Towards Multi-dimensional Elasticity for Pervasive Stream Processing Services

Boris Sedlak*, Andrea Morichetta*, Philipp Raith*, Víctor Casamayor Pujol†, and Schahram Dustdar*†

*Distributed Systems Group, Vienna University of Technology (*TU Wien*), Vienna 1040, Austria.

Email: {b.sedlak, a.morichetta, p.raith, dustdar}@dsg.tuwien.ac.at

†*Engineering Department*, Universitat Pompeu Fabra (*UPF*), Barcelona 08018, Spain.

Abstract—This paper proposes a hierarchical solution to scale streaming services across quality and resource dimensions. Modern scenarios, like smart cities, heavily rely on the continuous processing of IoT data to provide real-time services and meet application targets (Service Level Objectives – SLOs). While the tendency is to process data at nearby Edge devices, this creates a bottleneck because resources can only be provisioned up to a limited capacity. To improve elasticity in Edge environments, we propose to scale services in multiple dimensions – either resources or, alternatively, the service quality. We rely on a two-layer architecture where (1) local, service-specific agents ensure SLO fulfillment through multi-dimensional elasticity strategies; if no more resources can be allocated, (2) a higher-level agent optimizes global SLO fulfillment by swapping resources. The experimental results show promising outcomes, outperforming regular vertical autoscalers, when operating under tight resource constraints.

Index Terms—Elasticity, Edge Intelligence, Reinforcement Learning, Service Level Objectives, Stream Processing

I. INTRODUCTION

IoT adoption has grown significantly in areas where it provides clear value to human society, such as home automation, smart health, and smart city infrastructure [1] – these use cases are characterized by incessant amounts of streamed data. Smart city scenarios, for example, can involve collecting and analyzing video and images, sound, movement, and temperatures through stream processing services. On top of this, these services must operate within runtime requirements, e.g., latency or quality, which are called Service Level Objectives (SLOs). Latest advances in Edge computing help ensure SLOs because they extend the IoT domain with powerful, decentralized computing infrastructure [2] to process IoT data with low latency. Each service in a larger smart city application has specific SLOs to fulfill, which help maintain the overall equilibrium. This work focuses on a scenario where an Edge node provisions the execution of multiple services. We envision a smart city use case where connected devices, such as a video camera and a vehicle, each provide data to a dedicated service for workload execution at the Edge, as illustrated in Figure 1.

Transitioning from the Cloud to decentralized Edge computing platforms can improve real-time processing and privacy preservation; however, there is one thing that Edge computing cannot (yet) provide to the same extent: resources. Ensuring SLOs in the Cloud has a long history, where the successful recipe [3] is to allocate additional resources in case the performance is mitigated. Edge devices can also be equipped with powerful hardware – thus often called Edge servers – but
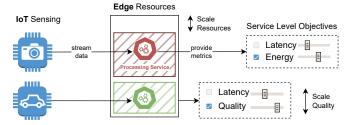

Fig. 1: Processing IoT data at resource-constrained devices; if SLOs are violated, scale either resources or service quality

their ability to scale according to changing demands inevitably hits a resource limit. To circumvent this, it is possible to compose Edge and Cloud computing layers – called the Computing Continuum (CC) [4] – which can provide collaboration between devices, e.g., offloading load within the CC. This work, however, excludes offloading and focuses on methods for guaranteeing SLO fulfillment at a single Edge device.

Ensuring SLOs at the Edge requires alternative elasticity strategies – apart from provisioning additional resources. To provide more flexibility, the solution can be to scale services within three *dimensions* [5]: quality, cost, and resources. In this sense, multi-dimensional scaling strategies [6], [7] can dynamically decide how to optimize the application. For example, in case there are unclaimed resources, provide them to a service, or otherwise, scale down the quality; this is shown in Figure 1. Contrarily, existing autoscalers, even if dynamic, solely scale resources. This can combine vertical and horizontal scaling into a hybrid approach, as done by Lombardi et al. [8], or adjusting SLOs based on environmental changes, e.g., when additional clients join, as done by Horovitz et al. [9]. Lightweight scaling solutions for serverless functions, as done by Zhao et al. [10], also help to ensure SLOs for stream processing. Although these solutions do target resource-restricted devices, they do not harness the potential of other elasticity dimensions in scenarios where no more unoccupied resources can be provisioned.

