This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3354062

Autoscaling Solutions for Cloud Applications under Dynamic Workloads

Giovanni Quattrocchi, Emilio Incerto, Riccardo Pinciroli, Catia Trubiani, Luciano Baresi

Abstract—Autoscaling systems provide means to automatically change the resources allocated to a software system according to the incoming workload and its actual needs. Public cloud providers offer a variety of autoscaling solutions, ranging from those based on user-written rules to more sophisticated ones. Originally, these solutions were conceived to manage clusters of virtual machines, while more recently, they have also been employed in the operation of containers. This paper analyses the autoscaling solutions provided by three major cloud providers, namely Amazon Web Services, Google Cloud Platform, and Microsoft Azure, and compares them against two solutions we develop based on control theory (*ScaleX*) and queuing theory (*QN-CTRL*). We evaluate the different approaches using both an in-house simulation engine and cloud deployments by feeding them with various synthetic and real-world workloads. Our extensive evaluation collects both simulation results and real measurements by which we can assess that both *ScaleX* and *QN-CTRL* outperform industrial techniques in most cases when considering the trade-offs between the service-level-agreement (SLA) violations and the optimal usage of resources.

Index Terms—autoscaling, elastic computing, cloud computing, containerization, containers, control theory, optimal control

1 Introduction

Software systems are increasingly sophisticated, and they frequently need to handle a wide range of dynamic workloads while maintaining a set service quality [1], [2]. As a result, system scalability is extremely important, and computational resources should be allocated as needed [3], [4], [5]. Provisioned resources should, ideally, match the intensity of dynamic workloads to be served and avoid both underprovisioning (i.e., resources are not enough to handle the workload) and overprovisioning (i.e., resources are more than needed) scenarios [6], [7].

Nowadays, cloud computing provides different means to change and provision resources to an application up to a theoretically infinite upper-bound [8]. These means lead to *autoscaling* systems. They were originally conceived for clusters of virtual machines (VMs), and users were in charge of defining the rules that change cluster sizes (i.e., horizontal scalability). VMs are "slow" to boot, scale, and manage, and thus limit the efficiency of autoscaling mechanisms that can only operate at low frequency. The autoscaling systems dedicated to VMs offered by Cloud providers are usually focused on horizontal scalability (adding/removing VMs), while their vertical scalability (resource configuration of an existing VM) is almost unexplored[1].

Compared to VMs, containers [9], i.e., a lightweight virtualization technology, are quicker to boot and manage. They provide means to quickly start new replicas in a few seconds, and to (partially) reconfigure provisioned resources at runtime in hundreds of milliseconds [10]. Containers can be run on bare-metal machines or on VMs to allow for multiple processes to share resources in a controlled way and without a significant overhead [11]. This motivates the usage of containers thus enabling faster and finer-grained autoscaling capabilities, compared to VMs, and allowing for the management of rapid changes and fluctuations of dynamic workloads [12], [13]. As for VMs, industrial approaches focus on the horizontal scalability of containers.

Public cloud providers allow users to directly manage both VMs and containers and also offer diverse autoscaling capabilities, either built-in or added by external orchestrators (e.g., Kubernetes [14]). Although these solutions are largely adopted by industry [15], they are based on rules that are defined by users or heuristics and they may encounter sub-optimality (i.e., violation of requirements due to under-provisioning, or high costs for over-provisioning).

This paper analyses these autoscaling solutions and focuses on those provided by the three major cloud providers: Google Cloud Platform, Amazon Web Services (AWS), and Microsoft Azure. In a continuous effort to foster the usage of theoretical-based autoscaling solutions, we propose an extensive comparison among industrial autoscalers and two novel approaches that we build on top of our previous work: *ScaleX* (initially conceived in [16], and extended here), which is based on control theory [17], and *QN-CTRL* (based on a symbolic solution of software performance models [18], and newly combined with optimal control in this manuscript), which is based on queuing theory [19]. While industrial solutions and most of the work in the literature focus on the horizontal scalability of VMs and containers, the main novelty of our approaches is that they exploit vertical scaling to allow containers to reconfigure allocated resources almost instantaneously [10].

- *G. Quattrocchi and L. Baresi are with Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy. E-mail: {name.surname}@polimi.it*
- *E. Incerto is with IMT School For Advanced Studies Lucca, Lucca, Italy E-mail: emilio.incerto@imtlucca.it*
- *R. Pinciroli and C. Trubiani are with Gran Sasso Science Institute (GSSI), L'Aquila, Italy. E-mail:{name.surname}@gssi.it*

1. To the best of our knowledge, Google and Amazon Web Services do not offer means to vertically scale VMs. Microsoft Azure supports this feature, but modifications must be applied to a pool of VMs and changes must be applied to all the VMs at the same time.

Name	Providers	Description
Rule-based	*AWS, Azure*	A solution that relies on user-defined rules, triggering scaling actions when set conditions are met, such as when CPU utilization crosses a certain threshold.
Step	*AWS*	An evolution of rule-based solutions, enabling users to associate specific conditions with multiple scaling actions (or steps), thereby offering more fine-grained control over resource allocation.
Target	*AWS, GCP*	Advanced scaling solution that automatically maintains a target metric value by adding/removing instances without requiring users to write complex rules or policies.
Others	*AWS, Azure, GCP*	Proactive approaches that are complementary to rule-based, step, and target solutions. They work on larger time spans and may involve long-lasting pattern analysis of the workload.

TABLE 1: Summary of the main autoscaling solutions provided by popular public cloud providers.

Our empirical evaluation, carried out on both simulated and cloud-based environments, shows how the proposed techniques are able to outperform industry solutions by both minimizing response time violations and allocating fewer resources under a variety of synthetic and real-world workloads. The comparison also includes an in-depth sensitivity analysis of *ScaleX* and *QN-CTRL*, which allows highlighting of important trade-offs.

To summarize the contributions of this article are i) a survey of the existing, industry-provided autoscaling mechanisms, ii) two novel autoscaling solutions, i.e., *ScaleX* and *QN-CTRL*, iii) an open-source and extendable simulation engine for autoscaling mechanisms, and iv) an extensive comparison of the different solutions based on the aforementioned simulator and on executions in the AWS cloud.

The rest of the manuscript is organized as follows. Section 2 discusses industrial solutions. Sections 3 and 4 provide details on *ScaleX* and *QN-CTRL*, respectively. Section 5 presents simulation- and cloud-based experiments, and reports on the assessment we carried out. Section 6 surveys related work and Section 7 concludes the article with future research directions.

2 Industrial Approaches

This section provides an overview of the main autoscaling systems available on cloud providers. We consider the solutions implemented in Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure, three of the main public cloud providers. Table 1 summarizes the main solutions and highlights their characteristics.

These systems exploit the horizontal scalability of either VMs or containers, while the vertical scaling in industrial approaches is still in its early days.

The control period of an autoscaling system is capped by the speed of actuation, i.e., the time it takes to complete a scaling action. While VMs take around one minute to be fully started or terminated in the three cloud providers analyzed, containers are faster and are bootstrapped in a few seconds. Re-configuring a container using vertical scalability is almost instantaneous (hundreds of milliseconds on average) [10].

Rule-based. AWS and Azure provide autoscaling mechanisms based on rules, known as *simple scaling* in AWS [20] and *AutoScale* in Azure [21]. Typically users define a constraint on a given metric (e.g., CPU utilization $< 80\%$) and, when it is violated, a scaling action is triggered (e.g., adding a VM). In AWS this approach is based on *CloudWatch*, i.e., the AWS monitoring system. CloudWatch allows users to observe the fluctuations of a set metric. Metrics in CloudWatch are either infrastructural (e.g., CPU utilization) or custom and provided by the user. Metrics are continuously collected and a data point is created by aggregating over a time window (usually 1 or 2 minutes). CloudWatch provides means for defining alarms when a threshold of the set metric is reached. Similarly, Azure Monitoring allows users to observe different metrics and trigger alarms.

