Performance Evaluation 167 (2025) 102452

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Trust your local scaler: A continuous, decentralized approach to autoscaling

Martin Straesser [a,*], Stefan Geissler [a], Stanislav Lange [b], Lukas Kilian Schumann [a], Tobias Hossfeld [a], Samuel Kounev [a]

[a] University of Würzburg, Sanderring 2, Würzburg, 97070, Germany
[b] Norwegian University of Science and Technology, Høgskoleringen 1, Trondheim, 7034, Norway

ARTICLE INFO

Keywords:
Autoscaling
Self-management
Discrete time analysis

ABSTRACT

Autoscaling is a critical component of modern cloud computing environments, improving flexibility, efficiency, and cost-effectiveness. Current approaches use centralized autoscalers that make decisions based on averaged monitoring data of managed service instances in fixed intervals. In this scheme, autoscalers are single points of failure, tightly coupled to monitoring systems, and limited in reaction times, making non-optimal scaling decisions costly. This paper presents an approach for continuous decentralized autoscaling, where decisions are made on a service instance level. By distributing scaling decisions of instances over time, autoscaling evolves into a quasi-continuous process, enabling great adaptability to different workload patterns. We analyze our approach on different abstraction levels, including a model-based, simulation-based, and real-world evaluation. Proof-of-concept experiments show that our approach is able to scale different applications deployed in containers and virtual machines in realistic environments, yielding better scaling performance compared to established baseline autoscalers, especially in scenarios with highly dynamic workloads.

1. Introduction

Cloud computing has rapidly advanced digital technologies and will be pivotal for future applications, such as widespread AI and virtual reality. In this context, autoscaling is vital in modern elastic cloud systems, ensuring optimal resource utilization and quality of service under varying loads. The shift to serverless computing underscores the need for efficient autoscaling, as this task falls into the hands of providers. The evolution from monolithic applications to microservices and fine-grained functions has transformed autoscaling mechanisms. Containers with short lifecycles, replacing long-lived virtual machines, present both challenges and opportunities. A core problem, for example, is that it is difficult to create explicit (white-box) models of applications due to the vast number of deployed services and the limited availability of source code. A promising opportunity is the higher elasticity potential enabled by low start-up times of containers and applications with low initialization times.

Autoscalers must essentially solve the problem of *which scaling action* to take *at which time*. Existing approaches range from threshold-based to control-theoretic and reinforcement learning mechanisms [1]. Most of these works use the paradigm that a central instance ("the autoscaler") makes scaling decisions at fixed intervals based on preprocessed monitoring data from multiple service instances. For example, the Kubernetes Horizontal Pod Autoscaler (HPA) scales a service based on average values for resource utilization in adjustable periods, e.g., 60 s. This paradigm is also used in established, often simple, autoscaling solutions offered by public cloud providers, such as AWS or Azure [2].

* Corresponding author.
E-mail address: martin.straesser@uni-wuerzburg.de (M. Straesser).

https://doi.org/10.1016/j.peva.2024.102452
Received 25 May 2024; Received in revised form 30 October 2024; Accepted 31 October 2024
Available online 8 November 2024

The described autoscaling paradigm has conceptual weaknesses. (1) The autoscaler acts as a single point of failure, impacting resource adjustment if it malfunctions or encounters incorrect or missing monitoring data [3]. (2) The scale-dependent nature of the autoscaling process poses challenges, making it uncertain whether it works equally well for varying system scales. Bottlenecks arise in the acquisition and processing of monitoring data, and scaling policies may be problematic when relying on absolute instance or load numbers. (3) Centralized autoscalers face the challenge of determining optimal decisions from a large action space, often requiring significant training data for learning-based approaches. Many approaches use configuration parameters like cooldown times, posing challenges for practitioners and hindering practical adoption [3–5]. (4) Reaction times to critical, unforeseen events are limited and suitable only for stable and predictable workloads. In summary, the autoscaler being a single point of failure, coupled with the need for optimal actions at fixed intervals, demands complex solutions lacking critical explainability for practical use.

In our proposed autoscaling approach, we diverge from the conventional paradigm by granting each service instance the authority to trigger autoscaling, enhancing failure resistance by eliminating a single point of failure. This decentralized, scale-invariant process relies on self-organizing instances rather than centralized monitoring. We view autoscaling as a continuous process where numerous minor decisions collectively contribute to global scaling behavior. Service instances, processing local monitoring data, are pivotal, feeding into scaling functions that determine probabilities for actions like adding an instance (UP), removing one (DOWN), or taking no action (HOLD). The decision is communicated to the cluster control plane via the execution layer, with scaling decisions occurring randomly over time. In the case of sufficiently large deployments, an asymptotically continuous autoscaling process ensues. Overloaded instances trigger the creation of new ones, and as the number of instances increases, overall utilization decreases, halting upscaling. Conversely, decreased load leads to underutilized instances, prompting the removal of replicas. In contrast to existing decentralized autoscaling approaches, our concept poses minimal assumptions on applications and runtime environments and features maximum flexibility through simple yet powerful configuration.

In this paper, we formally introduce and evaluate the outlined approach in three steps. First, we analyze formal properties using a discrete-time queueing model. Through simulations, we evaluate different configurations, scenarios, and key influencing factors. Using a proof-of-concept implementation, we show the applicability of our approach for scaling serverless functions in a realistic environment and compare it to established baseline autoscalers. Our results show the great adaptability of our approach to different workload profiles. Compared to the standard autoscaling mechanisms of Kubernetes and Knative, we improve various QoS metrics while simultaneously reducing costs. In addition, we showcase the applicability of our approach beyond CPU-bound container applications in a use case inspired by video streaming. Our source code, measurement data, and processing scripts are publicly available in our replication package [6,7]. In summary, the contributions of this paper are:

- The design of a novel decentralized autoscaling approach that enables an asymptotically continuous scaling process,
- a model- and simulation-based analysis for utilization-based autoscaling that shows the convergence, scalability, robustness, and configurability of the approach, and
- a proof of concept implementation evaluated with different test applications in realistic environments.

The remainder of this paper is structured as follows: Section 2 covers an extended background of this work, an outline of our idea, and a motivating example. Section 3 formally introduces our approach. Section 4 covers our model-based analysis, while Section 5 focuses on the simulative analysis. Section 6 introduces the implementation of our approach and real-world evaluation. Section 7 discusses the strengths and weaknesses of our approach, as well as threats to validity and extensions of this work. Section 8 gives an overview of related works, while conclusions are drawn in Section 9.

2. Background and motivation

In this section, we explain the preliminaries of our work, address open challenges in autoscaling, outline our idea, and demonstrate its effectiveness with a motivating example.

2.1. Open challenges in autoscaling

Cloud computing research has targeted autoscaling since its inception. However, further research is still needed, as many research autoscalers are not evaluated against real-world applications [3], and industrial solutions either rely on many user inputs or are simply threshold-based [2]. Furthermore, new lightweight runtimes like containers or unikernels bring new challenges and opportunities for autoscaling [4,8,9]. Likewise, new architectures for cloud applications, such as microservices or serverless functions, influence the workload patterns to be processed [10,11]. These developments motivate a new generation of autoscalers specialized in fine-grained, fast-starting applications rather than traditional heavy applications deployed on slow-starting virtual machines. In the following, we summarize some ongoing challenges in the area of autoscaling that recent surveys and industrial reports have identified.

Challenge C1: Reacting to structural or sudden workload changes. Structural changes in workload patterns are problematic for many autoscalers, especially those with explicit workload models. Proactive approaches are suitable for regular workloads but are often unfit to handle unexpected events [3,4]. Autoscalers based on machine learning (ML) experience a shift in the distribution of their input data, e.g., arrival rate patterns or resource usage profiles, compared to their training phase. This is why many learning-based approaches use periodic retraining by default [12,13]. Sudden workload changes that might outpace the autoscaler response and configuration issues have been identified as key challenges for modern autoscalers in industrial contexts [14]. In many cases, the reaction time to sudden workload changes is limited by design because a fixed scaling interval aligned with the scraping period of

the used monitoring tool (e.g., one minute [15]) is employed. This scaling interval is often also a hard-to-configure parameter for many practitioners [3–5]. A trade-off between sensitivity and stability of the autoscaling behavior has to be made [16]. Also, the start times of service instances have to be taken into account [5,8,14,17].

Challenge C2: Balancing Application-Agnostic and -Specific Autoscaling. Application-specific solutions strive to achieve the optimal scaling behavior for a single application. For example, novel approaches use graph neural networks [18,19] that take inter-service dependencies into account and thus achieve coordinated autoscaling for entire microservice architectures. Such approaches might not always be feasible, e.g., in the presence of continuous application updates, while application-agnostic and out-of-the-box usable autoscaling solutions are particularly suited for real-world applications [20]. In many use cases, e.g., in the context of short-lived serverless functions [21], only limited information is available for building application-specific models for autoscaling, as providers usually cannot access the source code and application-level metrics.

Challenge C3: Reducing interdependencies between monitoring and autoscaling. Monitoring data is critical for any autoscaler. Capturing the correct monitoring data reliably and efficiently is a key challenge in production systems [22]. The monitoring data source is always a single point of failure; if monitoring data is unavailable, no reliable autoscaling can occur. In heavily loaded systems, monitoring and tracing data can be missing, incomplete, or delayed. One reason for this is that service instances cannot submit statistics to a central data collector due to missing resources or heavy network traffic [3]. The impact of inaccurate data is significant, given that autoscaling plays a crucial role in alleviating the effects of heavy loads.

Furthermore, the compatibility between the deployed monitoring tool and the autoscaler must always be ensured. Changes in the system monitoring might require reconfiguration of the autoscaler [4]. Another factor is the monitoring overhead required to obtain the autoscaler's input data. For example, various approaches use latencies as scaling metrics, often measured by tracing or service meshes [1]. It has been shown that this kind of monitoring introduces considerable overhead [23], and multiple guidelines for practitioners recommend thoroughly weighing pros and cons before adoption [24,25]. However, not only does the monitoring unit pose a problem, but the expressiveness of scaling metrics is also always limited by their value range. For example, metrics like CPU or memory utilization are defined between 0 and 100 percent. In an overload (100% utilization) situation, we cannot deduce how many additional resources are needed. In this case, stepwise adjustments find the optimal supply, and overall reaction time is limited by the configured fixed scaling intervals (see C1).

Challenge C4: Increasing trust and explainability in novel autoscaling solutions. Besides its functional properties, trust in the autoscaler is of great importance for practical adoption since customer experience, as well as operational costs, are determined by autoscaling [26]. Explainability is an essential factor in enabling trust. Explainability includes both the comprehensibility of scaling decisions and intuitive adjustability of configuration parameters so that the effect of a configuration change is foreseeable and explainable [3]. While simple, threshold-based autoscalers fulfill those criteria, they are especially a showstopper for the practical adoption of autoscalers based on deep learning. A recent industrial blog post characterizes the issue of trust as one of the core challenges for using AI for autoscaling [27]. Recent research explores the use of explainable AI for autoscaling to mitigate these issues [28].

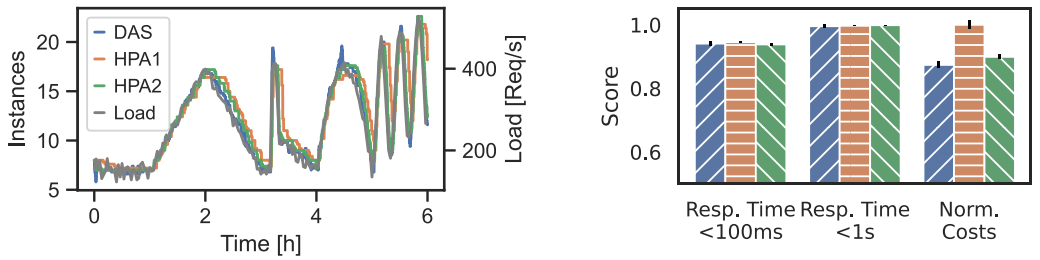
2.2. The idea of continuous decentralized autoscaling

To overcome the above challenges of conventional autoscalers, we propose an approach with two important properties: (i) Decentrality: In our approach, individual service instances make independent scaling decisions. (ii) Continuity: Instead of viewing autoscaling as a problem where the optimal number of instances must be decided at discrete points in time, we treat autoscaling as a continuous process that converges against an optimal system state. In the following, we provide a rough description of the method while introducing details and formal notation in Section 3.

Our approach assumes that service instances can collect and process their own monitoring data, which is also the first step in a scaling cycle. In every iteration, an instance chooses one of the three action alternatives: add a replica (UP), remove a replica (DOWN), or no action (HOLD). The decision is made based on the measured monitoring values. We generally distinguish three ranges: a downscaling range, an upscaling range, and a tolerance zone. For example, for CPU utilization limited between 0 and 100%, we could define the ranges as follows: upscaling range 80%–100%, downscaling range 0%–50%, and tolerance zone 50%–80%. The instance always decides to HOLD if the measured value is in the tolerance zone. If the measured value is in the upscaling (downscaling) range, the system evaluates whether UP (DOWN) should be executed as an action. In the following, we limit our explanation to the upscaling range, but the statements apply analogously to downscaling.

One way would be always to scale up if the measured value is in the upscaling range. This corresponds to a classical threshold-based approach. However, this approach has disadvantages in our decentralized decision-making (see Section 5.3). Hence, we consider a more flexible approach based on *scaling functions*. The upscaling function receives a measured monitoring value and returns the probability for an UP action. An intuitive choice would be a monotonically increasing function, returning greater values for inputs farther from the tolerance area. In our CPU utilization example, we could configure that at 81% utilization, the UP probability should be 10%, while at 100% utilization, an UP action should always be performed. The choice of the scaling function and the size of the tolerance, up-, and downscaling ranges allow a more flexible and powerful configuration of the scaling behavior. In the implementation of the approach, the execution of the decision is outsourced to an external component, the execution layer (see Section 6).

Before starting a new cycle, an instance waits some time. This waiting time is drawn from a previously defined distribution. Because all currently available instances make decisions at different times, there is a high frequency of scaling decisions. Hence,



(a) Deployed instances and workload.

(b) QoS and cost metrics.

Fig. 1. Motivating example with a decentralized autoscaler and two HPA configurations.

if many service instances are active, our approach converges into a quasi-continuous process [29,30]. Deterministic decision times can result in many up- or downscaling decisions within short intervals, resulting in bad scaling behavior.