To fill this gap, this work proposes a novel Edge-based autoscaler that elastically scales stream processing services in two dimensions: resources and quality. Our architecture consists of two layers of scaling agents: in the first layer, each service is extended with a Deep Reinforcement Learning (DRL) agent, which optimizes local SLO fulfillment by scaling resources or quality. These service-specific agents act greedily, which means that they can claim resources that other services might need.

Hence, if the processing resources are exhausted, a higher-layer agent tries to improve global SLO fulfillment by reallocating resources. Our evaluation underlines how this approach can improve SLO fulfillment under tight resource constraints, which motivates integrating it into existing CC platforms [11].

II. Methodology

In this section, we first describe the processing environment in which the scaling agents are embedded. Then, we present the design of the agents, focusing on how they interact with the environment for sensing, model training, and decision-making.

A. Processing Environment

Together, the data *provider*, processing *service*, and data *consumer* (cfr. Fig. 1) form a generalizable end-to-end streaming pipeline in which data streams from IoT devices are processed on nearby Edge devices and relayed to end users.

Service definition We define a *service* as $s = \langle f, C, M, Q \rangle$, which describes the processing function (f), the service configuration (C), and metrics (M) generated during execution. The quality of a processing service, such as video resolution, is adjusted through the service configuration. Q contains a list of SLOs that should be fulfilled during runtime, where a single SLO $q \in Q$ is defined by $q = \langle v, rel, t, w \rangle$. This implies that a variable (v) should either be higher or lower (rel) than a threshold (t). For example, a video processing service could aim for a satisfactory frame rate through $q = \langle fps, >, 25, 1.0 \rangle$, or save energy by limiting its allocated CPU cores through $q = \langle cores, <, 5, 0.5 \rangle$ These SLOs are ranked according to a weight (w), which will be used by the scaling agents to rank its objectives. Given a metric $m \in M$ that reflects variable v, the SLO fulfillment (ϕ) is calculated as in Eq. (1).

$$\phi(q, m) = \begin{cases} \frac{m}{t}, & \text{if } rel =' >' \\ 1 - \frac{m}{t}, & \text{if } rel =' <' \end{cases} \quad (1)$$

Metrics on a scale $m \in [0, \infty)$, such as *fps*, thus produce values that start from 0.0 (= 0% fulfillment) and can exceed 1.0. Values for *cores*, however, fall into $m \in [1, c_{phy}]$ with $t = c_{phy}$. Thus, allocating fewer cores provides higher SLO fulfillment.

Granular SLO fulfillment Most established Cloud platforms [3] employ binary logic to determine SLO fulfillment. In contrast, Eq. 1 provides a fuzzy ratio quantifying the extent of the SLO fulfillment. We favor this approach as it enables a more fine-granular elasticity control for autoscalers.

Service execution The service execution on an Edge device is wrapped in a Docker container. Thus, container resources can be vertically scaled by adjusting the number of allocated CPU cores. Notice, how Edge devices are constrained in numerous other ways (e.g., network or GPU), which can be scaled in future work. In the context of this paper, we define a device $d = \langle c_{phy} \rangle$ through its number of physical CPU cores.

B. Design of Scaling Agents

Within this processing environment, we develop a two-layer scaling solution based on a hierarchical setting of agents that ensures processing SLOs on Edge devices through multi-dimensional elasticity; Fig. 2 provides a high-level overview:
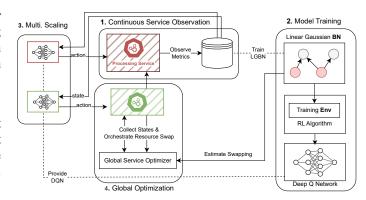

Fig. 2: High-level view of the three-step methodology; continuously observing service executions, training an inference model, and using it for multi-dimensional scaling

(1) Processing services are monitored to collect their states and respective SLO fulfillment; (2) each service is then extended with a Local Scaling Agent (LSA), which learns a scaling policy that optimizes SLO fulfillment; (3) if SLOs are violated, the LSA infers in which dimension to scale its service. When all resources are allocated, (4) resources can only be scaled through a mediator – called Global Service Optimizer (GSO) – which optimizes global SLO fulfillment by swapping resources. In the following, these steps are further elaborated:

1) Continuous Service Observation: During runtime, every service periodically logs a snapshot of its state to a local buffer, which will later be collected by the LSA. Recall, that a service's state includes its configuration, metrics, and SLOs. Depending on the use case, streaming services, e.g., video or audio processing, might log their state after processing one batch or frame. This creates a history of how a processing system was configured at a specific time, and to what degree (ϕ) its SLOs were fulfilled. Since processing services and scaling agents (i.e., LSAs and GSO) are executed on the same physical devices, the metrics can be accessed by all of them.

