driven simulation framework for serverless edge computing platforms," *Software: Practice and Experience*, vol. 53, no. 12, pp. 2327–2361, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3277

[20] H. E. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951. [Online]. Available: https://api.semanticscholar.org/CorpusID:16945044

[21] Knative. (2024) Autoscaling. Accessed: 2024-03-26. [Online]. Available: https://knative.dev/docs/serving/autoscaling/autoscaler-types/

[22] B. Jeong, S. Baek, S. Park, J. Jeon, and Y.-S. Jeong, "Stable and efficient resource management using deep neural network on cloud computing," *Neurocomputing*, vol. 521, pp. 99–112, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231222014850

[23] OpenFaaS. (2024) Autoscaling. Accessed: 2024-03-26. [Online]. Available: https://docs.openfaas.com/architecture/autoscaling/legacy-scaling-for-the-community-edition-ce

[24] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.995

[25] P. Raith, T. Rausch, P. Prüller, A. Furutanpey, and S. Dustdar, "An end-to-end framework for benchmarking edge-cloud cluster management techniques," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, 2022, pp. 22–28.

[26] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021.

[27] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419317931

[28] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, 2013, pp. 256–261.

[29] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Advances in Engineering Software*, vol. 140, p. 102734, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997819304375

[30] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic multi-metric thresholds for scaling applications using reinforcement learning," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1807–1821, 2023.

[31] K. Rehman, O. Kipouridis, S. Karnouskos, O. Frendo, H. Dickel, J. Lipps, and N. Verzano, "A cloud-based development environment using hla and kubernetes for the co-simulation of a corporate electric vehicle fleet," in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 47–54.

[32] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 101–116, 2021.

[33] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: a survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–33, 2018.

[34] S. Tuli and G. Casale, "Optimizing the performance of fog computing environments using ai and co-simulation," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, 2022, pp. 25–28.

[35] B. S. Onggo, N. Mustafee, A. Smart, A. A. Juan, and O. Molloy, "Symbiotic simulation system: Hybrid systems model meets big data analytics," in *2018 Winter Simulation Conference (WSC)*. IEEE, 2018, pp. 1358–1369.

[36] D. Oakley, B. S. Onggo, and D. Worthington, "Symbiotic simulation for the operational management of inpatient beds: model development and validation using δ-method," *Health care management science*, vol. 23, pp. 153–169, 2020.

[37] S. Frey, F. Fittkau, and W. Hasselbring, "Search-based genetic optimization for deployment and reconfiguration of software in the cloud," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 512–521.

[38] S. Tuli, G. Casale, and N. R. Jennings, "Cilp: Co-simulation based imitation learner for dynamic resource provisioning in cloud computing environments," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.

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