Our idea addresses the challenges identified in Section 2.1 as follows: By using a quasi-continuous scaling process, we achieve low reaction times to workload changes (C1), especially compared to autoscalers with fixed scaling intervals. In addition, our approach provides a good balance between application-specific and -agnostic autoscaling (C2). On the one hand, application-specific characteristics can be considered by implementing special scaling functions. On the other hand, our approach can be used similarly to conventional threshold-based autoscalers by choosing simple scaling functions. Our approach does not depend on centralized monitoring and thus eliminates single points of failure (C3). Last, the scaling functions as key tuning parameters provide a flexible, robust, and explainable configuration of the autoscaler (C4). With our decentralized design paradigm, we differ conceptually from many state-of-the-art autoscaling solutions. Our realization of a continuous autoscaling process and its flexible configuration through scaling functions separate our work from existing decentralized autoscalers.

2.3. Motivating example

In this section, we demonstrate the potential of our idea in a motivating experiment.

Experiment Setup, Application and Workload. Our experiment setup is a Google Cloud cluster with five worker nodes of type e2-medium and Kubernetes v1.26 with `containerd` as the container runtime. An HTTP load generator[1] is deployed on an external VM in the same compute zone. Our test app is a Python function [6] with an integrated Nginx gateway that repeatedly computes SHA256 hashes. The average service time is about 2 ms. The container resources are limited to 0.1 CPU cores. The test workload has a total length of six hours and combines the six workload patterns proposed in a recent autoscaler evaluation methodology [31], overlaid with uniformly distributed noise.

Autoscaler Setup. We evaluate a decentralized autoscaler (DAS) and the Kubernetes Horizontal Pod Autoscaler (HPA) [32] with different configurations. All scalers make their decisions based on CPU utilization. The HPA receives the metrics from the Kubernetes metrics server. DAS is placed as a binary in the container, runs as a background process, and calculates the CPU utilization directly in the container. In preliminary experiments, we tested 27 HPA configurations, two of which we selected for display in this paper. HPA1 acts in intervals of 60 s and uses the default values of all other HPA parameters. HPA2 is the best configuration (best QoS-to-cost ratio) for this workload in our parameter study. It is an aggressive configuration acting in intervals of 15 s with no downscaling stabilization. The waiting time for our decentralized autoscaler is drawn from a uniform distribution between 30 and 90 s so that the expected value corresponds to the sync interval of HPA1. A target CPU utilization of 75% was chosen for HPA1 and HPA2. For the decentralized autoscaler, we choose a tolerance zone between 70% and 80%. The upscaling probability increases exponentially between 80% and 95% utilization; upscaling always takes place if the utilization exceeds 95%. The downscaling probability increases linearly between 70% and 60%, limited to a maximum of 30%. We chose these scaling functions as they were recommended in our parameter study (see Section 5.3). Refer to Appendix A for formal definitions of the scaling functions and Appendix B for a complete parameter description.

Results. Fig. 1(a) shows the workload and the number of deployed instances over time for all autoscalers averaged over five repeated runs. All autoscalers generally follow the workload curve and adapt to the changing arrival rate. Fig. 1(b) shows autoscaler evaluation metrics. To assess QoS, we use the proportion of requests with a maximum response time of 100 ms or 1 s (corresponding to different SLO targets). Throughout this paper, we use a generic cost metric C [33], which is the area under the curves shown in Fig. 1(a). Hence, it is defined as the definite integral of the running instances over time $I(t)$ limited by the experiment start $t = 0$ and the end of the experiment t_e:

$$C = \int_0^{t_e} I(t)\, dt$$

[1] https://github.com/joakimkistowski/HTTP-Load-Generator.

For better display, we normalized the cost values in Fig. 1(b) by dividing all values by the mean costs of HPA1. The results clearly show that the DAS performs better than HPA1 and HPA2. It achieves the same QoS level with costs reduced by 12.7% compared to HPA1 and by 2.8% compared to the optimized configuration HPA2. This is mainly due to the increased scaling speed and elasticity. Fig. 1(a) shows the lower reaction times to load changes of DAS compared to the Kubernetes autoscalers. The experiment shows the potential of continuous decentralized autoscaling. Especially in complex scenarios with dynamically increasing or decreasing arrivals, the strengths of our concepts are revealed, as faster reactions are possible through the higher frequency of scaling decisions.

3. Approach

This section first describes our system model and then formally introduces continuous decentralized autoscaling.

3.1. System model

This section introduces our system model and the abstractions relevant to our approach. In this paper, we consider horizontal scaling only, meaning that scaling takes place by adjusting the number of *service instances*. We assume that scaling actions can be executed with a reasonable delay and high frequency. This assumption holds especially for modern containerized applications but not for environments where the addition or removal of instances is costly.

Service instances process requests from external sources. A *system arrival process* with an arbitrary inter-arrival time distribution with arrival rate λ dictates the mean number of incoming requests per time unit. In this work, we make no further assumptions or restrictions regarding the sources of the requests and assume that each request originates from one of an infinite number of sources. In practice, requests can be issued by users or external services. The requests first hit a *load balancer*, which distributes the arrival stream to k ($k \in \mathbb{N}$) service instances. Similar to previous works, we assume that the load balancer assigns incoming requests immediately to a service instance and has no centralized queue [34]. We do not further model the behavior of the load balancer, as commonly used commercial load balancers in cloud environments, e.g., Google Cloud Load Balancers (also used in this work), are closed-source.

The load balancing process splits the system arrival process into k *instance arrival processes* with arrival rates λ_i ($i \in [1; k]$). Service instances have queues with unlimited maximum length. In this paper, we assume that all instances are replicas, i.e., they are stateless and their properties, such as service time distribution, are identical. This is justified by real-world deployments of modern cloud applications, such as microservices and serverless functions. Furthermore, we assume there is no interference between the service instances. An infinite number of instances can be deployed without interfering with each other. In public cloud deployments, this is justified as (i) instances are usually resource-constrained or mechanisms are used to ensure performance isolation [35] and (ii) the amount of available computing resources is virtually infinite. We do not use an explicit model for the cloud environment, e.g., physical servers, resources, and networks. We do not consider performance overheads introduced by these entities (e.g., network delays or the processing time at the load balancer). We do this since we cannot control these factors in our public cloud experiment setup.

As a result, the key performance indicator of our system, the response time, consists of two components: the service time and the waiting time of a request in the queue of a service instance. We assume that once a request starts being processed, this request does not join a queue again. This includes the assumptions that there is no parallel processing of requests and that a resource is always fully loaded or idle. We assume that a request has no external dependencies, i.e., synchronous calls to other services for whose response the request must wait. Along with making no assumptions about request sources, we can limit ourselves in the analysis to single instance types, not networks of service instances. External dependencies may be considered indirectly via the choice of the service time distribution.

Autoscaling aims to find the optimal number of instances k^* for a concrete system arrival process with rate λ. k^* is the smallest number of instances for which all service level objectives (SLOs) are fulfilled. In our work, we formulate SLOs based on response times. If the instance arrival rates λ_i increase in our system, the waiting times and response times r increase. If r exceeds critical values, SLOs are violated. Adding new instances makes the instance arrival rates λ_i smaller, meaning SLOs can be met again.

3.2. Continuous decentralized autoscaling

In conventional centralized autoscaling systems, an autoscaler processes data collected from all service instances by a monitoring system and calculates the optimal number of instances. For this calculation, the autoscaler uses values averaged over the service instances. In contrast, in our decentralized approach, decisions are made at the instance level based on instance-level monitoring data. This assumes that instances can collect and process their own monitoring data. This is a valid assumption for many runtimes of modern cloud applications, such as containers, unikernels, VMs, and microVMs. Resource metrics such as CPU or memory utilization can be determined directly, e.g., via *cgroup* features in containers. Application-specific metrics, e.g., thread counts, are usually exposed to the outside via dedicated metric endpoints. Instead of querying these by a monitoring tool like Prometheus,[2] we can also process these values locally. By making the decisions at the instance level, we do not rely on a central monitoring system.

[2] https://prometheus.io/.

In our approach, each service instance executes the algorithm shown in Algorithm 1 during its runtime. A scaling cycle starts with the collection of the current monitoring data m_t. We consider $m_t \in M$ as a vector containing the current values of the *scaling metrics*, i.e., the metrics that should influence the autoscaling behavior. M is the set of all possible monitoring data. In each cycle, an instance must decide between three scaling decision alternatives. *UP* causes a new instance to be started. *DOWN* causes an instance to be switched off. *HOLD* does not change the number of service instances. Note that the service instances do not know the total number of instances deployed in the system. This would require global knowledge and would be against our decentralized design paradigm.

Algorithm 1 Algorithm for Decentralized Autoscaling

Input: $D : M \to [0;1]$, $U : M \to [0;1]$, W: Probability distribution in interval $[w_{min}; w_{max}]$

```
1:  while instance is running do
2:      Collect recent monitoring data into m_t
3:      Draw sample r from a uniform distribution in [0; 1]
4:      P(UP) = U(m_t)
5:      if P(UP) > 0 and r < P(UP) then
6:          decision = UP
7:      else
8:          P(DOWN) = D(m_t)
9:          if P(DOWN) > 0 and r < P(DOWN) then decision = DOWN
10:         else decision = HOLD
11:     end if
12:     Submit decision to execution layer
13:     Draw sample w from W and wait for time w
14: end while
```

Which action is executed is determined by the *scaling functions* U and D. In the following, we use the term *scaling policy* S to refer to a combination of a concrete U and D. Both functions are defined to return a probability that a scaling decision is made. U determines the probability for UP, D the probability for DOWN. Operators can flexibly define the desired autoscaling behavior by adjusting these scaling functions. By using probabilistic behavior, we enlarge the design space for scaling policies. As a special case of U and D, conventional threshold-based policies that always return either the probability 0 or 1 can also be implemented. In our work, we also assume that the decision is always based on the current measured values m_t. As a conceivable alternative, the functions could also be implemented to use the monitoring values of past cycles (see Section 7 for further extensions). What action should be executed is ultimately determined by a random number $r \in [0;1]$ drawn from a uniform distribution. We assume that for all possible monitoring values, only either the upscaling or downscaling probability is greater than 0:

$$\forall\, m_t \in M : U(m_t) > 0 \Rightarrow D(m_t) = 0 \wedge D(m_t) > 0 \Rightarrow U(m_t) = 0 \tag{1}$$

This means that no vector of monitoring data can cause both a UP and a DOWN action. Lines 4 to 11 of Algorithm 1 show the decision-making of the instance based on the evaluation of the functions D and U as well as the comparison with the random number r. As a consequence of constraint (1), it does not matter whether up- or downscaling is evaluated first. Dependent on the choice of U and D, a subset of M may be created where both the upscaling and downscaling probability is 0. This corresponds to a desired state where no autoscaling should be performed. In the following, we refer to this subset as the tolerance area τ, formally defined as:

$$\tau = \{m_t \in M : U(m_t) = 0 \wedge D(m_t) = 0\} \tag{2}$$

The instance submits the decision to the *execution layer*. The execution layer is an external component that realizes the scaling decision. In real-world systems, cloud applications are often deployed using container orchestration frameworks (e.g., Kubernetes), which start new or terminate running instances. The execution layer is an interface that passes scaling commands to the cluster control plane. We outsource the execution of the scaling decision to an external component for three reasons. First, we avoid dependencies of the instance scaling logic on particular deployment frameworks. Second, we can also use logging at the execution layer, increasing the explainability of our approach. Third, we can enforce global rules required in production systems, such as maximum or minimum instance numbers. The only rule used in this work is that at least one service instance must be deployed at any time, meaning that every decision made by the service instances is guaranteed to be executed as long as the resulting replica number is greater than zero. However, additional logic might be implemented at the execution layer in future work. Further details on the execution layer are discussed in Section 6, and its extensions in Section 7.

The last step is determining the amount of time to wait until a new scaling cycle begins. We use a random number w, drawn from a configured distribution W. It is assumed that W yields a minimum $w_{min} > 0$ and a maximum waiting time w_{max}. The advantage of using a probabilistic approach here is that the decisions of the individual scaling instances are distributed over time. The more instances are deployed, the higher the scaling frequency. This means that instead of interval-based autoscaling, we convert autoscaling into a continuous process. As a special case of W, a deterministic distribution can also be used, which, however, can lead to the fact that if several instances start simultaneously, they also make their scaling decisions simultaneously.

This concept can be applied to arbitrary scaling metrics and scaling functions. In the following, we focus on CPU-based scaling and choose the utilization $\rho \in [0; 1]$ as our scaling metric. For U and D, we consider monotonic functions, as they are the natural choice in this case. This means that $U(\rho_1) \leq U(\rho_2)$ if $\rho_1 \leq \rho_2$ and $D(\rho_1) \geq D(\rho_2)$ if $\rho_1 \leq \rho_2$. In this case, in order to satisfy constraint 1 there must be two values τ_{min} and τ_{max} that fulfill the following three conditions: (a) $\rho > \tau_{max} \iff U(\rho) > 0$, (b) $\rho < \tau_{min} \iff D(\rho) > 0$, and (c) $\tau_{min} \leq \tau_{max}$. In the case that $\tau_{min} < \tau_{max}$, the tolerance area τ is the interval $[\tau_{min}; \tau_{max}]$. Otherwise, τ is a single value. A valid example scaling policy with thresholds would be to scale down if ρ is less than 0.3 and scale up if ρ is greater than 0.7. The tolerance area τ and the desired system state would hence be a utilization between 0.3 and 0.7.

4. Discrete-time queueing model for utilization-based scaling

Based on the system introduced in the previous section, we now introduce computations for the following KPIs of the decentralized autoscaler.