2) Model Training: The LSA has one central objective: fulfill the SLOs (Q) of its assigned service. To find an optimal scaling policy for this multi-objective problem, the LSA is trained through Deep Reinforcement Learning (DRL), where it is rewarded for actions that lead to satisfying service states. The sweet spot is to fulfill SLOs with $\phi_{opt} = 1.0$, i.e., without overprovisioning resources or sacrificing too much quality. Hence, the agent aims to minimize the difference (Δ) between ϕ and ϕ_{opt}, which is expressed by Eq. 2, where w is used to scale the reward of each SLO.

$$\Delta \leftarrow \sum_{q \in Q} |\phi_{opt} - \phi(q, m)| \times w_q \quad (2)$$

Training Environment Model-free RL algorithms often require tens of thousands of iterations to converge to a satisfying result – this is not compatible with our processing environment because the effects of scaling actions, including rewards, are only reflected with a delay – prolonging each training cycle considerably. To address this, we simulate state transitions

and respective rewards through a virtual training environment[1]. To create this environment, the LSAs use historical service metrics (M), however, cutting out periods of two seconds[2] after an action took place. The remaining metrics are used to train a Linear Gaussian Bayesian Network (LGBN) that expresses the relations between system variables. For a video processing task, the LGBN can express how processing *fps* depends on video resolution (*pixel*) and provisioned *cores*. Given these continuous variable relations, we can estimate the state transitions and rewards for hypothetical actions.

Training Actions The LSAs use this training environment to learn a scaling policy through a Deep Q Network (DQN), where the agent can take 5 different actions: (1) do nothing if its current state is satisfactory, (2|3) increase or decrease *pixel* by $\pm\delta_{pixel}$, or (4|5) adjust *cores* by $\pm\delta_{cores}$. The LSA should maintain these values in the range *pixel* $\in [p_{min}, p_{max}]$ and *cores* $\in [1, c_{free}]$, where c_{free} is the number of unclaimed cores, with $c_{free} \leq c_{phy}$. Notice, that δ_{pixel} and δ_{cores} are both constant factors, which we plan to substitute with continuous actions in future work. The LSA uses these actions to interact with the training environment; as DRL converges, the DQN estimates the SLO improvement given an action and state.

Model Retraining The LGBN is very dependent on the collected metrics, which is problematic because (1) no metrics are available at service start, and (2) historical metrics become outdated due to variable drifts. Hence, LGBN and DQN must be retrained periodically, which is done in the background with restricted resources. This retraining frequency can either be decreased gradually or coupled to SLO fulfillment.

3) Multi-dimensional Scaling: During runtime, the LSA uses its latest DQN to infer how to scale its attached service. Given the current service state, the DQN produces one of the five trained actions, which means that the LSA now interacts with the physical processing environment by either: scaling the service quality as part of the service configuration or scaling the resources by adjusting the container limitations.

4) Global Optimization: To improve the SLO fulfillment – and minimize Δ – the LSAs act greedy when they scale resources, which can be unfair to the other tenants on the Edge device. However, even if all resources have been allocated, it is possible to improve global (i.e., device-wide) SLO fulfillment. For instance, swapping a core from a service a to a service b that operates under tight SLO boundaries may improve the global SLO fulfillment because a's gain is higher than b's loss. Finding such operations is the responsibility of the GSO.

To decide whether it would be beneficial to swap a core between services, the GSO uses the available LGBNs. Given the states of both services (a, b), the GSO can again estimate what the expected state transition and consecutive reward would be when either swapping a core from $a \rightarrow b$, or $b \rightarrow a$. If the GSO estimates that one of these options improves global SLO fulfillment, it updates the container limits accordingly.