When the alert is triggered, a so-called *policy* is executed. A policy can add or remove a given amount of VMs (e.g., $+3$ VMs) or proportionally to the current size (e.g., $+20\%$ of running VMs).

When an action is triggered, the autoscaling mechanism remains idle for a so-called *cool down period* (default is 180 seconds) so that the system stabilizes after executing the action. This way, different and possibly conflicting actions do not overlap.

In addition to the rule-based approach available for VMs, AWS provides a similar autoscaling system for containers in its dedicated container-as-a-service solution (ECS). Instead, other providers rely on Kubernetes for container orchestration which uses a different type of logic (i.e., *target*) for scaling containers, as explained in the following.

Step. Step scaling is provided by AWS [20] and it is an evolution of the rule-based approach. This mechanism allows users to provide a policy table to define the scaling actions to be carried out when certain conditions are met. In particular, for a CloudWatch alert, users define the amount or percentage of VMs/containers that should be added or removed when differences in metrics are observed. For example, Table 2 shows how a user can specify step scaling rules based on *CPU utilization* (i.e., metric U_{CPU}).

Step scaling requires the definition of a *warm-up time* (and not a cool-down period), i.e., the time to wait before considering a VM launched or terminated. This way, multiple policies can be executed consecutively, but their effect is taken into account only after a set amount of time.

Target. AWS [22] and GCP [23] offer a more advanced scaling technique that does not require users to write "complex" inputs (such as rules or policies) but only needs to define a target value for a given metric. In this case, the goal of the autoscaling system is to keep the metric as close as possible to the target value by adding/removing VM

Values	Action
$U_{CPU} > 90\%$	$+20\%$ *VMs*
$80\% < U_{CPU} \leq 90\%$	$+15\%$ *VMs*
$70\% < U_{CPU} \leq 80\%$	$+10\%$ *VMs*
$20\% < U_{CPU} \leq 30\%$	-20% *VMs*
$30\% < U_{CPU} \leq 40\%$	-15% *VMs*
$40\% < U_{CPU} \leq 50\%$	-10% *VMs*

TABLE 2: Examples of step scaling rules.

instances. Both AWS and GCP do not disclose the algorithm behind this technique. Cool-down and warm-up periods are used to keep the system stable and to avoid overlaps in action executions.

A similar approach is provided by Kubernetes, a popular container orchestrator, that is offered as-a-service by AWS, GCP, and Azure. Kubernetes uses target scaling for managing container instances through a dedicated component called Horizontal Pod Autoscaler (HPA) [24].

HPA computes the number of container instances (ci) needed to meet the target value (tm) for a given metric. ci is computed proportionally to the ratio between the current monitored metric value (cm) and tm: $ci = ceil(ci * cm/tm)$.

The new replicas are created (if needed) only if cm/tm is greater than a given threshold (default value is 0.1). Containers are much faster to be booted and managed compared to VMs, thus, HPA computes the new allocation every 30 seconds (control period). A so-called *stabilization period* (default 5 minutes) is waited after a new value of ci is computed and enacted.

Kubernetes also offers a similar approach based on vertical scaling (VPA) [25] of CPU and memory. This approach is still in beta[2] and does not provide in-place vertical scaling since containers must be restarted to be reconfigured. Containers are also not scaled independently and all the replicas must be reconfigured in the same way. Moreover, the integration with HPA and JVM-based containers is not fully supported.

Others. AWS [26], Azure [27] and GCP [28] provide *scheduled scaling*, that is the ability to set time triggers for executing a scaling action. For example, users can schedule to increase VMs during the weekend and decrease them on Monday. *Predictive scaling*, provided by AWS [29], is an evolution of this approach: it analyzes the workload patterns of the previous 14 days, calculates the required VM allocation for the next 2 days, and automatically computes scheduled scaling actions accordingly. The algorithm behind this approach is not disclosed. Scheduled and predictive scheduling can be seen as complementary *proactive* approaches to rule, step, and target scaling (reactive). The former produces long/mid-term actions, while the latter can quickly respond to unexpected changes in the execution environment. In the rest of this work, we focus on reactive approaches being them necessary when the execution environment (including the dynamicity of workloads) is not fully predictable.

3 *ScaleX*

ScaleX is our control theoretical autoscaling system. Its lightweight, hierarchical architecture allows one to manage container-based systems with extremely fast control loops by exploiting in-place vertical scaling which is the ability to reconfigure computing resources allocated to a system without the need to add, remove, or restart any component.

The scaling mechanism of containers is based on a feature of the Linux kernel called *cgroups*, which allows for (re)configuring at runtime CPU cores and memory. Cores can be allocated using either CPU *shares*, CPU *reservation*, or CPU *quotas*. Shares enforce a limit to the CPU allocated to each container only in case of resource contention, while reservations let users pin single cores to a given container (exclusive use). Instead, *ScaleX* uses quotas since they set a hard limit on resource usage and cores can be allocated with decimal precision. Memory can be allocated to containers in a hard or soft way. The former sets a strict upper bound to usable memory; the latter gives more freedom.

ScaleX unlocks the speed of container management by featuring a lightweight design. Each container is managed by an independent controller that continuously reconfigures allocated resources. Given that *ScaleX* supports multiple containers (i.e., different applications) running on the same machine, the sum of allocated resources may exceed available capacity. *ScaleX* does not employ any synchronization mechanism but provides an additional heuristic-based controller, i.e., the *Supervisor*, deployed on each machine. This component gathers the allocations computed by the controllers, and before enacting them, it scales them (if needed) proportionally to prevent resource contention. The Supervisor can be customized to accommodate non-proportional heuristics so that the most demanding applications can be prioritized in case of contention.

ScaleX uses lightweight Proportional-Integer (PI) controllers that are able to compute the next state of the system (i.e., the next resource allocation) in constant time, allowing for the aforementioned fast control period. Our PI controllers are grey-box meaning that they embed a characteristic function describing the dynamics of the controlled system. This function captures only the main behavior of the system and we rely on the PI feedback loop for runtime adjustments.

We have developed PI controllers for microservices [30], serverless functions [31], big-data batch applications [13], and GPU-accelerated machine learning applications [32]. The goal of all of these controllers is to guarantee that the response time does not exceed a set threshold while efficiently allocating resources. Each system shares the control architecture presented in Section 3.1 and, at the same time, considers the unique characteristics of each system. For example, in our work related to big-data applications, the response time is not computed as an aggregation of several (almost instantaneous) requests but, since execution may last minutes or hours, each single request is controlled and the requirement on its response time is considered a "deadline", that is the maximum allowed time to finish that specific execution.

Although the version of *ScaleX* presented in this paper focuses on vertical autoscaling only, it can easily interface with horizontal autoscaling mechanisms. Indeed, while each PI controller manages a single existing container, *ScaleX* computes significant indicators of the performance of each component. These values can be used by horizontal autoscaling systems (e.g., Kubernetes HPA) for allocating more containers and/or machines.

Unlike heuristic-based approaches (e.g., all the industrial ones described in Section 2), control-theory provides some key formal guarantees on the control carried out [17], [33]. In particular, *stability* defines the ability of the system under control to reach a set point and to remain in its neighborhood. *Settling time* describes the pace of the system in converging to a stable point. *Maximum overshooting* captures

2. https://github.com/kubernetes/autoscaler

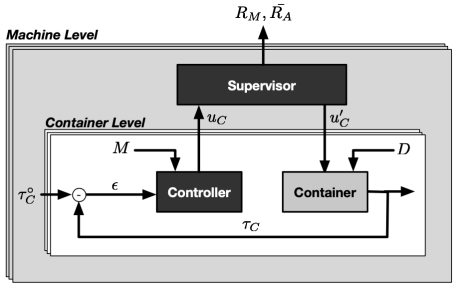


Fig. 1: Control architecture of *ScaleX*.