1. What is the distribution and expected time to the next scaling decision? (Reaction time)
2. What is the distribution of time between a change in load until the autoscaler has restored stable operation? (Convergence speed)
3. What is the distribution of the number of instances once the system has reached stability? (Steady state)

In the scope of this section, to distinguish between random variables (RVs), distributions, and distribution functions, we use the following convention: uppercase letters such as A denote RVs, their distribution is represented by $a(x) = \mathrm{P}(A = x),\ x \in [0, \infty)$ and the corresponding distribution function is defined as $A(x) = \mathrm{P}(A \leq x) = \sum_{i=-\infty}^{x} a(i),\ x \in [0, \infty)$. We start by computing the distribution of the time between two scaling decisions based on the superposition of multiple, in this case identical, stochastic processes [36]. Here, the events of n independent (non-synchronized) processes described by random variables $X_1, X_2, \ldots, X_n$ are merged into a superimposed process with inter-arrival times $\bar{X}_n$. The time between occurrences in the superimposed process can be computed using the minimum of the recurrence times R_i of the original processes.

$$\bar{X}_n = 1 - \min(R_1, R_2, \ldots, R_n) \text{ with } P(R_i = x) = r_i(x) = \frac{1 - X_i(x)}{E[X_i]}. \tag{3}$$

The minima $X^n_{min} = \min(X_1, \ldots, X_n)$ of n RVs $X_1, \ldots, X_n$ have the cumulative distribution function

$$X^n_{min}(x) = 1 - \prod_{i=1}^{n}(1 - X_i(x)). \tag{4}$$

Based on this superposition of stochastic processes, we are consequently able to compute the distribution of the time between two scaling decisions for a given number of running instances and develop a discrete-time queueing model that enables calculating both the duration until the system converges towards a stable state and the distribution of the expected number of instances based on easily obtainable input parameters.

Fig. 2 shows the development of this model over time and highlights critical aspects and parameters of the model. To this end, we abstract the whole system as a single entity and describe the system via an embedded Markov chain with embedding times right before each scaling decision that can subsequently be solved by means of fixed-point iteration by tracking the amount of unfinished work U_i in the system as well as the current number of processing instances K_i iteratively at each subsequent embedding time. In addition, we denote the inter-arrival time of requests to the system as A and the time between two scaling decisions of a single instance as W. The unfinished work in the system is continuously being processed by all available instances. Hence, the rate (slope) with which the unfinished work ($U(t)$ in Fig. 2) declines depends on the current scale of the system and may change at every decision time, depending on whether the number of instances increases, decreases, or remains unchanged. Fig. 2 summarizes all input and output parameters for the model.

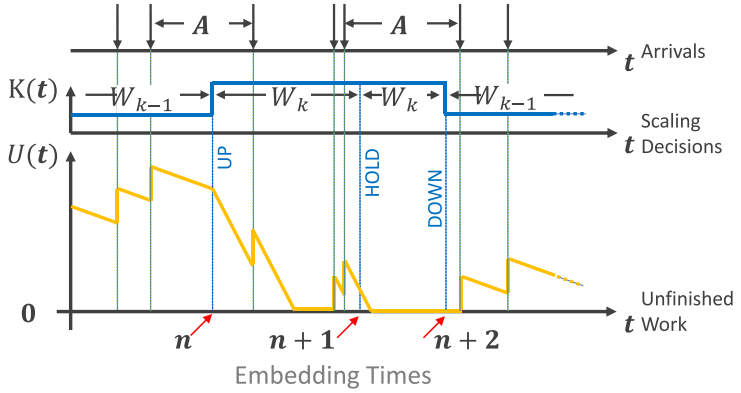
To realize the aforementioned fixed-point iteration, we define the following transition functions to compute the system state (U_i, K_i) at embedding time i based on their values at embedding time $i - 1$. Using these transition functions, it is then possible to iteratively compute the system state at the next embedding time based on the initial system state at embedding time 0, described by K_0 and U_0.

In the first step, Eq. (5) describes the computation of the change in unfinished work $\bar{U}_i$ since the previous embedding time $i - 1$ and subsequently the amount of unfinished work U_i at embedding time i.

$$\begin{aligned} \bar{u}_i(x) &= \sum_{k=-\infty}^{+\infty} (u_{A,k}(x) * u_{B,k}(-x)) \cdot k_{i-1}(k) \\ u_i(x) &= \pi_0\left(\bar{u}_i(x) * u_{i-1}(x)\right) \end{aligned} \tag{5}$$

Thereby, $*$ refers to the discrete convolution operation and π_0 represents the left-sided sweep operator which sums the probability mass of negative unfinished work in the system and accumulates it at $u(0)$.

$$\pi_0(u(x)) = \begin{cases} u(x) & x > 0 \\ \sum_{i=-\infty}^{0} u(i) & x = 0 \\ 0 & x < 0 \end{cases} \tag{6}$$



Param.	Description
λ	Arrival rate of requests at the system
μ	Service rate of a single instance
$W, w(x)$	Inter-decision time of a single instance
$U_i, u_i(x)$	Unfinished work at embedding time i
$K_i, k_i(x)$	No. of instances at embedding time i

Param.	Description
$\bar{\rho}_i$	Average system load in interval between embedding times $i-1$ and i
ρ_i	60sec average system load at embedding time i
$S_i, s_i(x)$	Scaling distribution at embedding time i

Fig. 2. Evolution of the discrete-time queueing model and model parameters.

Furthermore, the random variables $U_{A,k}$ and $U_{B,k}$, and their respective distributions $u_{A,k}(x)$ and $u_{B,k}(x)$, describe the amount of arriving and processed work since the last embedding time. In particular, those quantities are proportional to the time between consecutive decisions, and hence the number of active service instances k, as well as the arrival and service rate. Specifically, the amount of arriving and processed work is equal to the number of arrivals and departures occurring within each inter-decision interval. The distributions of these quantities are computed according to [37].

Based on the amount of unfinished work in the system, we can then compute the system load in the interval since the previous embedding time for specific realizations of K_i and U_i as $\bar{\rho}_i(k,u)$ and as shown in Eq. (7).

$$\bar{\rho}_i(k,u) = \min\left(1, \frac{1}{k\cdot\mu}\cdot\left(\frac{u}{\mathrm{E}[W_k]}+\lambda\right)\right) \tag{7}$$

Thereby, W_k denotes the $k-$fold superposition of the inter-decision time of a single instance W. Subsequently, the overall observed load at embedding time i can be computed as

$$\bar{\rho}_i = \sum_{k=-\infty}^{+\infty}\sum_{u=-\infty}^{+\infty}\bar{\rho}_i(k,u)\cdot k_{i-1}(k)\cdot u_{i-1}(u). \tag{8}$$

In accordance with the technical system [38], we further model an exponential moving average approach with exponentially decaying weights to compute the mean load over a specified historical interval. Similar to the technical system, we use the last 60 s to model the average load of the system and compute the average load for specific realizations of K_i and U_i as well as the total observed, average load of the system as described in Eq. (9).

$$\begin{aligned}
\rho_i(k,u) &= \frac{\bar{\rho}_i(k,u)+\sum_{j=1}^{t}\bar{\rho}_{i-j}\cdot e^{-j}}{\sum_{j=0}^{t}e^{-j}} \\
&\text{with } t: \sum_{j=0}^{t-1}\mathrm{E}[W^{i-j}] \le 60 \wedge \sum_{j=0}^{t}\mathrm{E}[W^{i-j}] > 60 \\
w^i(x) &= \sum_{k=-\infty}^{+\infty} w_k(x)\cdot k_i(k) \\
\rho_i &= \sum_{k=-\infty}^{+\infty}\sum_{u=-\infty}^{+\infty}\rho_i(k,u)\cdot k_{i-1}(k)\cdot u_{i-1}(u)
\end{aligned} \tag{9}$$

Thereby, W^i represents the inter-decision time at embedding time i.

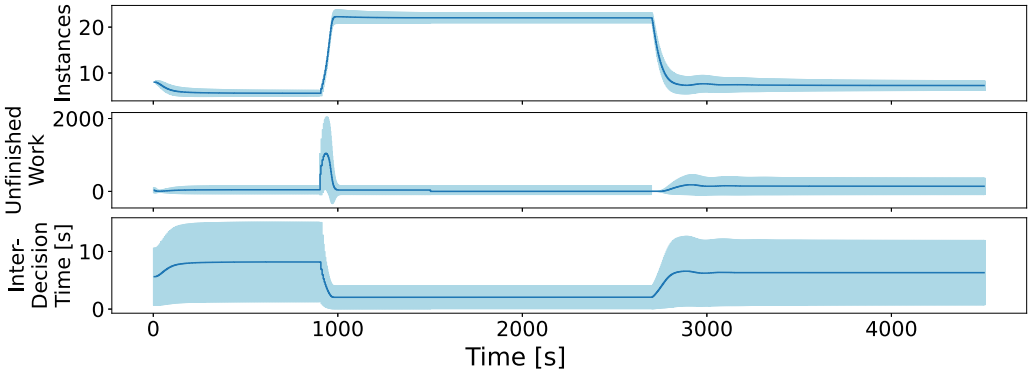


Fig. 3. The evolution of state variables of the model in a scenario with changing load.

Using the system load as its input, we can now apply the previously introduced scaling policy to determine the change in the number of running instances for a specific realization of K_i.

$$(S_i(k))(x) = \sum_{u=-\infty}^{+\infty} (S(\rho_i(k,u)))(x) \cdot u_{i-1}(u) \tag{10}$$

Here, $S(x)$ represents the system-specific scaling policy that combines the up- and downscaling functions from Section 3.2 and maps the system load to a discrete distribution of scaling decision values in $\{-1, 0, 1\}$ that indicate the probabilities to scale down, hold, or scale up, respectively. In a final step, we can compute the number of running instances at embedding time i based on the system state at the previous embedding time, as well as the scaling decisions resulting from this system state.

$$k_i(x) = \sum_{k=-\infty}^{+\infty} \pi_0(\delta_k(x) * (S_i(k))(x)) \cdot k_{i-1}(k) \tag{11}$$

Here, $\delta_k(x)$ denotes a deterministic distribution that has its entire probability mass at k:

$$\delta_k(x) = \begin{cases} 1 & x = k \\ 0 & x \neq k. \end{cases} \tag{12}$$

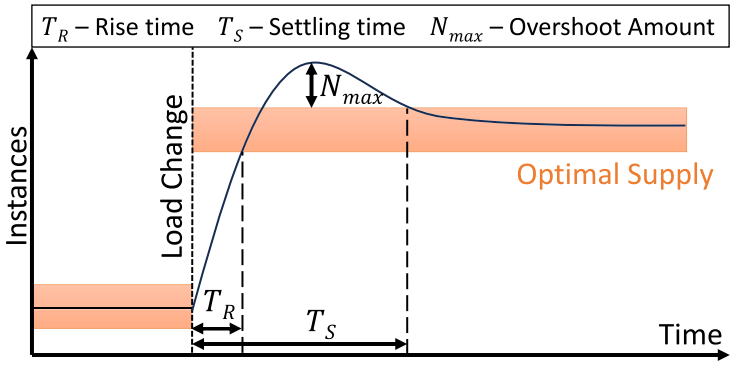
Using Eqs. (3) to (12), we can iteratively compute the system state until the model converges at embedding time j and $k_j(x) = k_{j+1}(x)$ holds for all x and embedding times $i > j$ to answer the previously posed questions. Fig. 3 shows the evolution of essential state variables of the model with expected value and standard deviation in an experiment with a load increase and decrease. The system is initialized with eight instances. Initially, the system is stressed with a load of 20 requests per second (rps); after 900 s, the load jumps to 80 rps before dropping back to 20 rps after 2700 s. The scaling policy has a tolerance area between 60% and 80% utilization; the probabilities for up- and downscaling increase exponentially with increasing distance to this tolerance area. Our model's embedding times are always right before the next scaling decision. Consequently, the time interval between two embedding times is determined by the distribution of the inter-decision times. The projection of the results of the fixed-point iteration onto the time axis is done by summing up the distribution of the inter-decision time and taking the expected value in each step. We have opted for this type of presentation as it is easier to interpret. The complete model results per iteration can be found in our replication package.

We see that when the load increases, the unfinished work in the system rises to over 1000 unanswered requests on average. This leads to increased instance utilization, and our scaling policy triggers adding new instances. As the number of instances increases, the unfinished work decreases, and the system settles into a steady state in which the distribution of instance utilization is entirely within the tolerance range. The steady state is also reached after downscaling. Here, the unfinished work drops significantly in average and variance during the load decline, which leads to a drop in instance utilization and, thus, to downscaling. For this configuration, the model estimates a low probability that the system overshoots downwards, i.e., removes too many instances. This leads to a counter-reaction explaining the slight increase in the number of instances on average. The inter-decision times behave inversely to the number of instances; if more instances are deployed, the inter-decision time is shorter. In this example, with about 22 instances, a scaling decision occurs about every 2 s, while the waiting time of each individual instance is equally distributed between 30 and 60 s. We provide the code to recreate this experiment in our replication package [6]. In Section 6.2, we compare these model results to both measurements in the real system and our simulation model and provide a detailed description of other experiment parameters.

5. Simulation results

We have developed a discrete event simulation using the *simmer*[3] framework to evaluate our approach further. The simulation implements the model described in Section 3.1. Our simulation supports both trace-based and model-based arrival streams. In

[3] https://r-simmer.org/.



Scale Factor s	T_R [s]	T_S [s]	N_{max}/s
1	241.6 ± 24.8	268.3 ± 68.2	1.00 ± 1.26
5	246.5 ± 9.9	322.9 ± 26.8	0.64 ± 0.23
10	238.9 ± 10.1	300.0 ± 61.7	0.36 ± 0.42
50	237.8 ± 7.5	311.4 ± 10.2	0.27 ± 0.03
100	240.1 ± 6.1	310.2 ± 7.9	0.21 ± 0.05

Fig. 4. Metrics for step response (top) and values for different system scales (bottom).

addition to our algorithm parameters W, U, and D, the simulation allows us to test different service time distributions, start time distributions, and load balancer policies. The simulation applies the same instance selection policy as Kubernetes to determine which instance should be stopped in case of downscaling (by default: last in, first out). As output, the tool displays various metrics such as response times, instances over time, instance utilization, and more. The tool is available at [6]. In the following, we state and answer three research questions that are related to the configuration, scalability, and robustness of our approach with the help of simulation:

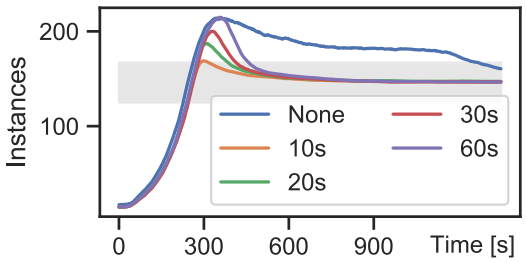
1. What role does the system scale play, i.e., the number of instances and the total number of incoming requests? (Scalability)
2. How do typical influencing factors from real environments affect our approach? (Robustness)
3. What is the influence of the waiting time distribution W and the scaling functions U and D on the scaling behavior? (Configuration)

To keep the description of the main results in the following sections compact, we report all input parameters of the simulation scenarios in Appendix D.