TABLE I: Constrained variables for the *CV* service

Var.	Description	Rel.	Weight	Impact
pixel	video streaming resolution	$> t$	0.8	*pixel* $\rightarrow$ *fps*
cores	container usable CPU cores	< 10	0.4	*cores* $\rightarrow$ *fps*
fps	processing throughput	$> t$	1.2	–

III. EXPERIMENTAL EVALUATION

To evaluate the presented scaling solution, we create an instance of the processing environment and develop a physical prototype of our scaling agents that we apply during two experimental scenarios. Namely, we evaluated: (1) How the LSA performs under tight resource constraints compared to established autoscalers, and (2) how the GSO can optimize the global SLO fulfillment after all resources were allocated.

A. Experimental Setup

To create a realistic stream processing scenario, we use OpenCV to continuously transform a video stream. The implementation of the processing service and the evaluation scenarios are publicly available on GitHub[3]. In the repository, you also find the description of how the Computer Vision (*CV*) service is containerized and how it is scaled during deployment.

Table I shows three types of SLOs (Q) for the *CV* service: to guarantee the quality of experience, the LSA should ensure high video resolution (*pixel*) and streaming framerate (*fps*). To save energy, the LSA also minimizes the number of allocated *cores*; however, this SLO can be traded off in favor of performance – hence it has a lower weight (w). Notice, how *fps* also has a higher weight than *pixel*[4]. For each SLO $q \in Q$, the thresholds (t) are specified in the following scenario.

B. Service Scaling under Resource Constraints

This scenario starts with an inactive LSA that does not yet possess any models (i.e., LGBN or DQN). The LSA awaits 30s of *CV* processing and then starts the first of five phases, in which: (1) the thresholds for *fps* and *pixel* are adjusted as shown in Tab. II; to restrict resources, we also adjust c_{phy}. The LSA then (2) trains the LGBN and DQN from all existing metrics and (3) operates 50s in the environment by autoscaling the *CV* service. This concludes the first phase; the next 4 phases are conducted the same way. To increase the stability of our results we repeat this entire scenario 5 times. We compare the LSA's effective SLO fulfillment with a baseline, similar to the Kubernetes VPA: initially, the VPA assigns *pixel* $= t$, which means it cannot sacrifice service quality. During runtime, the baseline VPA scales resources according to $\phi(fps)$, i.e., scaling *cores* $+ 1$ if $\phi(fps) < 1.0$ of *cores* $- 1$ if $\phi(fps) > 1.0$

As depicted in Fig. 3, the LSA initially performed slightly under the baseline, when its DQN was not yet accurate in the first two phases. In the subsequent rounds, however, the LSA outperformed the baseline VPA because it was able to trade off parts of the *pixel* SLO to fulfill the higher-weighted *fps* SLO. Notice, how the y-axis shows the cumulative SLO fulfillment (ϕ_Σ) with a maximum of $\phi_\Sigma \leq \sum_{q \in Q} w$; hence ≤ 2.4

---

[1] As described by OpenAI's Gymnasium environments
[2] We observed that roughly after this time scaling actions are reflected
[3] github.com/borissedlak/multiScaler
[4] During evaluation, these weights showed to reflect the intended tradeoff

TABLE II: SLOs as variable constraints for the *CV* service

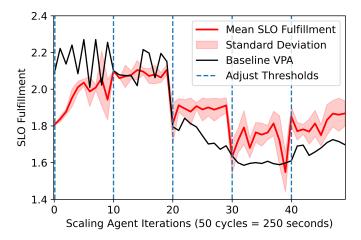| Var.	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
pixel	> 800	> 1000	> 1700	> 1900	> 1800
fps	> 33	> 33	> 35	> 35	> 34
max core	9	7	8	2	3

Fig. 3: SLO Fulfillment during runtime; every 10 iterations the SLO thresholds and available resources are changed

Due to page limitations, we cannot provide detailed results on the training overhead, still, we would like to comment that we limited the training to one core. Within these limited resources, LGBN training took roughly 1s, and roughly 10s for the DQN. Suppose we train the models less frequently than every 50s the overhead should not impact performance.