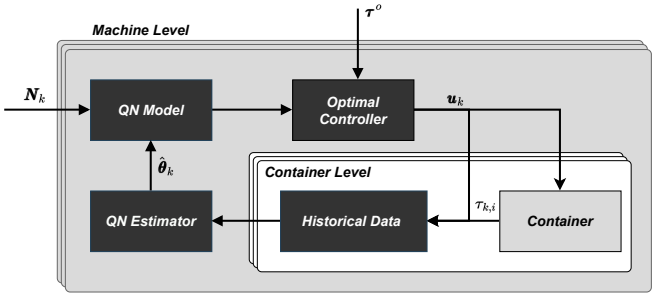


Fig. 2: Control architecture of *QN-CTRL*.

the maximum difference between the set-point and the measured controlled variable (e.g., the monitored response time). Finally, *steady-state error* is the difference between the value reached at steady-state and the set-point.

3.1 Control Architecture

Figure 1 depicts the control architecture of *ScaleX*. Each container C (the controlled system) is deployed along with a dedicated PI controller given the defined application type. For each application/container, users provide a set-point, that is a target response time (τ_C°), similarly to the *target* controllers described in Section 2.

At runtime, unknown disturbances (D) affect the container's response time (τ_C), which is monitored together with other metrics (M). At each control step (1 second), the controller computes the error ϵ, defined as the difference between τ_C° and τ_C. Then it computes a resource allocation for the container u_C by using the error and M so that, ideally, $\tau_C = \tau_C^\circ$.

Each controller on a machine m transfers to the *Supervisor* the computed u_C that aggregates all the allocations in a vector[3] $\bar{U}_m$, and, if needed, computes a feasible resource allocation u'_C for each container according to a specified policy (e.g., proportional, priority-based, requirement-based). Moreover, if the sum of to-be-allocated resources is lower than the capacity of the machine, the supervisor can (optionally) scale up the allocations to speed up applications' performance at the expense of a sub-optimal allocation (over-provisioning).

Before enacting the computed u'_C on each container, the supervisor produces two indicators: R_M and $\bar{R}_A$, that are, respectively, the saturation level of the machine and the needs of each application running on it.

The complexity of the control loop is constant (in time and space), and no synchronization among different controllers is necessary thanks to the highly distributed and hierarchical architecture of *ScaleX*. This is key to allow for very short control periods (e.g., 1 second) even when controlling large-scale systems.

4 *QN-CTRL*

QN-CTRL is the novel optimal autoscaler for containerized applications that we propose in this paper. Its objective is similar to that of *ScaleX*, i.e., to determine the computational resources required by application containers for meeting performance requirements under highly variable workloads. As the objective of the two controllers is similar, so is how they interact with the controlled containers. After the computation of each new allocation, *QN-CTRL* translates it into proper actuation signals via the Linux *cgroups* interface (i.e., through the period and the quota parameters) or the chosen container engine (e.g., Docker). *QN-CTRL* is based on two main components: i) an automatically generated queuing network (QN) model of the system, ii) an efficient optimization problem. At each control loop, the optimization problem is solved to compute the minimum amount of CPU quota required by the container to reach the desired service level objective under the measured load. In addition, the QN model is updated in a moving horizon estimation fashion, taking into account past measurements of response time and the system load (the output and input from the previous phase, respectively). By doing so, *QN-CTRL* keeps its performance model always up-to-date with the recent behavior of the system.

The main difference between *ScaleX* and *QN-CTRL* is in the architecture of their control loops. As presented in Section 3, *ScaleX* is hierarchical: each container is equipped with an independent PI controller and a supervisor ensuring compliance with the computational limits of the platform. This way, *ScaleX* prioritizes controller efficiency (i.e., algorithmic complexity) over the quality of the computed allocations. In contrast, *QN-CTRL* uses a centralized approach and calculates the allocation for all containers simultaneously through a single optimization problem that also takes into account the constraints imposed by the platform. As a result, *QN-CTRL* favors the precision of the allocations over their computational complexity. However, the formulation of the control and estimation problem as a nonlinear local optimization makes *QN-CTRL* very efficient even for hundreds of variables and constraints [34] (i.e., in the experimentation of Section 5, *QN-CTRL* takes few milliseconds for both the model estimation and optimal allocation computation). In the following, we detail *QN-CTRL* and its estimator.

4.1 Control Architecture

Figure 2 depicts the architecture of *QN-CTRL* on a machine with M containers where, $\boldsymbol{N}_k = (N_{k,i})_{1 \le i \le M}$ is the vector of sampled load (i.e., number of users) for each container i at time k, $\boldsymbol{\tau}^o = (\tau_i^o)_{1 \le i \le M}$ is the vector of desired service level objective, $\boldsymbol{u}_k = (u_{k,i})_{1 \le i \le M}$ is the vector of optimal allocation, $\boldsymbol{\tau}_k = (\tau_{k,i})_{1 \le i \le M}$ is the vector of measured

3. A variable super-scripted with a bar is a vector.

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3354062

response time and $\hat{\boldsymbol{\theta}}_k = (\hat{\theta}_{k,i})_{1 \leq i \leq M}$ is the vector of the QN's model parameters for container i at time instant k.

The key ingredient of *QN-CTRL* is an efficient steady-state solution [18], [35] of a closed QN model [19]. It allows predicting the system's key performance indices (i.e., response time, throughput, and utilization) as a function of the number of active users, CPU cores, and container's service time (i.e., the average time required for the container to execute a request when there are no queuing phenomena). Then the QN solution is encoded in a nonlinear optimization problem (i.e., the optimal controller of Figure 2) that is solved at runtime to compute the CPU capacity (i.e., $\boldsymbol{u}_k$). Such capacity value is required by the controlled container to achieve the desired performance objective (i.e., the user-defined target response time $\boldsymbol{\tau}^o$). Finally, to create an optimal autoscaler that is effective and easy to use in practice, we equipped *QN-CTRL* with a QN model estimator (the QN estimator box of Figure 2) suitable to calibrate the performance model. The technical details of the different components of *QN-CTRL* are discussed in the following.

4.2 Efficient Steady-state Solution of Closed QNs

QNs are a family of stochastic models widely used in software performance engineering [36], [37]. Their essential idea is to model the traffic of customers/jobs that are routed across different services (i.e., stations) and compete for a limited pool of computational resources. To model parallelism, each station is composed of multiple, independent, and identical servers (e.g., the thread concurrency levels for software resources or multiple cores for hardware resources). Service times are described by a probability distribution, where a commonly used one is the exponential one. In any case, the proposed optimal control approach is easily extensible to a more general class of service time distributions (i.e., the Coxian one that can approximate any given general distribution arbitrarily closely [38]), as already partially done in [18]. A QN may be closed or open depending on whether or not a fixed population of customers remains within the system. In this work, we focus on closed QNs with a multiple class of users, i.e., with each class, we model the performance of co-located containers and how they affect each other. Finally, a multi-class think station is used to model the incoming request rate to the services provided by the system.

Following the mathematical notation already adopted in [18], [39], [40], a QN is formally specified by a set of multi-class think and service stations denoted by $N = \{0, 1, \ldots, n\}$ and by the following parameters:

- $\mu_{i,c}$ denotes the server multiplicity (i.e., number of CPU cores) assigned to class c at the i-th station, with $i \in N$; the total number of cores available in station i is equal to $\sum_{c=1}^{M} \mu_{i,c}$ where M is the number of classes deployed on that station (e.g., the number of containers sharing the same machines).
- $e_{i,c}$ is the average value of the exponentially distributed service time of service class c at the i-th station, with $i \in S$;
- $\mathcal{C}_c$ is the total number of clients of class c simultaneously interacting with the system.