5.1. Approach scalability

In this section, we analyze the role of the system scale, i.e., the number of incoming requests and active instances in the system. We aim to analyze whether our approach can be used with a fixed configuration (consisting of W, U, and D) for different system scales. For this, we use a well-known methodology from control theory. As input, we use a step function that increases the load by a factor of 10 after 1350 s. Our evaluation metrics (rise time, overshoot amount, and settling time) follow definitions from control theory (see Appendix F) and are visualized in Fig. 4. In the reference case (scale $s = 1$), the load increases from 3 rps to 30 rps. In all other cases, the arrival rate before and after the jump is multiplied by the scale factor. This means that in the case $s = 100$, the load increases from 300 rps to 30,000 rps. We use a uniform distribution between 30 and 90 s for W. The desired range for CPU utilization is between 60% and 80%. The service time follows an exponential distribution with an expected value of 200 ms. The scenario was repeated with five different random seeds.

Fig. 4 shows our evaluation metrics and their standard deviations for different system scales. Note that the rise and settling times are nearly equal for any scale. This is because a higher system scale increases the load jump in absolute numbers but concurrently increases the potential elasticity and number of scaling decisions due to the higher number of deployed instances. In the case $s = 100$, more than 1000 service instances are deployed, causing inter-decision times to be lower than 100 ms on average. The normalized overshoot gets less with increasing system scale because of the increased absolute size of the tolerance area. The results show that the policy provides consistent and predictable performance for our approach at different system scales. We achieve this even though there is no coordination among instances. Especially in large-scale systems, our advantage is that we do not rely on a scalable monitoring system, where data has to be collected and processed from many instances. In our approach, the monitoring overhead is stable at every scale because the instances monitor themselves.



Timeout	T_S [s]	N_{max}
10s	295.6 ± 41.1	3.7 ± 4.1
20s	381.2 ± 16.5	23.3 ± 4.8
30s	411.9 ± 17.4	37.2 ± 4.5
60s	455.3 ± 15.8	51.6 ± 5.1
None	1261.5 ± 66.0	52.9 ± 4.9

Fig. 5. Deployed instances (left) and step response metrics (right) for different timeouts.

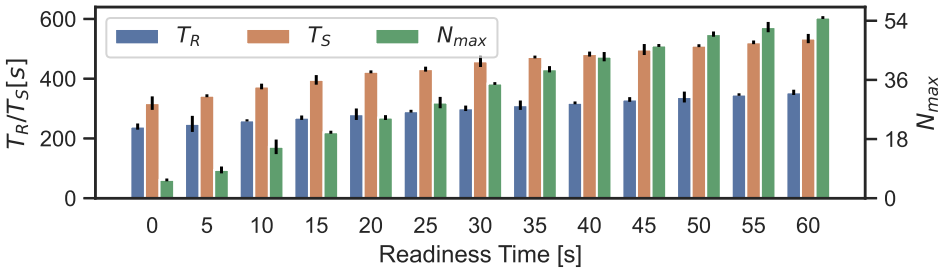


Fig. 6. Step response metrics for different readiness times.

5.2. Further influencing factors

In this section, we analyze how our approach is influenced by other factors that occur in production cloud environments but that we have not considered in the model-based analysis. These factors are external factors that cannot be controlled by the autoscaler but still influence its performance.

Request timeouts. Timeouts are a fixed part of many network protocols, such as TCP or HTTP. Clients abort requests if they are not acknowledged or answered within a specific time. Different solutions exist on the server side to detect stale requests, e.g., by checking the underlying TCP connection. In the following, we assume that the processing time of a timed-out request at an instance is 0. In this experiment, we analyze our system's response to a sudden increase in load from 50 to 500 rps. Like in our system model from Section 3.1, the simulation generates the arrival stream with a constant rate from an infinite number of clients. Hence, we do not model single clients' behavior and reactions to request timeouts. Fig. 5 shows the mean of 30 runs of the number of deployed instances converging into the tolerance area indicated by the gray box. We see that request timeouts accelerate the convergence of our scaling process after an overload situation. Immediately after the load increases, the queues of the deployed instances fill up very quickly. If new instances are deployed, they never reach the queue size of the older instances due to the lower instance arrival rate. This means that the unfinished work in the system is unevenly distributed. Request timeouts lead to the system's queue sizes equalizing more quickly, as unfinished work on the old instances is eliminated. As Fig. 5 shows, timeout values of 10, 20, 30, and 60 s significantly reduce the overshoot amount and the settling time compared to the case with no timeouts. Lower timeout values lead to better convergence while more requests are dropped.

Non-zero instance readiness times.[4] Our model-based analysis assumes that new instances are spawned immediately after an upscaling decision. In practice, container readiness times typically range from a few milliseconds to tens of seconds [17], while other runtimes, such as VMs, might have even higher initialization times. Fig. 6 shows that non-zero readiness times degrade the performance of our approach in the step response experiment. We evaluate constant readiness times from 0 to 60 s in 5-second steps. We see a clear increase in settling time and overshoot amount. The result of a linear regression shows a high likelihood of a linear dependence both for setting time (R^2 score: 0.898, p-value: 8e-195) and overshoot amount (R^2 score: 0.911, p-value: 2e-205). The results are as expected, as readiness times delay the implementation of upscaling decisions. This prolongs the transient phase in which the unfinished work is unevenly distributed.

Other factors. As part of our simulation study, we also investigated other factors potentially influencing our scaling behavior. However, none of these factors significantly impacted our autoscaler evaluation metrics.

In practice, the resources of containers can be limited to create better performance isolation. Container runtimes and VM hypervisors support *CPU throttling* to enforce such limits. We implemented the default throttling procedure of Docker and Kubernetes [39]. We found that the scaling behavior quantified by settling time and overshoot amount is identical within the standard deviation when we compared a throttled system with CPU share $1/s$ and an unthrottled system with an s-times higher

[4] We use the term readiness time instead of start time to clearly reference when an instance is ready to process requests [17].

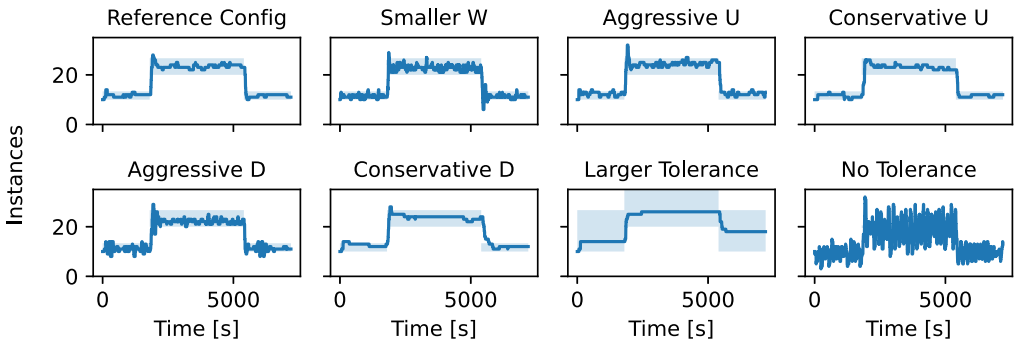


Fig. 7. Examples illustrating the influence of various configuration parameters.

service time. The primary motivation was that simulating CPU throttling makes the simulation runs very computationally intense and slow. As we focus on evaluating the scaling behavior here, we conclude that it is sufficient to evaluate unthrottled systems in our simulative analysis.

In addition, we tested different simple *load balancing policies* (round-robin, shortest queue, random, power of d) and *non-Markovian inter-arrival and service time distributions* with different variation coefficients and auto-correlation. Like CPU throttling, these mainly affect the response time distribution but not the scaling behavior, especially not the convergence of our method. We report more details on these experiments in [7].

5.3. Parameter study and configurability

This section analyzes the autoscaler performance with respect to our algorithm parameters U, D, and W. First, we show the intuitive effect of different configurations in an illustrative example. In the second step, we present the results of an extensive parameter study.

Fig. 7 shows the scaling behavior of eight configurations of our algorithm. The workload consists of one sudden load increase and decrease. The diagrams show the deployed instances over time. The optimal supply over time is marked as a blue area in the background. The top left is the reference configuration with a moderate up- and downscaling behavior. The other seven figures each represent configurations where one parameter has been changed. In this section, we have deliberately chosen to visualize the results of only one run in Fig. 7 to avoid slanting effects due to averaging.

In the first example, we reduced the mean of W from 60 s to 15 s compared to the reference configuration. The reaction time to load changes is reduced, but continuous scaling occurs even under constant load as the scaling frequency is high. With a more aggressive U, the probability that a new instance is spawned is high, even for values close to the tolerance range. With a conservative U, only values close to 100% utilization cause scaling. An aggressive U accelerates the upscaling period, but we observe a high overshoot and an increased fluctuation of the instance numbers even in the tolerance range. A conservative U prevents overshoot and provides more stability at constant load. These results can also be transferred to the D parameter. Finally, we consider the influence of the tolerance area. In the reference case, this lies between 60% and 80% utilization. Fig. 7 shows the results with a larger range between 30% and 80% and a configuration where only exactly 80% utilization guarantees no scaling. Choosing a larger tolerance reduces the number of scaling decisions significantly. In contrast, permanent up-and-down scaling occurs in the scenario with no tolerance. In summary, we can see in this intuitive example that our algorithm parameters W, U, and D have an explainable and measurable influence on the scaling behavior.

We see that configuring the scaling behavior is not trivial and that trade-offs exist, so there is no single optimal configuration. In the following, we present the results of a parameter study examining the trade-off between deployment cost and quality of service in more detail. We emulate the setup from Section 2.3 in terms of test trace, a request timeout of 10 s, and a measured readiness time distribution with a mean of 2 s. For W, we choose deterministic, normal, and uniform distributions with mean values between 15 s and 90 s. For U and D, we consider monotone functions of three groups. For lin- (exp-) functions, the probability of up- and downscaling increases linearly (exponentially) with the distance from the tolerance range. Furthermore, we consider thr-functions that always scale up or down from a certain threshold. Appendix A provides a formal definition and visualizations of the functions, while Appendix E describes our full parameter search space. In our grid search, we test a total of 3390 configurations, each of which is repeated with five random seeds. We always consider the mean values from these five repetitions in the following.

Fig. 8 shows the key insights of our parameter study. The upper left shows the trade-off between the fraction of requests with a response time less than 2 s and deployment costs. The Pareto front shows a linear cost increase for increasing QoS up to about 90%. Beyond that, the costs increase exponentially for higher performance.

The colors of the data points represent different function classes for the parameter U. The tested threshold-based configurations result in high QoS. The main reason for this behavior is that threshold-based scaling creates a large tolerance range, so fewer scaling actions happen overall. This ensures stable behavior but tends to generate high costs. Linear and exponential scaling are

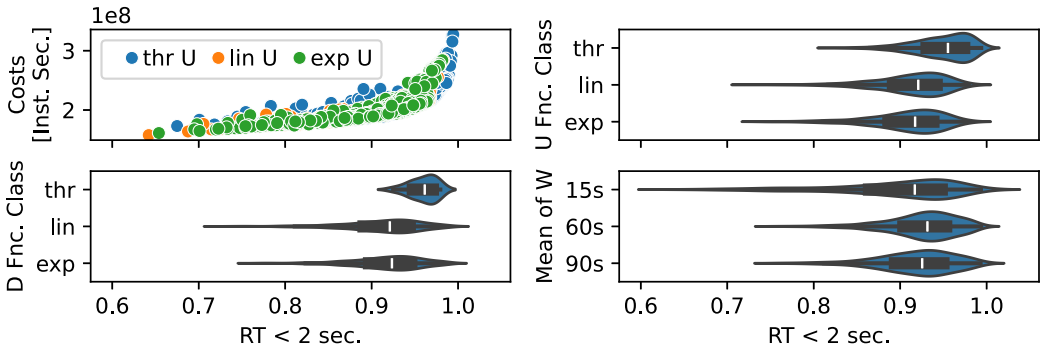


Fig. 8. Configuration trade-offs and influence of the algorithm parameters.

more balanced and largely cover the Pareto front. Here, the tolerance range can be kept small with low probabilities for scaling actions close to the tolerance range, enabling cost-reduced supply if the workload remains stable.

The lower right figure shows the results for different mean values of W. While a low mean of 15 s leads to unstable outcomes, the results for higher mean values of W are almost identical. In our simulations, more than 150 instances are active at the same time, making the scaling frequency small for any tested W. Overall, we conclude that our approach can represent a variety of configurations. The influence of U, D, and W can be explained. Users must determine their system's trade-off parameters and choose an adequate configuration. We see huge optimization potential in future works, as all algorithm parameters can be arbitrary functions.

6. Proof of concept

In this section, we analyze our approach with a proof-of-concept implementation. We focus on validating the simulation and the model by comparing them with a real system. We also compare our approach with established baseline autoscalers using realistic workloads and explore a use case with network-limited virtual machines. Our research questions are:

1. How can the approach be implemented with the low overhead for container-based cloud applications?
2. Are the model and the simulation suitable for approximating the scaling behavior of a real system?
3. How is the performance of our approach compared to established baseline autoscalers for different serverless test functions?
4. Can the approach be applied in a scenario beyond CPU-bound container applications?

6.1. System implementation

In this section, we describe the implementation of our approach for containerized applications and introduce all the components involved. The source code is available at [6]. Fig. 9 shows an overview of our system. We add a layer to the container image of the service instances to perform the logic of decentralized autoscaling (DAS). The logic is encapsulated in a tiny C program that implements Algorithm 1 and reads the CPU utilization from the container's file system. In future work, the gathering process of the monitoring data must be extended if more or different scaling metrics are to be used. The core parameters of the algorithm D, U, and W can be set either by implementing functions in the source code or via environment variables. The operator can set different configurations for different services.

This type of implementation offers the following advantages: (1) It minimizes the assumptions on the application container, (2) it can be easily integrated into existing build pipelines, and (3) it promises low overhead as no additional containers need to be started, and the process idles most of the time. The latter statement depends on the choice of W, U, and D and assumes that one iteration of the algorithm has a low runtime and uses a negligible amount of resources. We could not detect any measurable runtime overhead for the test applications we used in the following sections (e.g., significant change in throughput, response time, resource usage). By injecting the DAS binary, the image size of our test containers increased by about 1 MB to 20 MB. This value depends on whether dynamic linking is used and which libraries are present in the container. Note that there is potential for optimization here for future work.