> **Implication**: With sufficient training, the LSA can be applied in resource restricted scenarios to choose between multi-dimensional elasticity strategies; this showed to improve SLO fulfillment further than a regular VPA

C. Global Optimization after Resource Exhaustion

The following scenario evaluates if the GSO can optimize global SLO fulfillment (= $\phi_{\Sigma,Alice} + \phi_{\Sigma,Bob}$) when no free resources can be allocated: We start two instances of CV – called Alice and Bob – which are supervised by two LSAs. Both LSA should ensure an SLO for *pixel* > 1300; additionally, we put a tight SLOs for Alice with *fps* > 30, whereas Bob only requires *fps* > 10. As soon as all resources are exhausted, the GSO takes action. Notice, how $\Delta = (\sum_{q \in Q} w) - \phi_{\Sigma}$.

As depicted in Fig 4, the GSO decides in iterations $i = 2$ and $i = 3$ to swap a core from Bob $\rightarrow$ Alice, which improves $\phi_{\Sigma,Bob}$, while showing no notable impact on $\phi_{\Sigma,Alice}$. Thus, increasing global SLO fulfillment. However, at $i = 5$, the same operation did not provide the expected result because it would harm Alice and not provide much benefit for Bob. This blunder, however, is resolved shortly after, when at $i = 7$ the GSO decided to swap back a core from Alice $\rightarrow$ Bob.

> **Implication**: In situations where no unclaimed resources are available, the GSO can improve the global SLO fulfillment by shifting resources between greedily acting autoscalers
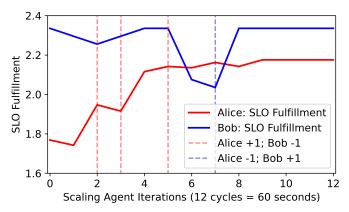

Fig. 4: SLO fulfillment of two services operating with resource contention; the GSO swaps resources to globally improve SLOs

IV. CONCLUSION

In this paper, we presented a multi-dimensional scaling solution that ensures processing SLOs on Edge devices. To improve SLO fulfillment in resource-constrained devices, we built a two-layer architecture with local, service-specific agents that scale either quality or resources; if all resources are allocated, a global mediator swaps resources between services to further improve SLO fulfillment. The innovation of our solution relies on the injection of domain-specific knowledge, i.e., the relation between system variables through an LGBN, and scaling processing services in multiple elasticity dimensions depending on the context. For future work, we have a clear research agenda for which we plan to improve the following aspects: (1) eliminate the DQN to operate directly on the LGBN to infer scaling actions, (2) produce continuous scaling actions for fine-grained control, (3) extend the architecture with more edge devices to support offloading, and (4) use scenarios with different, heterogeneous stream processing services.

REFERENCES

[1] M. Chui, M. Collins, and M. Patel, "The internet of things: Catching up to an accelerating opportunity," 2021.
[2] A. Morichetta, N. Spring, P. Raith, and S. Dustdar, "Intent-based management for the distributed computing continuum," in *IEEE SOSE*, 2023.
[3] S. Verma and A. Bala, "Auto-scaling techniques for IoT-based cloud applications: a review," *Cluster Computing*, vol. 24, no. 3, Sep. 2021.
[4] S. Dustdar, V. C. Pujol, and P. K. Donta, "On Distributed Computing Continuum Systems," *IEEE TKDE*, Apr. 2023.
[5] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of Elastic Processes," *Internet Computing, IEEE*, vol. 15, pp. 66–71, Nov. 2011.
[6] B. Sedlak, V. Casamayor Pujol, P. K. Donta, and S. Dustdar, "Controlling Data Gravity and Data Friction: From Metrics to Multidimensional Elasticity Strategies," in *2023 IEEE Services*, Jul. 2023.
[7] S. Laso, I. Murturi, P. Frangoudis, J. L. Herrera, J. M. Murillo, and S. Dustdar, "A Multidimensional Elasticity Framework for Adaptive Data Analytics Management in the Computing Continuum," Jan. 2025.
[8] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems," 2018.
[9] S. Horovitz and Y. Arian, "Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning," in *2018 FiCloud*, Aug. 2018.
[10] Y. Zhao and A. Uta, "Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions." IEEE Comp. Society, 2022.
[11] S. Nastic, P. Raith, A. Furutanpey, T. Pusztai, and S. Dustdar, "A serverless computing fabric for edge and cloud," in *IEEE CogMI*, 2022.