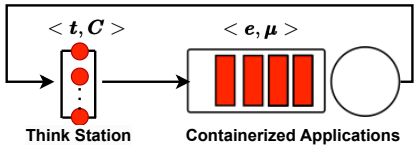


Fig. 3: QN model of M containerized application within *QN-CTRL* autoscaler. We denotes with $\boldsymbol{t} = (t_c)_{1 \leq c \leq M}$, $\boldsymbol{e} = (e_c)_{1 \leq c \leq M}$, $\boldsymbol{\mu} = (\mu_c)_{1 \leq c \leq M}$, $\boldsymbol{\mathcal{C}} = (\mathcal{C}_c)_{1 \leq c \leq M}$ the vectors of think times, service times, CPU quotas, and class users, respectively.

- t_c is the average value of the exponentially distributed think time between two subsequent requests to service class c;

The performance dynamics of M co-located containerized applications within *QN-CTRL* is modeled by following the scheme depicted in Figure 3. In particular, we model the different applications through a single multi-class station (i.e., with M classes) that represents the computational load induced by the corresponding users (i.e., $\mathcal{C}_c$). Since we adopted a closed QN model, upon completion, a user waits for an exponentially distributed delay before submitting a new request to the system. Section 5 will show how this modeling choice allows to effectively control real-world applications. Indeed, the reported numerical results demonstrate the benefit of the QN estimator in tuning the model at runtime to resemble the performance behavior of different co-located applications with high fidelity.

Grounded on the theoretical foundation reported in [35], the equations that allow computing the key performance indices (i.e., response time and throughput) of the QN model described in Figure 3 are defined. Eq. (1) reports the mathematical expressions for the steady-state (i.e., in the long-run) performance indices of service class c:

$$\mathcal{X}_c = \min\left(\frac{C_c}{t_c + e_c}, \frac{\mu_c}{e_c}\right) \quad \text{and} \quad \mathcal{T}_c = \frac{\mathcal{C}_c}{\mathcal{X}_c}. \tag{1}$$

As expected, the throughput of the c-th class, i.e., $\mathcal{X}_c$ in Eq. (1), is strongly influenced by the number of cores available for that application and by the number of its clients $\mathcal{C}_c$. Finally, the steady-state response time (i.e., $\mathcal{T}_c$) can be calculated through Little's law [19] which relates the number of clients with the corresponding throughput. In the following, we use these equations to create the *QN-CTRL* optimal autoscaler and the QN Estimator.

4.3 Autoscaler Formulation

The definition of the *QN-CTRL* optimal autoscaler goes through the encoding of Eq. (1) in a nonlinear optimization problem able to compute, at runtime, the number of CPU cores (i.e., $u_c, 1 \leq c \leq M$) needed to achieve the desired value of response time (i.e., $\mathcal{T}_c^o$) for service c when $\mathcal{C}_c$ clients are simultaneously interacting with the system. Eqs. (2)–(6) depicts the formulation of the aforementioned optimization problem.

$$\underset{u_c}{\text{minimize}} \quad \sum_{c=1}^{C} |\mathcal{T}_c - \mathcal{T}_c^o| \tag{2}$$

subject to:

$$\mathcal{X}_c = \min(\frac{C_c}{t_c + e_c}, \frac{\mu_c}{e_c}) \tag{3}$$

$$\mathcal{T}_c = \frac{\mathcal{C}_c}{\mathcal{X}_c} \tag{4}$$

$$0 \leq u_c \leq \bar{u}_c \tag{5}$$

with $1 \leq c \leq M$

$$\sum_{c=1}^{M} u_{i,c} \leq \bar{u} \tag{6}$$

In particular, through Eqs. (3) and (4), we encode the performance behavior of all the controlled containers as a constraint of the optimization problem. With the objective function, i.e., Eq. (2), we drive the selection of the number of servers to ensure that the application's response time is as close as possible to the desired one, i.e., by minimizing the absolute value of the error between the reference value $\mathcal{T}_c^o$ and the one computed by Eq. (4) while minimizing the total number of allocated cores. Equation (6) concludes *QN-CTRL* optimal autoscaler by enforcing that the sum of the CPU quotas assigned to each container (i.e., to each service class c) is less or equal to the available CPU cores on the hosting machine (i.e., $\bar{u}$). We remark that to make the optimization problem as efficient as possible, in the experimentation of Section 5, we substitute the minimum function of Eq. (3) with its smooth approximation [41].

4.4 QN Estimator formulation

$$\underset{e_c}{\text{minimize}} \quad \sum_{i=1}^{W} |\mathcal{T}_{i,c} - \mathcal{T}_{i,c}^m| \tag{7}$$

subject to:

$$\mathcal{X}_{i,c} = \min(\frac{C_{i,c}}{t_{i,c} + e_c}, \frac{\mu_{i,c}}{e_c}), \quad 1 \leq i \leq W \tag{8}$$

$$\mathcal{T}_{i,c} = \frac{\mathcal{C}_c}{\mathcal{X}_{i,c}}, \quad 1 \leq i \leq W \tag{9}$$

The QN Estimator is defined as an optimization problem, i.e., via the encoding of Eq. (1). However, unlike Eqs. (2)–(5), the inputs and outputs are reversed, i.e., the number of cores u_c is a measured variable while the service time (i.e., e_c) is the decision variable. Indeed, the goal of the estimator is to compute the parameters of the QN model of Figure 3 starting from the last W samples of response times, number of servers, and number of clients (i.e., the QN model used by *QN-CTRL* at control step t is estimated by using the information collected in control steps from $t - W$ to $t - 1$). For estimating the think times $t_{i,c}$ we considered the difference between the completion and arrival times of two consecutive requests for service class c made by the same user. Considering that for estimation purposes the optimization problem of each container can be solved in parallel (i.e., no joint constraints are enforced), below we present the estimator focusing on the single container c. The extension to the multi-container case is achieved by duplicating M times the optimization problem (7)–(9) solved with the data corresponding to the container under estimation.

Eqs. (8) and (9) encode the response time of the QN model to be estimated as a function of its unknown service time (for doing so, we use the last W measured number of cores u_i and number of clients $\mathcal{C}_i$ with $t - W \leq i \leq t - 1$). The estimator is completed by the objective function (7), which steers the service time to be the one that minimizes the absolute error between the response times calculated in (8)–(9) and the measured ones (i.e., $\mathcal{T}_{i,c}^m$). The accuracy of our QN estimator is evaluated in terms of the achieved SLA, as usually done for tuning classical controllers [42]. Section 5 reports a similar strategy to determine the *ScaleX* parameters used in the numerical evaluation, and *QN-CTRL* provides evidence of satisfactory results across different applications and workloads.

Combining the techniques presented in Sections 4.3 and 4.4, we formulate an efficient optimal autoscaler that adapts to different applications by estimating the corresponding QN-based performance model at runtime.

5 Evaluation

This section presents a comprehensive comparison of different autoscaling solutions for containerized applications performed in simulated and cloud environments (AWS cloud). Simulation provides a controlled environment tailored to application service time and control signal actuation. Cloud-based experiments, instead, allow one to assess techniques with applications executed in realistic environments. In both cases, we compare the industrial approaches presented in Section 2 against *ScaleX* and *QN-CTRL* whose implementation is discussed in the following.

Autoscaling Systems. We analyze and compare the number of violations and allocated cores of 8 autoscaling solutions dedicated to containers.