The service instances receive a URL as an environment variable to reach the execution layer, which then forwards the decision to the control plane. By using this abstraction layer, no cluster-specific logic must be integrated into the DAS binary. Moreover, it can enforce global constraints defined by the user. In our case, we set the constraint that at least one instance of each service should always be deployed. In addition, we log every scaling decision and up- and downscaling probabilities and persist them in a database. The degree of logging might be customized and may not require a database. The communication between the instances and the execution layer is unidirectional, which means that the instances do not receive feedback on their decision. The executor containers are stateless, making them horizontally scalable at will. We emphasize that, except for constraint checking, the execution

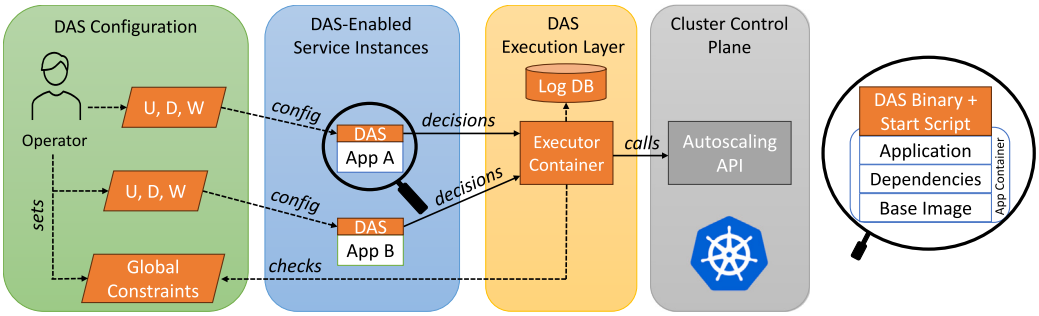


Fig. 9. Proof of concept implementation.

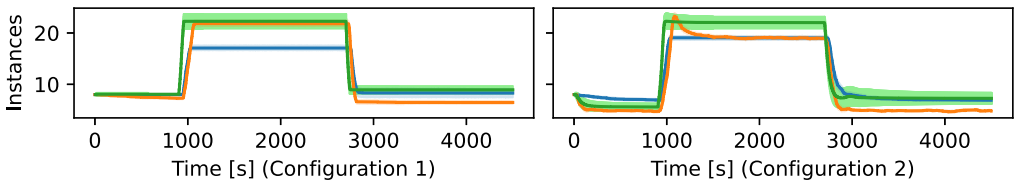


Fig. 10. Comparison of real system (blue), simulation (orange), and model (green). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

layer does not execute any other logic that might influence the scaling behavior. However, it might be worth exploring this option in future work, shifting from a purely decentralized to a hybrid approach.

In summary, to run our autoscaler, the execution layer first has to be deployed. Our replication package [6] contains an executor for Kubernetes clusters. Operators might implement further global constraints there. In the second step, our algorithm parameters D, U, and W have to be set in our C program that contains the scaling logic. Our version contains realizations for the waiting time distributions and scaling function stereotypes used in this paper (see Appendix B) that take parameters via environment variables. Operators might implement the retrieval of further scaling metrics, custom scaling functions, and waiting time distributions in dedicated functions in the source code. In the last step, the source code needs to be compiled and packaged into a binary, which must then be added to the application container image (e.g., via adding a step in the build process). The container's start command needs to change so that the DAS binary is started as a background process first, and then the original entrypoint is executed. In contrast to conventional autoscalers, no additional setup or connection to monitoring tools is required, and the whole scaling behavior can be controlled by our three parameters D, U, and W.

If certain requirements are met, our approach can be implemented for other types of instances beyond containerized applications, such as unikernels or microVMs. First, an execution layer must be deployable, able to trigger scaling and enforce user-defined global constraints. Second, the instances must be able to capture the scaling metrics and run Algorithm 1 concurrently with their business logic (e.g., as a background process). Moreover, instances must be able to submit their decisions to the control plane.

6.2. Comparing model, simulation and prototype

This section presents an experiment comparing our model, simulation, and prototype. The SHA256 hash function from Section 2.3 in the identical test setup serves as the test app. Eight instances are deployed in the beginning. The workload is 20 rps for the first 900 s. Then, the load suddenly jumps to 80 rps, while after 2700 s, the rate drops back to the initial state. We test two DAS configurations. Config 1 is threshold-based with a tolerance range between 30% and 90% utilization. The waiting time is drawn from a uniform distribution between 30 and 90 s. Config 2 has a smaller tolerance range between 60% and 80%; the probabilities for up- and downscaling increase exponentially with increasing distance to the tolerance range. The waiting time distribution is uniform between 30 and 60 s. The measurements and simulations are repeated 30 times. After carrying out the prototype measurements, we used the results to estimate the average service rate, which is then used in the model and simulation.

Fig. 10 shows the deployed instances over time expressed as mean and standard deviations. Those values can be obtained directly from the prototype and the simulation. For the model, we extracted these values from the distributions resulting from Eqs. (5) and (11). We see that simulation and model can qualitatively duplicate the behavior of the real system. All three systems converge to a value within the tolerance range. Simulation and model can also predict the behavioral change due to a different configuration, especially during downscaling.

It is noticeable that both the simulation and model overestimate the system's reaction to load increase. In Config 1, this results in an offset that is not corrected due to the large tolerance range. In the real system, about 17 instances are deployed on average, while model and simulation estimate 22. In Config 2, the simulation correctly predicts the system's steady state of about 19 instances.

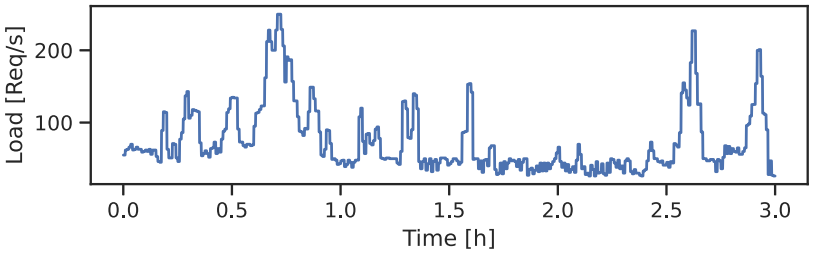


Fig. 11. Test trace for comparing the decentralized, HPA, and KPA autoscalers.

Three factors can explain the deviations from the prototype. First, the model does not consider request timeouts (here: 10 s). As shown in Section 5.2, these reduce the overshoot notably. Second, in the prototype, some instances report utilizations up to 103%. This happens because the containers' CPU limits are not strictly enforced [35] and leads to a temporarily increased service rate, further reducing the overshoot. Last, we found that the phase in which a significant difference exists in the instances' utilization is shorter than in the simulation. One reason for this could be a utilization-aware load balancer. In the simulation, the requests are always distributed randomly. An intelligent load balancer could also be a root cause for deviations in the downscaling case. In Config 2, the prototype and model converge to 7, while the simulation predicts 5 deployed instances. Another explanation is the inaccurate system state right before the load drops, leading to more downscaling decisions, especially in the simulation. These deviations are then not corrected as a steady state is hit.

A general uncertainty factor is the service time estimation. In the simulation, we assume an exponentially distributed service time, while the model considers only the rate. This also contributes to an incorrect prediction of the steady state. However, considering that only a few incorrectly predicted instance decisions are the root causes, the deviations are smaller than the visual impression conveys. Overall, both the simulation and the model provide valuable approximations for the real system despite some systematic deviations.

6.3. Comparison with established autoscalers

In this section, we compare our approach with two established autoscalers and evaluate its performance for two different stateless containerized applications, in this case, serverless functions. In addition to the HPA and its cost-efficient variant HPA2 introduced in Section 2.3, we also consider the Knative Pod Autoscaler (KPA), which is often used in the FaaS context. We chose these three baselines as they are de-facto industry standards [2,3] and can, similar to our approach, be used out-of-the-box. A comparison to other state-of-the-art autoscalers that require a more extensive setup (e.g., pretraining) is out of the scope of this paper. The test workload shown in Fig. 11 is a trace from a production system [3]. We evaluate two test functions from a serverless test dataset [40]. The *Matrix* function (Python) inverts a random matrix of variable size while *Image* (Node.js) generates a random image and then resizes it. Matrix is much faster and more stable in its performance than Image. Consequently, different SLO values of 100 ms and 2 s have been chosen. All autoscalers aim to maintain a target utilization of 75%. The decentralized autoscaler uses a tolerance range between 70% and 80% with exponentially increasing probabilities for up- and downscaling outside this range. The HPAs run with a sync interval of 15 s. HPA uses the default configuration, while HPA2 allows faster downscaling. The KPA does not scale based on CPU utilization but either on the average concurrency (queue size) or the arrival rate. To achieve comparability, we tested the KPA in different configurations and chose an arrival rate-based policy that keeps the utilization around 75%. A detailed list of all parameters can be found in Appendix B.

Table 1 shows the QoS and cost metrics of all autoscalers in the average and standard deviation of five repetitions. The costs are normalized to the mean costs of HPA. DAS causes the lowest costs for both test functions. For Matrix, it achieves this with minimally reduced QoS values compared to HPA. In the case of the more unstable Image function, the QoS metric decreases by about 10%, with a cost saving of about 60%. Our approach beats HPA2 in costs and QoS for both test functions. The KPA ranks third in costs but last in QoS for Matrix and Image. However, it should be noted that the KPA uses a different scaling metric and processes the requests via a queue proxy. Therefore, the comparison is not fair. Nevertheless, we decided to use it as a reference and to compare the achieved QoS at a cost level similar to the other scalers. In addition to the primary cost and SLO metrics, Table 1 also shows the total number of container starts and shutdowns. This metric helps to further understand the behavior of the decentralized autoscaler since the missing coordination between the instances could raise concerns over many up- and downscaling decisions that cancel each other out. As expected, the numbers for DAS are significantly higher than for HPA, which has longer scaling and stabilization periods. However, the numbers of DAS are smaller than those of HPA2 (Matrix and Image) and KPA (Matrix). This shows that the scaling behavior of DAS can be as stable as that of centralized solutions. As our parameter study in Section 5.3 shows, the algorithm parameters U, D, and W can effectively control the scaling frequency and stability.

In general, the experiments shown here are to be understood as proof of concept. Due to the extensive measurement effort, we only show selected configurations of all autoscalers here. Other configurations and other test applications may produce different results. A detailed experimental study is left for future work. However, we conclude that our decentralized approach has great potential, creating at least comparable results to state-of-the-art autoscalers. In the case of rapid load changes, we adapt faster than the HPA due to the higher scaling frequency.

Table 1
Scaling metrics for different autoscalers.

Scaler	Matrix			
	RT < 100 ms [%]	Costs	Starts	Shutdowns
DAS	98.52 ± 0.08	0.381 ± 0.012	171.6 ± 5.7	166.2 ± 5.6
HPA	98.85 ± 0.05	1.000 ± 0.019	92.4 ± 4.8	68.2 ± 4.2
HPA2	98.47 ± 0.02	0.502 ± 0.014	177.2 ± 5.8	174.6 ± 5.7
KPA	94.36 ± 4.65	0.838 ± 0.000	233.0 ± 4.6	209.8 ± 4.0
Scaler	Image			
	RT < 2 s [%]	Costs	Starts	Shutdowns
DAS	80.72 ± 2.16	0.393 ± 0.048	125.8 ± 12.2	134.2 ± 12.6
HPA	89.23 ± 0.26	1.000 ± 0.025	65.4 ± 2.9	60.0 ± 2.3
HPA2	77.72 ± 0.87	0.442 ± 0.006	174.0 ± 9.9	182.4 ± 9.9
KPA	63.81 ± 2.24	0.807 ± 0.001	90.6 ± 1.5	81.6 ± 1.5

6.4. Beyond CPU-bound container applications

In the previous sections, we showed that our approach can scale fine-grained serverless functions deployed in containers with the CPU as their bottleneck. While this was the primary motivation of this work, we introduced our approach on an abstract level and referred generically to *instances* as the entities to scale. In this section, we demonstrate the applicability of our approach in a scenario involving virtual machines, and the limiting resource is the egress network bandwidth.

Our scenario is inspired by a video streaming use case where a variable number of customers c request streams from relay servers. A minimum bitrate b_{min} is necessary to maintain the desired video quality. In our scenario, customers have enough bandwidth while a single relay server has a limited egress network bandwidth n_{max}. When more customers request streams from a single relay server, the fraction of n_{max} and c could fall below b_{min}, which means the customers cannot view their videos with the desired quality. New relay servers must be deployed (in a different network location) to serve the growing number of customers. The video streaming provider is interested in maintaining the quality of service and minimizing costs, meaning that when fewer customers are streaming, fewer relay servers should be deployed.

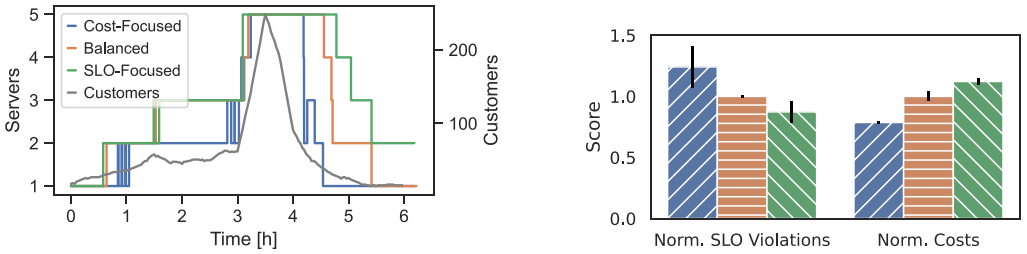
We implement this scenario using virtual machines in the Google Cloud. The relay servers are VMs of type e2-highcpu-4. The per-VM egress bandwidth in GCP is generally limited to 2 Gbps per vCPU.[5] To create bottlenecks with a reasonable amount of workload (customers), we limit the egress bandwidth further to 600 Mbps using Linux's `tc` command. The customers are deployed on an e2-medium VM with 30 Gbps ingress bandwidth. Another e2-medium VM acts as the control plane. The control plane takes care of deploying and removing relay servers. The control plane balances the load between relay servers using a least active connections strategy. We use `iperf3`[6] to emulate video streaming, which establishes a constant bitrate UDP stream between the relay server and the client. We set the desired bitrate per stream to 15 Mbps, the recommendation for 4K streams on Netflix.[7]

We deploy our decentralized autoscaler on every relay server. It tracks the number of sent bytes using Linux's built-in network interface stats. It initiates upscaling when the sent rate in the last scaling interval goes close to the limit of 600 Mbps and downscaling when the used bandwidth is low. Scaling decisions are submitted to the control plane, which acts as the execution layer. In the case of downscaling, it checks that at least one relay server remains active. If so, the control plane removes the server with the least active connections from the load balancer. When all streams end on that server, it is shut down. When a new relay server is started, it is added to the load balancer once it passes a health check. Customers do not switch the relay server during a stream once assigned to one; there is no rebalancing when the number of servers changes.

Fig. 12(a) shows our workload as the number of active customers over time. It represents a 24-hour cycle that starts and ends at 6am and has its peak at primetime in the evening hours [41]. Customers request videos of different lengths. In our scenario, the stream duration follows a normal distribution with a mean of 8 min and a standard deviation of 2 min. The deployment of new virtual machines took 15 s on average. These values differ significantly from the ones in earlier sections, where a client request was processed within milliseconds or a few seconds, and the readiness time of the containers was negligibly small. Hence, we must use significantly higher waiting times for the decentralized autoscaler; we choose a uniform distribution between 3 and 5 min here.

To reduce the experiment time and allow for multiple repetitions, we decided to speed up the described scenario by a factor of 4. This means that the actual experiment duration was six instead of 24 h. The stream durations and waiting times are scaled accordingly. To gain control of VM readiness times, we do not actually delete the VMs when scaling down. Instead, we remove them from the load balancer and keep them running. During upscaling, the health check of the inactive VMs succeeds after the virtual readiness time. To determine an empirical distribution for this virtual readiness time, we measured 50 VM readiness times in GCP and divided their value by our scale factor 4. Adjusting the experiment duration was necessary to save costs. However, the results are still valid as the dynamic of the original setting is preserved, which is most important for autoscaler evaluation. In

[5] https://cloud.google.com/compute/docs/network-bandwidth.
[6] https://iperf.fr/.
[7] https://help.netflix.com/en/node/306.



(a) Deployed servers and workload.

(b) QoS and cost metrics.

Fig. 12. Comparison of different DAS configurations in a video streaming use case.

addition, we only consider normalized metrics in our evaluation. We test three DAS configurations and repeat every setting five times, yielding 90 h of measurement data overall. The three DAS configurations (*cost-focused*, *balanced*, and *SLO-focused*) all use different exponential scaling functions U and D that represent different weights for the opposing goals of QoS maintenance and cost minimization. All parameters are reported in Appendix C.

Fig. 12(b) shows the costs and SLO violations of the three tested DAS configurations. The values have been normalized to the costs and SLO violations of the *balanced* scaler. The costs are defined as the integral of the active relay servers over time. For the SLO violations, we consider the number of customers that experience a decreased bitrate and, therefore, potentially a video quality drop. We consider a bitrate decreased when it is less than 98% of the desired value of 15 Mbps; `iperf3` reports this metric once every second. Fig. 12(b) confirms the expected costs and QoS tradeoff and, therefore, underlines the configurability of our approach. In addition, Fig. 12(a) shows exemplary runs of every DAS configuration. All settings generally react adequately to changing workloads in this scenario, while the *cost-focused* setting scales up last and initiates downscaling first.

This scenario shows the applicability of our approach to a horizontal autoscaling scenario beyond CPU-bound container applications. We showed that the configuration parameters remain powerful and explainable. The waiting time distribution W had to be adjusted to match the changed workload dynamics and instance readiness times. The scaling functions U and D retained their exponential shape and were fitted to the new scaling metric. Overall, we tested our approach in (1) a serverless FaaS environment with different test applications, workloads, and baselines and (2) in a VM-based use case inspired by video streaming with different algorithm configurations. However, we can only cover some potential scenarios and cannot guarantee that our approach performs better than conventional autoscalers in all cases. A comprehensive discussion follows in the next section.

7. Discussion

Evaluation summary. A key finding is that although we do not coordinate between instances, our approach can achieve a scaling performance comparable to established autoscalers. This has been shown for different system sizes, test applications, and workload patterns and was evident from the model, the simulation, and the prototype. Furthermore, our parameter study has shown that our autoscaling approach is configurable, i.e., by changing parameters that adjust the instances' scaling behavior, the overall scaling behavior is significantly and explainably influenced (see C4 from Section 2.1). In scenarios with highly dynamic loads, we outperform baselines with our approach's ability to react quickly to sudden workload changes (C1). This is enabled by high-frequency decisions made from a small action space. In addition to pure performance, we differ conceptually from established autoscalers. In particular, there is no dependency on central monitoring systems; instead, arbitrary scaling policies can be defined (C3). A core characteristic of our approach is the flexibility an operator has in defining scaling functions without being bound to a given configuration space. However, we could also show that specific configurations, such as exponential up- and downscaling, yield good results for different test apps. This enables a good balance between application-specific and agnostic solutions (C2). Our results indicate that our model and simulation can be tools to estimate the scaling behavior in different scenarios and configurations.

Strengths and weaknesses of our approach. A core advantage of our approach is the faster reaction time to load changes compared to baselines, which we achieve by distributing scaling decisions randomly over time. Both in the simulation and experiments, we were able to show that we can better match the actual resource demand, which leads to fewer SLO violations in the upscaling case and less resource wastage in the downscaling case. This increase in scaling frequency also means we take many small scaling decisions instead of a few critical ones. This makes it possible to use comparatively simple, explainable scaling policies. Through our decentralized decision-making, we eliminate the dependency on a specific centralized monitoring tool, which is a single point of failure, making us more robust against missing, delayed, and inaccurate monitoring data. Also, in our approach, there is no need to submit measurement values to a central instance, which could save network bandwidth in constrained environments. Our scaling decisions can be represented with a constant message size, essentially 1 bit (UP or DOWN), while sending HOLD decisions or logging information is optional. Our scaling functions impact the scaling behavior intuitively and provide a compact configuration interface compared to other autoscalers that require extensive manual configuration through multiple parameters. Moreover, they offer a significantly higher configuration space than traditional threshold-based approaches, so our approach can be very well customized for specific use cases. In summary, these conceptual advantages could make our approach beneficial in a wide variety of usage scenarios.

However, the large configuration space can also be intimidating at first. More experimentation is needed here, and adequate scaling policies must be found. This is a problem for any autoscaler that allows for more sophisticated configurations beyond simple thresholds. Other works use automated approaches like (reinforcement) learning to find suitable settings, lowering the manual experimentation effort. We have decided against this in the paper because of the issues of trust and explainability (see Section 2.1). In this study and several preliminary experiments, we found some solid default strategies, but further exploration and optimization are out of the scope of this paper.

Overall, a certain amount of expertise is required to use our approach; in addition to the parameter selection for up- and downscaling and the waiting behavior, the instrumentation of the test application currently also requires some manual work, although this can certainly be automated in the future. For smaller use cases with low load intensity and few deployed replicas, our approach might not have huge advantages over established autoscalers, as the monitoring overhead here is not large anyway, and the deterministic behavior of traditional autoscalers achieves greater consistency. Our autoscaler, with its probabilistic nature, is especially advantageous for larger system sizes. As outlined in Section 6.1, the steps to run our autoscaler are manageable and are not dependent on the system size, just like the scaling policies. The efforts to set up our approach are slightly higher than for integrated solutions such as the Kubernetes HPA, but lower than for many other research autoscalers that require manual or trained models. A core assumption of our approach is fast instance readiness times and generally high elasticity of cloud environments, enabling scaling decisions to be executed at high frequency. Different instance start times and workload dynamics might be addressed by adjusting the waiting time distribution as demonstrated in Section 6.4.

Threats to validity. Our measurement results from the cloud systems are subject to variability [42,43]. We addressed this issue by repeating all measurements at least five times. In general, there are open questions regarding the generalizability of our (experimental evaluation) results. We chose representative test applications and environments used in prior works but could not evaluate all possible use cases of our approach. It is not guaranteed that our autoscaler outperforms established solutions in every use case. The focus of this paper is to introduce a novel autoscaling approach that differs conceptually from the state-of-the-art and to perform formal, simulative, and experimental analyses. Especially in the experimental analysis, there is much more to explore as many factors play a role here (e.g., test workloads, test applications, baseline autoscalers, and parameter settings of our approach).

We have limited ourselves to comparisons with the standard scalers of Kubernetes and Knative. These are very well suited as baselines because they are frequently used in industry and can be used universally and without training, just like our approach. A comparison with specialized solutions, such as reinforcement learning autoscalers, remains for future work. By design, we are limited to capturing metrics per instance only; we have yet to evaluate how services with external dependencies, e.g., synchronous calls to databases, scale. Moreover, the role of the load balancer has yet to be fully explored. We cannot make provable statements about its impact, as we used closed-source Google Cloud load balancers in our experiments. We decided to choose the default and maintained load balancer for our environment instead of a custom solution to preserve the representativeness of our test environment. Lastly, our prototype evaluation was conducted in relatively small, homogeneous clusters as we lacked the resources to conduct large-scale experiments. However, Section 5.1 indicates that our approach performs well in larger systems.

Possible extensions. Future work will evaluate more complex scaling functions with more input metrics or a reasonable history (e.g., the last five time steps), incorporating trends into the decision. Our model and experiments with serverless functions only covered cases where the QoS of an instance is defined by a single response time threshold, assuming that there is only one class of request. Other entities, e.g., microservices with different endpoints, might use different SLOs for different request types. In general, this could make the autoscaler configuration more difficult. However, depending on the technology stack, metrics for different request types (such as arrival rates) might be available at the instance (e.g., in log files) and could be fed into our autoscaling algorithm as well.

However, it must be kept in mind that not arbitrarily complex policies can be defined at the instance level, as the autoscaling logic could impact the instance performance. It is also possible to integrate additional logic in the execution layer, shifting from a purely decentralized to a hybrid approach. The execution layer could also be used to enable learning for our framework. For this purpose, our prototype already allows to reconfigure scaling policies via environment variables. The model and the simulation could be refined and extended to serve as tools for optimally tuning scaling functions in an automated way. To reduce the manual effort in instrumenting test applications, improvements towards integrating the injection of the DAS binary into existing build and deployment pipelines can be targeted.

8. Related work

Numerous surveys [1,4,9,16] provide an overview and taxonomies of the current state of autoscaling research in general. Our autoscaler can be categorized as a reactive, horizontal autoscaler in these taxonomies. Our reasoning approach and scaling metrics are flexible and depend on the choice of the scaling functions. In contrast to many reviewed works, we evaluate our autoscaler using modeling, simulation, and experiments. In the following, we take a closer look at related works in the areas of decentralized autoscaling, probabilistic autoscaling, autoscaler modeling and verification, and decentralized cloud orchestration.

Decentralized autoscaling. DEPAS [44–46] is an interval-based decentralized autoscaler built for P2P architectures. Similar to our work, scaling decisions are made at the service instance level. In contrast to our work, the work assumes that each instance is aware of a set of other instances and can approximate the total system load. DESA [47] is a decentralized autoscaler for fog computing environments. In contrast to our approach, the work is limited to threshold-based scaling and simulative evaluation. Fractal [48] embeds orchestration logic into service replicas, including scaling, failure recovery, and more. In contrast to our work, the approach is strongly coupled to a concrete orchestrator and uses threshold-based scaling only. In the Swayam [49] approach, frontend services

can autonomously trigger scaling for backend ML inference services. One key difference to our work is that the authors focus on ML backend services with high setup times and explicitly keep idle services deployed for some time.

An autoscaling approach where decisions are made at the last instance of a call chain is proposed in [50]. In contrast to our work, an explicit model of the load balancer and only threshold-based scaling is used. Two works [51,52] propose decentralized scaling for microservices where decisions are made at the node level, making them not easily applicable in the FaaS use case where servers are hosting many different service instances. A hybrid threshold-based autoscaling approach, evolving both local decisions and global monitoring data, is proposed in [53]. In general, several decentralized scaling approaches [10,54,55] specialized in stream processing tasks have been proposed. In summary, the uniqueness of our work is that we propose a pure-local continuous scaling approach with flexible configuration.

Use of probability in autoscaling. DEPAS uses explicit probabilities for scaling actions as an outcome of their decision policies. In [56], a model outputs probabilities for resource over- and underprovisioning. Similar to our work, comparisons with random numbers are used to ultimately decide which action to take. Our work adopts this idea and extends it with the execution layer that has the right to deny the execution of a decision (e.g. if it violates a global constraint). Other works use probabilities as part of intermediate steps to determine the scaling action, primarily to handle uncertainties. For example, MagicScaler [57] uses probabilistic demand prediction and a sampling method to approximate an optimal scaling policy.

Autoscaler modeling and verification. A review [58] finds that discrete- and continuous-time Markov chains (MC) or decision processes (MDP) are the most popular model specifications for autoscaler verification. Comparative studies outline the strengths and weaknesses of different modeling formalisms for cloud autoscaling [59,60]. A recent work [61] studies VM autoscaling approaches based on hysteresis policies, i.e., multi-threshold rules, using MCs and MDPs. These hysteresis policies can be interpreted as special cases of our scaling functions. An alternative modeling formalism for microservice autoscalers is presented in [62], where the authors use hierarchical parallel Petri Nets for verifying the scaling behavior. PEAS [63] allows for computing probabilistic bounds for the autoscaler behavior by evaluating many workload scenarios.

Decentralized cloud orchestration. Several papers dealing with the decentralization of orchestration tasks in cloud and edge computing are reviewed in [64]. For example, Swarmchestrate [65] and CODECO [66] envision decentralized orchestration frameworks for the cloud–edge continuum. The paper [67] motivates using autonomic, self-organizing approaches to cope with decentralized multi-cloud setups. MiCADO [68] is an application-level cloud orchestration framework. MiCADO supports scaling both at the virtual machine and application level and allows for creating flexible scaling rules that can also be optimized with a machine learning model [69]. In contrast to our work, scaling is based on metrics from Prometheus, a metric collector that combines application- and VM-based metrics. ENORM [70] is a resource management framework for edge servers with support for autoscaling, resource provisioning, and more. Edge nodes can make scaling decisions here, but only in periodic intervals. In the HYDRA [71] approach, orchestration tasks are performed collectively by a set of nodes, and each node in the system serves both as a computation resource and controller. Another related research area is decentralized monitoring systems. Recent research from this area originates in the edge computing community aiming to increase fault tolerance [72] and workload balance [73]. In contrast to these works, our approach targets autoscaling only and leaves other orchestration tasks (such as placement or load balancing [74]) to other management components.