- *Static (1)* —available in *AWS, GCP,* and *Azure*: we only allocated one core independently of the observed workload and its intensity;
- *Rule-based (+1)* —available in *AWS* and *Azure*: We define the following rules:

response time $> 0.9 * SLA$	$+1$ *core*
response time $< 0.5 * SLA$	-1 *core*

- *Rule-based (+3)* —available in *AWS* and *Azure*: We defined rules similar to those above, but we add/remove 3 cores when triggered;
- *Step* —available in *AWS*: We defined the following steps:

response time $\geq 1.3 * SLA$	$+30\%$ cores
$SLA \leq$ response time $< 1.1 * SLA$	$+20\%$ cores
$0.9 * SLA \leq$ response time $< SLA$	$+10\%$ cores
response time $< 0.8 * SLA$	-10% cores

- *Target* —available in *AWS* and *GCP*: We exploited Kubernetes' HPA (the only available algorithm);

- *TargetFast* —available in *AWS* and *GCP*: We started for the above implementation and adopted a faster control period;
- *ScaleX*, the solution presented in Section 3;
- *QN-CTRL*, the approach presented in Section 4.

As explained in Section 2 most of the available techniques exploit horizontal scalability that requires longer actuation times compared to vertical scaling (employed by *ScaleX* and *QN-CTRL*). For this reason, we configure all the industrial approaches, except *TargetFast*, with a control period of 30 seconds, i.e., the default value of multiple solutions (e.g., Kubernetes HPA), while *ScaleX* and *QN-CTRL* are set with a control period of 1 second to properly exploit vertical scaling capabilities. Being *Target* the most evolved and wide-spread solution among industrial approaches, we also considered a "fast" version of the approach (control period 1 second), namely *TargetFast*, that can be employed to scale containers vertically (e.g., Kubernetes VPA). All industrial approaches are configured with no cool-down and warm-up periods so that they are able to scale up and -down resources as fast as possible since we assume an immediate actuation (e.g., in-place reconfiguration of containers).

5.1 Simulation-based experiments

To enable and ease the comparison of different autoscaling solutions in a controlled environment, we developed a simulator called *RAS*[4] (Resource Allocation Simulator).

RAS is a lightweight simulation environment written in Python, that allows mocking different autoscaling solutions, workloads, and applications. It comes with a library of existing components but developers can easily customize them and conceive new ones. In particular, RAS provides three main interfaces, *Generator*, *Controller*, and *Application*, plus a *Runner* class. The *Generator* aims to abstract the workloads used for the experimentation (i.e., they map a number of users to each simulated time-instant according to the chosen load function). The *Controller* exposes a control method that returns a CPU allocation for each control time instant and workload intensity. The *Application* interface abstracts the logic necessary to calculate the application response time as a function of the number of users and the cores computed by an instance of *Controller*. The *Runner* class coordinates the components of *RAS*: it determines the progress of the simulation, generates the control events, and collects the measurement points. Thanks to this class hierarchy, new autoscaling scenarios are easily implemented by providing the appropriate concrete implementation of the interfaces mentioned above and by using the runner provided by *RAS*. Applications, workloads, and autoscaling solutions described in the following are implemented with these means.

Applications. We compare the autoscaling solutions using two applications that are based on two different response time models. These models assume: (i) the behavior (i.e., response time) of the considered system depends on the load intensity (i.e., number of requests to process) and on the number of allocated resources (i.e., CPUs or cores); (ii) other system resources (e.g., memory) are scaled proportionally to CPUs and are always sufficient when CPU-bound applications are considered, as in our case.

The model in [30] (hereinafter *AppDet*) considers requests with a deterministic service time and defines the response time as a function that decreases monotonically as the number of cores increases, whose minimum value is constrained by a lower bound to which the response time tends when all requests of the application are served concurrently by the available cores. The model defines the response time as:

$$RT = (1+\nu) \cdot \frac{(c_1 + c_2) \cdot req + c_1 \cdot c_3 \cdot cores}{req + c_3 \cdot cores}, \qquad (10)$$

where req is the number of requests to serve, $cores$ is the number of available CPUs to process the requests, and c_1, c_2, c_3 are obtained through profiling. For this evaluation we used $c_1 = 0.00763$, $c_2 = 0.0018$, and $c_3 = 0.5658$ as in [30]. We add a noise, $\nu \sim \mathit{Unif}(-0.1, 0.1)$, i.e., $\pm 10\%$ disturbance, to the model presented in [30] to consider interference among requests.

The other model (hereinafter *AppExp*) is a recursive technique (Mean Value Analysis, MVA [43]) that analytically derives the performance metrics (e.g., average response time) of a closed QN with a low computational cost [44] and assumes all requests are processed with an exponential service time. MVA and its approximations are widely used in the literature to study the performance of various real-world systems, e.g., Industry 4.0 warehouse automation [45], Apache Cassandra [46], cloud applications [47].

Workloads. Autoscaling solutions are evaluated with synthetic and real-world workloads whose characteristics are summarized in Table 3. The load intensity (i.e., the number of requests to process) depends on time, $req(t)$, and is defined by different workload types. *Sine* (SN1 and SN2) is a periodic workload that follows a sine function; *Stair* (ST1 and ST2) is a workload whose intensity varies between two values (e.g., low and high) instantaneously (impulse like); *Ramp* (RP1 and RP2) increases/decreases linearly up to a given value where it stabilizes. These are synthetic workloads whose shapes are typically used to test (and stress) control systems at design time; they are meant to capture the main characteristics of real-world workloads (e.g., a Stair workload captures an abrupt increase in the system load) [48], [49]. *Twitter* (TW) is a publicly available real-world trace [50] that collects the number of tweets submitted every second on January 1st, 2021. It is used in the literature, e.g., to study the performance of frameworks for machine learning inference requests [51].

Service-level Agreement (SLA). To analyze and compare the autoscaling solutions, we set the SLA to 0.6 seconds, i.e., requests must be served within the given time to avoid SLA violations. For target/set-point based solution, a common strategy to make the system meet the SLA is to use a target response time lower than the SLA [52]:

$$\mathit{TargetRT} = \alpha \cdot \mathrm{SLA}, \quad \text{with } 0 < \alpha \leq 1. \qquad (11)$$

4. The source code and the replication data of the simulated and cloud-based experiments are available at https://github.com/deib-polimi/RAS and https://github.com/deib-polimi/RAS-real, respectively.

Name	Type	Description
SN1	Sine	$req(t) = 500 * sin(t\frac{\pi}{100}) + 700$
SN2	Sine	$req(t) = 1000 * sin(t\frac{\pi}{50}) + 1000$
ST1	Stair	$req(t) = 1000 * (1 + floor(t/100))$
ST2	Stair	$req(t) = 5000$ if $50 \leq t < 800$ else $req(t) = 50$
RP1	Ramp	$req(t) = 10t$ if $t < 800$ else $req(t) = 8000$
RP2	Ramp	$req(t) = 20t$ if $t < 800$ else $req(t) = 16000$
TW	Twitter	Real trace collected on January 1st, 2021 [50]

TABLE 3: Workloads used to evaluate autoscaling solutions in simulated experiments.

For these experiments, $\alpha = 0.8$ and $TargetRT = 0.48$.

5.1.1 Results

For each considered autoscaling approach, we present the number of SLA violations and the average number of cores allocated to all containers. All autoscaling solutions control the system for 1000 seconds. SLA violations and the number of allocated cores are monitored at the end of each control window with a different time horizon depending on the controller being evaluated. Specifically, for TargetFast, *ScaleX*, and *QN-CTRL* data are averaged over the last second (i.e., their control period), while results of all other approaches are carried out on the previous 30 seconds.