9. Conclusion

In this paper, we introduced a novel approach to autoscaling applicable to a wide range of modern cloud applications. In our approach, service instances make autoscaling decisions autonomously based on local monitoring data. By distributing the decisions over time, we significantly increase the scaling frequency compared to interval-based approaches and achieve a quasi-continuous process when many instances are deployed. In our model-, simulation-, and experiment-driven evaluation, we could show that: (i) Although there is no coordination between instances, our approach converges independently of the system size, test application, and workload pattern. (ii) Our approach can cover a large configuration space, allowing us to tune the autoscaling behavior as desired in particular use cases. (iii) Our autoscaler can improve scaling performance compared to established baseline autoscalers, especially in scenarios with high load fluctuations. Therefore, we conclude that our approach has great potential and that there are many interesting research questions for future works. The main focus in the future will be the testing and optimization of more complex scaling functions and the evaluation in larger systems.

CRediT authorship contribution statement

Martin Straesser: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Conceptualization. **Stefan Geissler:** Writing – review & editing, Writing – original draft, Validation, Software, Formal analysis. **Stanislav Lange:** Writing – review & editing, Writing – original draft, Software, Formal analysis. **Lukas Kilian Schumann:** Software, Investigation. **Tobias Hossfeld:** Writing – review & editing, Supervision, Formal analysis. **Samuel Kounev:** Writing – review & editing, Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 510552229.

Appendix A. Definitions of scaling functions

In this section, we define the scaling functions that are used at various points in the work. In our case, most scaling functions receive only the CPU utilization ρ as input and output the probabilities for upscaling and downscaling. For both upscaling and downscaling, we test monotonic functions in this work. The downscaling function D decreases with increasing utilization ρ. The upscaling function U increases with increasing utilization. In the following, we introduce the variables τ_{min} and τ_{max}. τ_{min} is the smallest utilization value for which the downscaling probability is 0. Analogously, τ_{max} is the largest utilization value for which the upscaling probability is 0.

$$\tau_{min} = \min(\rho \in [0;1] : D(\rho) = 0) \qquad \tau_{max} = \max(\rho \in [0;1] : U(\rho) = 0)$$

According to constraint 1 from Section 3.2, there is no value ρ for which both U and D are greater than 0. Together with U and D being monotonic functions it follows that $\tau_{min} \leq \tau_{max}$. In the following, we introduce three classes of scaling functions, which we look at in more detail in this work. Fig. A.13 shows a schematic representation of the scaling functions and their parameters. For all function classes, we introduce the parameter $c \leq 1$. c represents the maximum of the scaling function and, therefore, the maximum probability for a scaling action. This means it is not set that there are input values for which a scaling action is guaranteed. This allows for defining policies that implement a hysteresis-like behavior. As the first class of scaling functions, we consider threshold functions. We denote threshold functions used for downscaling as $\text{thrD}_{\alpha,c}$ and define them as follows:

$$thrD_{\alpha,c}(\rho) = \begin{cases} c & \rho < \alpha \\ 0 & \text{else} \end{cases} \qquad 0 \leq \alpha$$

If a threshold function is used for upscaling, we call this function $\text{thrU}_{\beta,c}$ and define it analogously:

$$thrU_{\beta,c}(\rho) = \begin{cases} c & \rho > \beta \\ 0 & \text{else} \end{cases} \qquad \beta \leq 1$$

In addition to threshold functions, we also consider functions with linear and exponential slopes. In the downscaling case, these functions are constructed as follows. Consider utilization values τ_{min} and α with $D(\tau_{min}) = 0$ and $D(\alpha) = c$. As explained in Section 3.2, τ_{min} denotes the lower bound of the tolerance range. linD-functions are linearly decreasing between α and τ_{min} and defined as:

$$linD_{\alpha,\tau_{min},c}(\rho) = \begin{cases} \min(c, \frac{\tau_{min}-\rho}{\tau_{min}-\alpha} \cdot c) & \rho < \tau_{min} \\ 0 & \text{else} \end{cases}$$

expD-functions have an exponential slope in the interval $[\alpha; \tau_{min}]$ and are defined as:

$$\text{expD}_{\alpha,\tau_{min},c}(\rho) = \begin{cases} \min(c, e^{k\cdot(\rho-\alpha)} - 1) & \rho < \tau_{min} \\ 0 & \text{else} \end{cases} \quad \text{with} \quad k = \frac{log(1+c)}{\alpha - \tau_{min}}$$

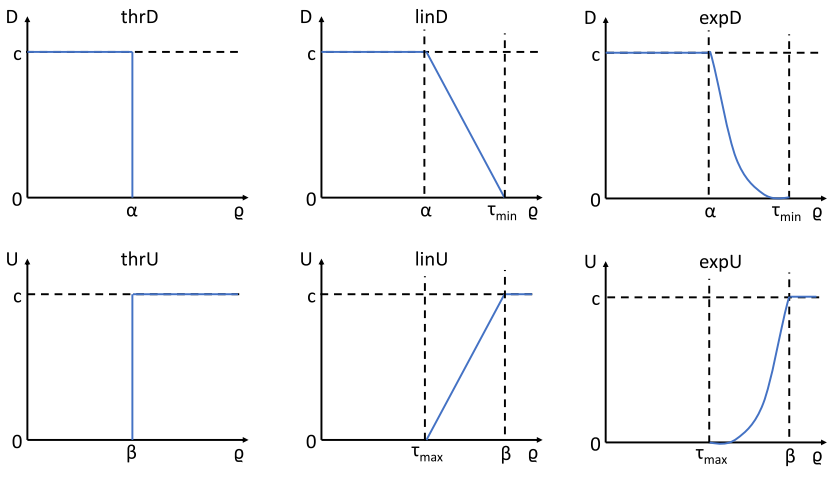


Fig. A.13. Schematic representation of the used scaling functions and their parameters.

In the upscaling case, the definitions are similar. We first define two points τ_{max} and β with $U(\tau_{max}) = 0$ and $U(\beta) = c$. linU-functions have a linear gradient between these two points and are defined as:

$$linU_{\beta,\tau_{max},c}(\rho) = \begin{cases} \min(c, \frac{\rho-\tau_{max}}{\beta-\tau_{max}} \cdot c) & \rho > \tau_{max} \\ 0 & \text{else} \end{cases}$$

Last but not least, expU-functions have an exponential slope between τ_{max} and β and are defined as:

$$expU_{\beta,\tau_{max},c}(\rho) = \begin{cases} \min(c, (1+c) \cdot e^{k\cdot(\rho-\beta)} - 1) & \rho > \tau_{max} \\ 0 & \text{else} \end{cases} \quad \text{with} \quad k = \frac{-log(1+c)}{\tau_{max} - \beta}$$

Appendix B. Parameters for prototype experiments

The following tables show the parameters used for our experiments with the proof-of-concept implementation. The columns represent experiments described in different paragraphs in this work. Cells marked with * indicate that different values were chosen for this parameter in this experiment. Unless otherwise stated, these values can be found in the main body of the paper and in our replication package [7]. All parameters for the Horizontal Pod Autoscaler (HPA) and Knative Pod Autoscaler (KPA) that are not stated in this table have their default values.

Parameters	Section 2.3	Section 6.2	Section 6.3
Repetitions	5	30	5
DAS parameters			
W	Uniform(30 s, 90 s)	*	Uniform(30 s, 90 s)
U	$\text{expU}_{0.8,0.95,1}$	*	$\text{expU}_{0.8,0.95,1}$
D	$\text{expD}_{0.6,0.7,0.3}$	*	$\text{expD}_{0.3,0.6,1}$
HPA parameters			
Sync interval [s]	60 (HPA1) 15 (HPA2)	N/A	15 (HPA) 15 (HPA2)
Target utilization [%]	75	N/A	75
Downscale stabilization [s]	300 (HPA1) 0 (HPA2)	N/A	300 (HPA) 0 (HPA2)
KPA parameters			
Scaling metric	N/A	N/A	rps
Target	N/A	N/A	9 (Matrix) 3 (Image)
Other parameters			
Request timeout [s]	10	10	10
CPU throttling [mCores]	100	100	100 (Matrix) 200 (Image)
Load balancer	Google Cloud	Google Cloud	Google Cloud

Appendix C. Parameters for the video streaming use case

The following table lists the parameters used for the video streaming use case discussed in Section 6.4. The parameters for the expU and expD functions represent the fraction of the measured egress bandwidth divided by the per-server bandwidth limit of 600 Mbps.

Parameters	Cost-optimized	Balanced	QoS-optimized
Repetitions	5	5	5
DAS parameters			
W	Uniform(45 s, 75 s)	Uniform(45 s, 75 s)	Uniform(45 s, 75 s)
U	$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.7,0.95,1}$	$\text{expU}_{0.7,0.8,1}$
D	$\text{expD}_{0,0.6,1}$	$\text{expD}_{0,0.3,0.5}$	$\text{expD}_{0,0.2,0.3}$
Other parameters			
Load balancer	Least active connections		

Appendix D. Simulation parameters

The following tables show the simulation parameters used for our experiments in Section 5. The columns represent experiments described in different paragraphs in this work. The column name indicates the focus of the experiment. Cells marked with * indicate that different values were chosen for this parameter in this experiment. Unless otherwise stated, these values can be found in the main body of the paper and in our replication package [7].

Parameters	Section 5.1	Section 5.2	
	Scalability	Timeouts	Readiness times
Repetitions	5	30	30
DAS parameters			
W	unif(30 s, 90 s)	unif(30 s, 90 s)	unif(30 s, 90 s)
U	$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.8,0.95,1}$
D	$\text{expD}_{0.2,0.6,1}$	$\text{expD}_{0.2,0.6,1}$	$\text{expD}_{0.2,0.6,1}$
Workload parameters			
Shape	Step	Step	Step
Min rate [rps]	*	50	50
Max rate [rps]	*	500	500
Other parameters			
Service time distribution	Exponential	Exponential	Exponential
Mean service time [ms]	200	200	200
Request timeout [s]	10	*	10
Readiness time [s]	0	0	*
CPU throttling	Off	Off	Off
Load balancer	Random	Random	Random

Section 5.2 (contd.)				Section 5.3
CPU throttling	Load balancer	Non-Markov service	Non-Markov interarrival	Parameter study
30	30	30	30	30
DAS parameters				
unif(30 s, 90 s)	unif(30 s, 90 s)	unif(30 s, 90 s)	unif(30 s, 90 s)	* (see Appendix E)
$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.8,0.95,1}$	$\text{expU}_{0.8,0.95,1}$	* (see Appendix E)
$\text{expD}_{0.2,0.6,1}$	$\text{expD}_{0.2,0.6,1}$	$\text{expD}_{0.2,0.6,1}$	$\text{expD}_{0.2,0.6,1}$	* (see Appendix E)
Workload parameters				
Step	Step	Step	* (see [7])	Combined
50	50	50	* (see [7])	120
500	500	500	* (see [7])	548
Other parameters				
Exponential	Exponential	* (see [7])	Exponential	Exponential
*	200	* (see [7])	200	200
10	10	10	10	10
0	0	0	0	Measured from Section 2.3
*	Off	Off	Off	Off
Random	*	Random	Random	Random

Appendix E. Search space of the simulation-driven parameter study

The following table describes all parameters tested for our parameter study described in Section 5.3. The study is structured as a grid search, so all combinations between all parameter variants were tested.

DAS parameter	Tested values
W	Deterministic distributions with values of 15 s, 30 s, and 60 s Normal distributions $N(\mu, \sigma)$: $N(30\text{ s}, 5\text{ s})$, $N(60\text{ s}, 10\text{ s})$, $N(90\text{ s}, 15\text{ s})$ Uniform distributions $U(a, b)$: $U(15\text{ s}, 90\text{ s})$, $U(30\text{ s}, 90\text{ s})$, $U(15\text{ s}, 120\text{ s})$, $U(30\text{ s}, 120\text{ s})$

DAS parameter	Tested values
U	Threshold-based thrU functions with $\beta \in \{0.7, 0.8, 0.9\}$ and $c = 1$ Linear Upscaling linU with $\tau_{max} = 0.8$, $\beta = 0.95$, and $c \in \{0.3, 0.5, 1\}$ Exponential Upscaling expU with $\tau_{max} = 0.8$, $\beta = 0.95$, and $c \in \{0.3, 0.5, 1\}$
D	Threshold-based thrD functions with $\alpha \in \{0.2, 0.3, 0.4\}$ and $c = 1$ Linear Downscaling linD with $\alpha \in \{0, 0.3\}$, $\tau_{min} \in \{0.6, 0.7, 0.8\}$, and $c \in \{0.3, 0.5, 1\}$ Exponential Downscaling expD with $\alpha \in \{0, 0.3\}$, $\tau_{min} \in \{0.6, 0.7, 0.8\}$, and $c \in \{0.3, 0.5, 1\}$

Appendix F. Definition of step response metrics for autoscaler evaluation

In this section, we define the metrics shown schematically in Fig. 4 used to quantify the autoscaler behavior after a load increase. In the following, t_0 denotes the time the load on the system is changed. λ is the arrival rate at the system after the load increase. μ denotes the service rate. In order to calculate the step response metrics, the system's desired target state G after the load jump must be defined. In our case, it is an interval in which the number of deployed instances should be. G results directly from the scaling functions U and D and the derived tolerance range for the utilization $\tau = [\tau_{min}; \tau_{max}]$. G includes all instance numbers k for which the mean instance utilization lies within the tolerance range τ. Using the general definition of the utilization $\rho = \lambda/\mu$, G results as:

$$G = \left[\frac{\lambda}{\tau_{max} \cdot \mu}; \frac{\lambda}{\tau_{min} \cdot \mu} \right]$$

In the following, let $I(t)$ be the time series of deployed instances. The rise time T_R is defined as the time difference between the time of the load increase t_0 and the first time the target state G is reached:

$$T_R = \min(\{t : I(t) \in G\}) - t_0$$

The settling time T_S is defined as the time difference between t_0 and the point in time at which $I(t)$ has a value in G and after which all further values of $I(t)$ lie in G:

$$T_S = \min(\{t : (I(t) \in G) \wedge (\exists \bar{t} < t : I(\bar{t}) \notin G)\}) - t_0$$

Last but not least, the overshoot amount N_{max} is defined as the difference between the maximum of $I(t)$ and the maximum of G:

$$N_{max} = \max(I(t)) - \max(G) = \max(I(t)) - \frac{\lambda}{\tau_{min} \cdot \mu}$$

Data availability

A link to the code and data relevant to this work is shared in the article.

References

[1] P. Singh, P. Gupta, K. Jyoti, A. Nayyar, Research on auto-scaling of web applications in cloud: survey, trends and future directions, Scalable Comput.: Pract. Exp. (2019).
[2] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani, L. Baresi, Autoscaling solutions for cloud applications under dynamic workloads, IEEE Trans. Serv. Comput. (2024).
[3] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, S. Kounev, Why is it not solved yet? Challenges for production-ready autoscaling, in: Proceedings of the ACM/SPEC on International Conference on Performance Engineering, Association for Computing Machinery, 2022.
[4] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: State of the art and research challenges, IEEE Trans. Serv. Comput. (2018).
[5] M.S. Aslanpour, A.N. Toosi, R. Gaire, M.A. Cheema, Auto-scaling of web applications in clouds: A tail latency evaluation, in: IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC, 2020.
[6] URL https://github.com/DescartesResearch/ContinuousDecentralizedAutoscaling.
[7] URL https://doi.org/10.24433/CO.6318632.v2.
[8] L. Baresi, G. Quattrocchi, A simulation-based comparison between industrial autoscaling solutions and COCOS for cloud applications, in: IEEE International Conference on Web Services, ICWS, 2020.

[9] C. Qu, R.N. Calheiros, R. Buyya, Auto-scaling web applications in clouds: A taxonomy and survey, ACM Comput. Surv. (2018).
[10] L. Xu, D. Saxena, N.J. Yadwadkar, A. Akella, I. Gupta, Dirigo: Self-scaling stateful actors for serverless real-time data processing, 2023, arXiv preprint arXiv:2308.03615.
[11] M. Xu, L. Yang, Y. Wang, C. Gao, L. Wen, G. Xu, L. Zhang, K. Ye, C. Xu, Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster, Softw. - Pract. Exp. (2023).
[12] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, C. Xu, The power of prediction: Microservice auto scaling via workload learning, in: Proceedings of the 13th Symposium on Cloud Computing, Association for Computing Machinery, 2022.
[13] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, B. Mukherjee, Auto-scaling network service chains using machine learning and negotiation game, IEEE Trans. Netw. Serv. Manag. (2020).
[14] Datadog Knowledge Center, What is auto-scaling? 2024, https://www.datadoghq.com/knowledge-center/auto-scaling/.
[15] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, S. Kim, Horizontal pod autoscaling in kubernetes for elastic container orchestration, Sensors (2020).
[16] T. Chen, R. Bahsoon, X. Yao, A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, ACM Comput. Surv. (2018).
[17] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, S. Kounev, An empirical study of container image configurations and their impact on start times, in: IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing, CCGrid, 2023.
[18] C. Meng, S. Song, H. Tong, M. Pan, Y. Yu, DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning, in: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2023.
[19] Z. Wang, S. Zhu, J. Li, W. Jiang, K.K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, A.X. Liu, DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems, in: Proceedings of the 13th Symposium on Cloud Computing, Association for Computing Machinery, 2022.
[20] M. Wajahat, A. Karve, A. Kochut, A. Gandhi, Mlscale: A machine learning based application-agnostic autoscaler, Sustain. Comput.: Inform. Syst. (2019).
[21] N. Mahmoudi, C. Lin, H. Khazaei, M. Litoiu, Optimizing serverless computing: Introducing an adaptive function placement algorithm, in: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2019.
[22] Noname Security, What is auto-scaling? Types, benefits and challenges, 2024, https://nonamesecurity.com/learn/what-is-auto-scaling/.
[23] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X.K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, R. Mahajan, Dissecting overheads of service mesh sidecars, in: Proceedings of the ACM Symposium on Cloud Computing, Association for Computing Machinery, 2023.
[24] InfraCloud, D. Pooja, Should you always use a service mesh?, 2023, https://www.infracloud.io/blogs/should-you-always-use-service-mesh/.
[25] Tigera, Service mesh: Benefits, challenges, and 7 key concepts, 2024, https://www.tigera.io/learn/guides/devsecops/service-mesh/.
[26] F. Klinaku, S. Speth, M. Zilch, S. Becker, Hitchhiker's guide for explainability in autoscaling, in: Companion of the ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, 2023.
[27] Unthinkable Solutions, Applications of AI in autoscaling of cloud infrastructure, 2024, https://www.unthinkable.co/blog/applications-of-ai-in-autoscaling-of-cloud-infrastructure/.
[28] M. Mekki, B. Brik, A. Ksentini, C. Verikoukis, XAI-enabled fine granular vertical resources autoscaler, in: IEEE 9th International Conference on Network Softwarization, NetSoft, 2023.
[29] T. Hoßfeld, F. Metzger, P.E. Heegaard, Traffic modeling for aggregated periodic IoT data, in: 21st Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN, 2018.
[30] F. Wamser, P. Tran-Gia, S. Geißler, T. Hoßfeld, Modeling of traffic flows in internet of things using renewal approximation, in: Advances in Optimization and Decision Science for Society, Services and Enterprises: ODS, Genoa, Italy, September 4-7, 2019, Springer International Publishing, 2019.
[31] J. Xie, S. Zhang, M. Pan, Y. Yu, A comprehensive evaluation method for container auto-scaling algorithms on cloud, in: Computer Supported Cooperative Work and Social Computing, Springer Singapore, 2021.
[32] Kubernetes Documentation, Horizontal pod autoscaling, 2023, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[33] M. Straesser, S. Eismann, J. von Kistowski, A. Bauer, S. Kounev, Autoscaler evaluation and configuration: A practitioner's guideline, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, 2023.
[34] D. Mukherjee, S. Dhara, S.C. Borst, J.S. van Leeuwaarden, Optimal service elasticity in large-scale distributed systems, Proc. ACM Meas. Anal. Comput. Syst. (2017).
[35] Kubernetes Documentation, Resource management for pods and containers, 2024, https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.
[36] S. Geissler, S. Lange, G. Hasslinger, P. Tran-Gia, T. Hossfeld, Discrete-time analysis of multi-component queuing networks under renewal approximation, in: 34th International Teletraffic Congress, ITC 34, IEEE, 2022.
[37] S. Gebert, T. Zinner, S. Lange, C. Schwartz, P. Tran-Gia, Discrete-Time Analysis: Deriving the Distribution of the Number of Events in an Arbitrarily Distributed Interval, Tech. Rep., University of Wuerzburg, 2016.
[38] N.J. Gunther, UNIX load average part 1: How it works, Perform. Dyn. Company (2003).
[39] Kubernetes Documentation, Assign CPU resources to containers and pods, 2023, https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/.
[40] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, S. Kounev, Sizeless: predicting the optimal size of serverless functions, in: Proceedings of the 22nd International Middleware Conference, Association for Computing Machinery, 2021.
[41] Comscore Inc., 2017 U.S. cross-platform future in focus, 2017, https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/2017-US-Cross-Platform-Future-in-Focus.
[42] P. Leitner, J. Cito, Patterns in the chaos—A study of performance variation and predictability in public IaaS clouds, ACM Trans. Internet Technol. (2016).
[43] C. Laaber, J. Scheuner, P. Leitner, Software microbenchmarking in the cloud. How bad is it really? Empir. Softw. Eng. (2019).
[44] N.M. Calcavecchia, B.A. Caprarescu, E. Di Nitto, D.J. Dubois, D. Petcu, DEPAS: a decentralized probabilistic algorithm for auto-scaling, Computing (2012).
[45] B.A. Caprarescu, E. Kaslik, D. Petcu, Theoretical analysis and tuning of decentralized probabilistic auto-scaling, 2012, arXiv preprint arXiv:1202.2981.
[46] B.A. Caprarescu, D. Petcu, Decentralized probabilistic auto-scaling for heterogeneous systems, 2012, arXiv preprint arXiv:1203.3885.
[47] E. Park, K. Baek, E. Cho, I.-Y. Ko, Fully decentralized horizontal autoscaling for burst of load in fog computing, J. Web Eng. (2023).
[48] M. Koleini, C. Oviedo, D. McAuley, C. Rotsos, A. Madhavapeddy, T. Gazagnaire, M. Skejgstad, R. Mortier, Fractal: Automated application scaling, 2019, Preprint arXiv:1902.09636.
[49] A. Gujarati, S. Elnikety, Y. He, K.S. McKinley, B.B. Brandenburg, Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Association for Computing Machinery, 2017.
[50] Y. Desmouceaux, M. Enguehard, T.H. Clausen, Joint monitorless load-balancing and autoscaling for zero-wait-time in data centers, IEEE Trans. Netw. Serv. Manag. (2021).
[51] E.D. Nitto, L. Florio, D.A. Tamburri, Autonomic decentralized microservices: The gru approach and its evaluation, in: Microservices: Science and Engineering, Springer International Publishing, 2020.
[52] S.M.R. Nouri, H. Li, S. Venugopal, W. Guo, M. He, W. Tian, Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications, Future Gener. Comput. Syst. (2019).
[53] S. Merkouche, C. Bouanaka, A hybrid approach for containerized microservices auto-scaling, in: IEEE/ACS 19th International Conference on Computer Systems and Applications, AICCSA, 2022.

[54] M.M. Belkhiria, C. Tedeschi, Design and evaluation of decentralized scaling mechanisms for stream processing, in: IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2019.
[55] G. Russo, Towards decentralized auto-scaling policies for data stream processing applications., in: ZEUS, 2018.
[56] A. Mazidi, M. Golsorkhtabaramiri, M. Yadollahzadeh Tabari, An autonomic risk- and penalty-aware resource allocation with probabilistic resource scaling mechanism for multilayer cloud resource provisioning, Int. J. Commun. Syst. (2020).
[57] Z. Pan, Y. Wang, Y. Zhang, S.B. Yang, Y. Cheng, P. Chen, C. Guo, Q. Wen, X. Tian, Y. Dou, Z. Zhou, C. Yang, A. Zhou, B. Yang, MagicScaler: Uncertainty-aware, predictive autoscaling, Proc. VLDB Endow. (2023).
[58] S.N.A. Jawaddi, M.H. Johari, A. Ismail, A review of microservices autoscaling with formal verification perspective, Softw. - Pract. Exp. (2022).
[59] S.N. Agos Jawaddi, A. Ismail, M.N.H. Mohammad Hatta, A.F. Kamarulzaman, Insights into cloud autoscaling: a unique perspective through MDP and DTMC formal models, J. Supercomput. (2023).
[60] S. Burroughs, H. Dickel, M. van Zijl, V. Podolskiy, M. Gerndt, R. Malik, P. Patros, Towards autoscaling with guarantees on kubernetes clusters, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C, 2021.
[61] T. Tournaire, H. Castel-Taleb, E. Hyon, Efficient computation of optimal thresholds in cloud auto-scaling systems, ACM Trans. Model. Perform. Eval. Comput. Syst. (2023).
[62] S. Merkouche, C. Bouanaka, E. Benkhelifa, A Petri net-based formal modeling for microservices auto-scaling, in: 20th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA, 2023.
[63] A.V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, E. Elmroth, PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications, ACM Trans. Model. Perform. Eval. Comput. Syst. (2016).
[64] A. Ullah, T. Kiss, J. Kovács, F. Tusa, J. Deslauriers, H. Dagdeviren, R. Arjun, H. Hamzeh, Orchestration in the cloud-to-things compute continuum: taxonomy, survey and future directions, J. Cloud Comput. (2023).
[65] T. Kiss, A. Ullah, G. Terstyanszky, O. Kao, S. Becker, Y. Verginadis, A. Michalas, V. Stankovski, A. Kertesz, E. Ricci, et al., Swarmchestrate: Towards a fully decentralised framework for orchestrating applications in the cloud-to-edge continuum, in: International Conference on Advanced Information Networking and Applications, Springer, 2024.
[66] R.C. Sofia, J. Salomon, S. Ferlin-Reiter, L. Garcés-Erice, P. Urbanetz, H. Mueller, R. Touma, A. Espinosa, L.M. Contreras, V. Theodorou, N. Psaromanolakis, L. Mamatas, V. Tsaoussidis, X. Fu, T. Yuan, A. del Rio, D. Jiménez, A. Stam, E. Paraskevoulakou, P. Karamolegkos, V. Vieira, J. Martrat, I.M. Prusiel, D. Matzakou, J. Soldatos, D. Remon, M. Jahn, A framework for cognitive, decentralized container orchestration, IEEE Access (2024).
[67] O. Tomarchio, D. Calcaterra, G.D. Modica, Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks, J. Cloud Comput. (2020).
[68] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, G. Terstyanszky, MiCADO—Microservice-based cloud application-level dynamic orchestrator, Future Gener. Comput. Syst. (2019).
[69] J. Kovács, Supporting programmable autoscaling rules for containers and virtual machines on clouds, J. Grid Comput. (2019).
[70] N. Wang, B. Varghese, M. Matthaiou, D.S. Nikolopoulos, ENORM: A framework for edge node resource management, IEEE Trans. Serv. Comput. (2020).
[71] L.L. Jiménez, O. Schelén, HYDRA: Decentralized location-aware orchestration of containerized applications, IEEE Trans. Cloud Comput. (2022).
[72] S. Ilager, J. Fahringer, S.C.d.L. Dias, I. Brandic, DEMon: Decentralized monitoring for highly volatile edge environments, in: IEEE/ACM 15th International Conference on Utility and Cloud Computing, UCC, 2022.
[73] R.P. Centelles, M. Selimi, F. Freitag, L. Navarro, REDEMON: Resilient decentralized monitoring system for edge infrastructures, in: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID, 2020.
[74] M. Straesser, J. Mathiasch, A. Bauer, S. Kounev, A systematic approach for benchmarking of container orchestration frameworks, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2023.