AppDet. Table 4 shows SLA violations (V) and the number of allocated cores (A_μ) of the 8 autoscaling strategies (Approach) under different workloads (W), when they control the AppDet application. Response time statistics (i.e., average, μ, standard deviation, σ, minimum value, m, and maximum value, M) are computed by observing the response time of each control period. For each workload, we highlight (in bold) the *best* autoscaling strategy, i.e., *the one with the lowest number of SLA violations and, in the event of a draw, with the lowest number of allocated cores.*

With AppDet, *ScaleX* is the autoscaling approach that performs better except when used to control the ST2 workload. Generally, *ScaleX* does not violate the SLA and is the solution with the smallest number of allocated cores among strategies with the same number of SLA violations. *Static (1)* is the autoscaling approach with the largest number of violations and in some cases (i.e., ST1 and TW) it serves no requests within the SLA. Intuitively, the constant allocation provided by *Static (1)* cannot properly handle highly dynamic and fluctuating workloads. When the system is controlled by *QN-CTRL* only a few violations are observed and the number of allocated cores is comparable to the one of *ScaleX*. *QN-CTRL* performs particularly badly (i.e., 62 violations) with the RP1 workload due to the initial small number of requests in the system that does not facilitate the learning process on which the approach relies [53]. Instead, *QN-CTRL* is better than *ScaleX* when it is used to control the ST2 workload. Although the initial number of requests is still small (i.e., only 50 requests, see Table 3), the workload grows soon to a large value. As discussed in Section 4, *QN-CTRL* is a technique based on differential equations and works better (i.e., its predictions are more accurate) when the number of requests increases.

Figures 4(c) and 4(e) show the response times and allocated resources obtained by *ScaleX*, *QN-CTRL*, and the best performing industrial solution when controlling SN1 plotted in Figure 4(a). *ScaleX*, *QN-CTRL*, and *TargetFast* do not violate the SLA. *TargetFast* is generally far from the SLA since it allocates more resources with respect to our approaches. *TargetFast* provides an allocation that is only proportional to the mismatch between the target response time and its monitored value (i.e., the error). This approach is more sensible to noise compared to our solutions (e.g., *ScaleX* embeds an integral factor to mitigate overshooting) and leads to a more unstable system. This is also clearly observable in the standard deviation of response times: 0.40 of *TargetFast* against 0.21 of *ScaleX* and 0.05 of *QN-CTRL*.

QN-CTRL and *ScaleX* are close to the SLA (they meet well the specified target, i.e., 0.48 seconds), but *QN-CTRL* initially uses more cores than *ScaleX* to serve the requests (i.e., due to the inaccuracy of the underlying performance model in the early phases of calibration).

		RT					
W	**Approach**	μ	σ	m	M	V	A_μ
SN1	*Static (1)*	0.98	0.27	0.50	1.31	846	1.00
	Rule-based	0.38	0.16	0.12	0.75	91	4.93
	Rule-based (+3)	0.41	0.15	0.16	0.88	110	4.99
	Step	0.28	0.12	0.10	0.45	0	7.00
	Target	0.43	0.15	0.17	0.79	145	4.29
	TargetFast	0.41	0.06	0.27	0.53	0	4.57
	ScaleX	**0.48**	**0.02**	**0.28**	**0.51**	**0**	**3.77**
	QN-CTRL	0.48	0.03	0.01	0.54	0	4.47
SN2	*Static (1)*	1.10	0.39	0.32	1.52	820	1.00
	Rule-based	0.37	0.21	0.05	0.70	181	7.68
	Rule-based (+3)	0.43	0.27	0.05	1.19	272	6.96
	Step	0.32	0.18	0.05	0.57	0	9.24
	Target	0.56	0.34	0.06	1.37	436	5.20
	TargetFast	0.43	0.10	0.18	0.67	37	8.14
	ScaleX	**0.46**	**0.06**	**0.29**	**0.55**	**0**	**5.90**
	QN-CTRL	0.48	0.04	0.01	0.64	2	7.14
ST1	*Static (1)*	1.65	0.17	1.19	1.82	1000	1.00
	Rule-based	0.58	0.10	0.29	0.67	624	20.68
	Rule-based (+3)	0.51	0.06	0.29	0.67	41	25.24
	Step	0.51	0.07	0.29	0.66	34	25.60
	Target	0.49	0.06	0.29	0.73	28	28.13
	TargetFast	0.53	0.25	0.11	1.00	369	147.57
	ScaleX	**0.49**	**0.03**	**0.30**	**0.59**	**0**	**27.97**
	QN-CTRL	0.48	0.03	0.01	0.77	2	29.68
ST2	*Static (1)*	1.32	0.66	0.16	1.74	755	1.00
	Rule-based	0.65	0.43	0.01	1.72	553	14.28
	Rule-based (+3)	0.47	0.31	0.01	1.69	188	18.94
	Step	0.54	0.41	0.01	1.69	228	18.35
	Target	0.46	0.27	0.01	1.71	101	18.88
	TargetFast	0.45	0.28	0.01	1.00	353	89.81
	ScaleX	0.41	0.14	0.07	0.87	9	19.98
	QN-CTRL	**0.44**	**0.11**	**0.01**	**1.72**	**5**	**19.73**
RP1	*Static (1)*	1.57	0.32	0.01	1.80	969	1.00
	Rule-based	0.62	0.07	0.01	0.67	842	17.14
	Rule-based (+3)	0.52	0.07	0.01	0.64	17	22.29
	Step	0.54	0.06	0.01	0.66	98	22.02
	Target	0.49	0.06	0.01	0.65	18	24.93
	TargetFast	0.52	0.26	0.01	1.02	381	134.38
	ScaleX	**0.48**	**0.05**	**0.01**	**0.50**	**0**	**24.86**
	QN-CTRL	0.52	0.15	0.01	1.39	62	25.19
RP2	*Static (1)*	1.70	0.27	0.01	1.87	982	1.00
	Rule-based	0.93	0.10	0.01	1.00	982	17.14
	Rule-based (+3)	0.54	0.06	0.01	0.90	24	43.44
	Step	0.60	0.14	0.01	0.97	235	43.23
	Target	0.53	0.10	0.01	0.95	112	48.70
	TargetFast	0.55	0.29	0.01	1.04	434	329.58
	ScaleX	**0.49**	**0.04**	**0.01**	**0.52**	**0**	**48.48**
	QN-CTRL	0.48	0.03	0.01	0.65	3	50.99
TW	*Static (1)*	1.71	0.03	1.66	1.79	1000	1.00
	Rule-based	0.30	0.03	0.25	0.39	0	52.32
	Rule-based (+3)	0.33	0.04	0.26	0.43	0	47.14
	Step	0.46	0.08	0.27	0.57	0	31.74
	Target	0.46	0.04	0.26	0.54	0	30.71
	TargetFast	0.54	0.26	0.12	0.99	436	131.34
	ScaleX	**0.47**	**0.04**	**0.29**	**0.52**	**0**	**30.11**
	QN-CTRL	0.48	0.03	0.01	0.57	0	35.56

TABLE 4: Autoscaling solutions for a simulated system under different workloads to control the response time of AppDet.

AppExp. Table 5 shows the performance of the 8 autoscaling solutions when they control the AppExp application. In this case, *QN-CTRL* is the autoscaling approach with

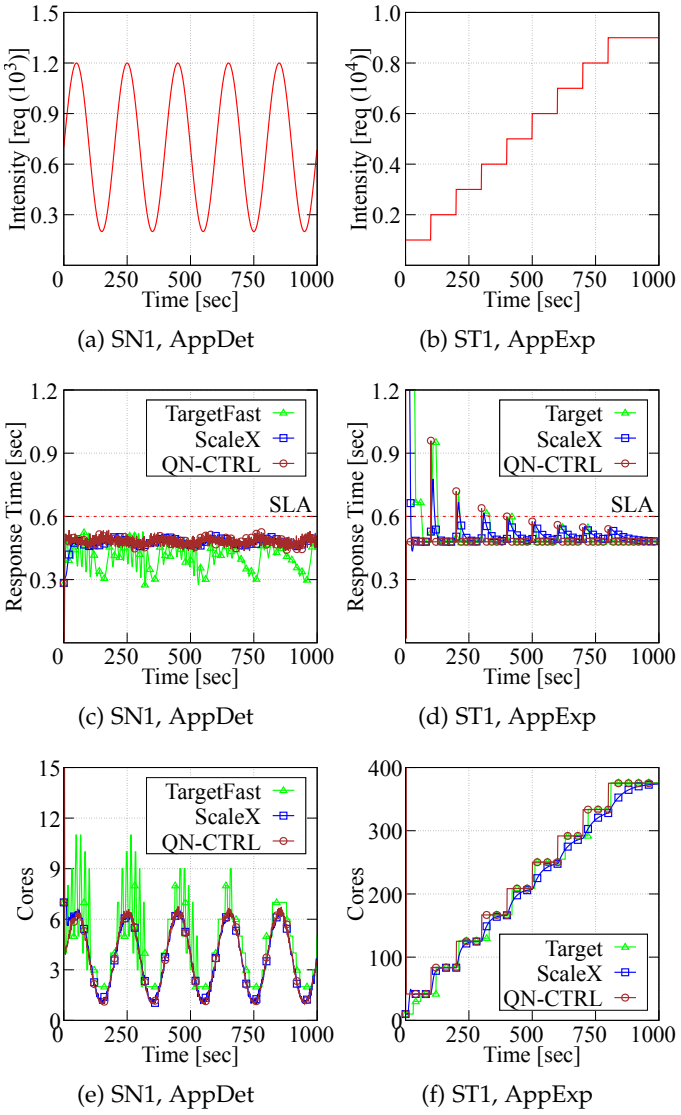


Fig. 4: Workload (1st row), response time (2nd row), and number of allocated cores (3rd row), for *ScaleX*, *QN-CTRL*, and the best industrial controller.

		RT					
W	**Approach**	μ	σ	m	M	V	A_μ
SN1	*Static (1)*	14.00	7.05	4.03	23.95	1000	1.00
	Rule-based	0.94	0.69	0.17	3.02	614	17.34
	Rule-based (+3)	0.69	0.58	0.13	3.02	476	24.74
	Step	0.58	0.58	0.11	3.02	327	31.48
	Target	0.59	0.44	0.19	3.02	352	28.40
	TargetFast	0.66	0.40	0.04	2.02	588	1236.92
	ScaleX	0.51	0.21	0.43	2.24	21	28.85
	QN-CTRL	**0.48**	**0.05**	**0.02**	**2.00**	**1**	**29.85**
SN2	*Static (1)*	22.02	13.92	2.32	41.66	1000	1.00
	Rule-based	1.20	0.88	0.08	3.79	667	20.02
	Rule-based (+3)	0.86	0.72	0.06	3.79	571	29.51
	Step	0.60	0.67	0.03	3.79	352	51.12
	Target	1.19	1.22	0.04	5.73	569	33.47
	TargetFast	0.67	0.40	0.03	2.06	590	2388.59
	ScaleX	0.54	0.27	0.31	2.60	316	43.67
	QN-CTRL	**0.48**	**0.06**	**0.02**	**2.00**	**1**	**47.06**
ST1	*Static (1)*	107.28	54.63	20.00	180.00	1000	1.00
	Rule-based	3.84	0.91	1.55	5.00	1000	26.14
	Rule-based (+3)	1.82	0.18	1.07	2.14	1000	58.41
	Step	0.73	0.38	0.49	2.00	333	194.34
	Target	0.56	0.28	0.48	2.00	124	218.01
	TargetFast	0.73	0.45	0.03	2.00	609	16778.47
	ScaleX	0.53	0.18	0.44	2.00	49	213.62
	QN-CTRL	**0.48**	**0.05**	**0.02**	**2.00**	**4**	**225.76**
ST2	*Static (1)*	75.25	42.51	1.00	100.00	1000	1.00
	Rule-based	7.74	8.39	0.04	50.00	797	15.75
	Rule-based (+3)	2.98	3.65	0.02	25.00	791	45.25
	Step	4.43	8.60	0.02	50.00	478	103.53
	Target	1.26	3.51	0.02	33.33	146	142.32
	TargetFast	0.70	0.61	0.02	5.44	496	13342.03
	ScaleX	0.55	0.49	0.02	5.94	63	152.88
	QN-CTRL	**0.48**	**1.49**	**0.02**	**47.53**	**2**	**214.03**
RP1	*Static (1)*	95.22	52.52	0.02	160.02	994	1.00
	Rule-based	5.49	0.75	0.02	5.97	994	17.14
	Rule-based (+3)	2.02	0.36	0.02	5.12	994	49.41
	Step	1.63	1.58	0.02	5.78	482	154.09
	Target	0.70	0.68	0.02	5.12	122	192.85
	TargetFast	0.71	0.44	0.02	1.20	589	14911.06
	ScaleX	0.51	0.06	0.02	1.17	18	191.52
	QN-CTRL	**0.48**	**0.02**	**0.02**	**0.56**	**0**	**199.84**
RP2	*Static (1)*	190.42	105.03	0.02	320.02	997	1.00
	Rule-based	10.98	1.49	0.02	11.93	997	17.14
	Rule-based (+3)	4.04	0.72	0.02	10.22	997	49.41
	Step	3.03	3.33	0.02	11.55	600	275.30
	Target	0.96	1.49	0.02	10.22	153	383.46
	TargetFast	0.73	0.45	0.02	1.21	602	31598.89
	ScaleX	0.55	0.15	0.02	2.33	22	365.00
	QN-CTRL	**0.48**	**0.02**	**0.02**	**0.58**	**0**	**399.46**
TW	*Static (1)*	109.47	15.32	88.26	154.30	1000	1.00
	Rule-based	1.43	0.30	0.95	2.06	1000	78.14
	Rule-based (+3)	1.09	0.36	0.56	2.06	907	110.41
	Step	0.62	0.30	0.46	2.06	128	196.38
	Target	0.54	0.26	0.44	2.06	49	220.00
	TargetFast	0.74	0.46	0.03	2.05	609	17311.00
	ScaleX	0.52	0.21	0.45	2.06	51	221.42
	QN-CTRL	**0.48**	**0.07**	**0.02**	**2.10**	**2**	**233.66**

TABLE 5: Autoscaling solutions for a simulated system under different workloads to control the response time of AppExp.

the smallest number of violations. This is due to *QN-CTRL* assuming that the service time is exponentially distributed, see Section 4. Independently of the considered workload, there is no other approach that performs as well as *QN-CTRL* when considering SLA violations. This comes with a larger number of cores allocated by *QN-CTRL*, i.e., it is always larger than the one of other strategies, except *Step* (Sine workloads) and *TargetFast* (all workloads). *TargetFast* presents a large number of violations, independently of the considered workload. Interestingly, it is also the approach that allocates the largest number of cores. In this case, combining a fast control period and proportional-only resource provisioning leads to a highly fluctuating system that struggles to reach stability.

Figures 4(d) and 4(f) show the response times and allocated resources obtained by *ScaleX*, *QN-CTRL*, and the best performing industrial solution when controlling ST1 depicted in Figure 4(b).

All autoscaling solutions show some violations, especially when the number of requests in the system suddenly changes. *QN-CTRL* reacts faster than *ScaleX* and *Target* to workload variations by allocating additional cores, especially when the number of requests is large. This allows *QN-CTRL* to significantly reduce the number of SLA violations, see Figure 4(d). *QN-CTRL* violates the SLA during initial control periods, i.e., while learning the workload to control, but it allocates an optimized amount of resources faster than other approaches for the rest of the experiment.

Target vs. SLA. The sensitivity of *ScaleX* and *QN-CTRL* to α, i.e., the parameter used to define the target/set-point of the controller in Eq. (11), is shown in Tables 6 and 7 for two workloads, ST2 and TW, respectively, when scaling strategies control the response time of AppDet. When the two approaches control ST2, *QN-CTRL* always allocates a (slightly) smaller number of cores than *ScaleX*, independently of α. However, *ScaleX* allows violating the SLA less than *QN-CTRL* when $\alpha > 0.90$. Looking at the average

		RT					
α	Approach	μ	σ	m	M	V	A_μ
0.70	*ScaleX*	0.36	0.12	0.06	0.84	8	23.75
	QN-CTRL	0.39	0.09	0.01	1.76	4	23.70
0.75	*ScaleX*	0.38	0.13	0.07	0.86	9	21.77
	QN-CTRL	0.42	0.10	0.01	1.87	4	21.56
0.80	*ScaleX*	0.41	0.14	0.07	0.87	9	19.98
	QN-CTRL	0.44	0.11	0.01	1.72	5	19.73
0.85	*ScaleX*	0.43	0.16	0.07	0.88	10	18.44
	QN-CTRL	0.46	0.13	0.01	1.85	5	18.08
0.90	*ScaleX*	0.45	0.17	0.07	0.89	19	17.10
	QN-CTRL	0.49	0.15	0.01	1.78	10	16.70
0.95	*ScaleX*	0.47	0.18	0.08	0.91	24	15.83
	QN-CTRL	0.51	0.16	0.01	1.86	104	15.44
0.99	*ScaleX*	0.49	0.19	0.08	0.89	104	14.92
	QN-CTRL	0.53	0.18	0.01	1.88	389	14.55

TABLE 6: Sensitivity of *ScaleX* and *QN-CTRL* to different threshold values. The simulated system is subject to the *ST2* workload and the AppDet application.

		RT					
α	Approach	μ	σ	m	M	V	A_μ
0.70	*ScaleX*	0.41	0.02	0.28	0.46	0	35.52
	QN-CTRL	0.42	0.03	0.01	0.50	0	41.18
0.75	*ScaleX*	0.44	0.03	0.28	0.50	0	32.64
	QN-CTRL	0.45	0.03	0.01	0.51	0	38.18
0.80	*ScaleX*	0.47	0.04	0.29	0.52	0	30.11
	QN-CTRL	0.48	0.03	0.01	0.57	0	35.56
0.85	*ScaleX*	0.50	0.04	0.28	0.55	0	27.88
	QN-CTRL	0.51	0.03	0.01	0.59	0	33.31
0.90	*ScaleX*	0.53	0.05	0.28	0.58	0	25.86
	QN-CTRL	0.54	0.03	0.01	0.62	7	31.13
0.95	*ScaleX*	0.55	0.05	0.29	0.62	13	24.10
	QN-CTRL	0.57	0.03	0.01	0.67	117	29.29
0.99	*ScaleX*	0.58	0.06	0.28	0.64	194	22.85
	QN-CTRL	0.59	0.04	0.01	0.68	431	27.99

TABLE 7: Sensitivity of *ScaleX* and *QN-CTRL* to different threshold values. The simulated system is subject to the *TW* workload and the AppDet application.

response time observed for the two autoscaling solutions, *QN-CTRL* is always closer to the given target than *ScaleX*.

Considering the real-world workload (i.e., TW), see Table 7, *ScaleX* works always better than *QN-CTRL* with a smaller number of SLA violations and allocated cores, independently of the value of α. Moreover, *ScaleX* meets the SLA for larger values of α (i.e., the SLA is not violated even with $\alpha = 0.90$), thus enabling larger savings for what concerns the allocation of cores. With this workload, both approaches keep the response time close to the desired target.

5.2 Cloud-based experiments

After evaluating *ScaleX* and *QN-CTRL* in a simulated environment, we analyze the performance of two applications deployed in AWS that serve different loads.

Applications. We consider two CPU-intensive systems (i.e., *dynamic-html* and *graph-mst*) from the SeBS benchmark [54], i.e., a widely used suite for the performance analysis of microservice applications. *Dynamic-html* is a web application that generates dynamic HTML from a predefined template. *Graph-mst* replicates analytic and engineering applications that solve irregular graphs using minimum spanning tree.

Workloads. Synthetic (SN3 and RP3) and real-world (TW and WK) traces are used to load the two SeBS applications, see Table 8. The Wikipedia workload (WK) is a trace of

Name	Type	Description
SN3	Sine	$req(t) = 20 * sin(t\frac{\pi}{100}) + 20$
RP3	Ramp	$req(t) = \begin{cases} 10 & t < 150 \\ 10 + 0.15 * t & 150 \leq t < 450 \\ 78 & \text{otherwise} \end{cases}$
TW	Twitter	Real trace collected on January 1st, 2021 [50]
WK	Wikipedia	Real trace collected on September 19th, 2007 [55]

TABLE 8: Workloads used to evaluate autoscaling solutions in cloud-based experiments.

traffic extracted by the Wikipedia archives which has been widely used in the literature [56], [57] as a benchmark. The intensity (i.e., requests per second) of real-world workloads is downsized by a factor of 40 to meet the limited number of resources that we can deploy on AWS. The shape and autocorrelation [58] of these workloads remain unchanged and identical to those observed in the available traces. TW and WK workloads are depicted in Figures 5(a) and 5(b), respectively. Requests are generated using Locust [59], a Python-based load-testing tool.

Environment. Applications and the workload generator are deployed in a *c3.8xlarge* AWS instance (32 vCPUs, 60GB memory, 8GB SSD, and Amazon Linux 2 OS). The SeBS applications are instantiated using Docker containers whose resource availability is 16 vCPUs. The OS uses 8 vCPUs for internal operations, while Locust generates new requests using the remaining 8 vCPUs. Since applications are deployed in a real environment, each experiment is repeated five times, for a total of 320 runs.

Service-level Agreement (SLA). For both applications, we set the SLA to 0.25 seconds and $\alpha = 0.7$. Therefore, *TargetRT* $= 0.175$, see Eq. (11).

5.2.1 Results

Dynamic-html. Table 9 reports the performance of all industrial controllers, *ScaleX*, and *QN-CTRL* when they are used with the dynamic-html application. The experiments confirm the trends observed in simulation-based tests: industrial approaches violate the SLA a significantly higher amount of times compared to *ScaleX* and *QN-CTRL*. On average, they violate 122 times the SLA against only 2.75 and 0 times of *ScaleX* and *QN-CTRL*, respectively. If we focus only on the top-performing industrial approach for each workload (namely, *TargetFast* for SN3 and WK, and *Step* for RP3 and TW), we still observe that the average number of SLA violations is 15, which is over 5 times more than the violations obtained by our solutions. The number of violations of *TargetFast* with SN3 and WK workloads is comparable to the one of *ScaleX*, but the industrial approach allocates a larger amount of resources. Furthermore, *ScaleX* and *QN-CTRL* provide, in most cases, faster response times compared to industrial approaches with standard deviations that are always lower. This means that our solutions can keep the system responsive and stable, while industrial solutions obtain more fluctuations.

In terms of resource allocation, *QN-CTRL* provision between 2 to 3 cores depending on the considered workload and never violates the SLA. *ScaleX* provides better performance (i.e., no violation and fewer allocated cores) compared to *QN-CTRL* only with workload RP3. Figures 5(c) and 5(e) depict the performance (response time and allo-

		RT					
W	**Approach**	μ	σ	m	M	V	A_μ
SN3	*Static (1)*	0.38	0.20	0.06	0.72	262	1.00
	Rule-based	0.20	0.12	0.06	0.55	86	1.62
	Rule-based (+3)	0.22	0.12	0.06	0.56	131	2.40
	Step	0.13	0.07	0.06	0.32	24	1.95
	Target	0.17	0.09	0.06	0.44	62	1.81
	TargetFast	0.15	0.05	0.06	0.25	5	2.49
	ScaleX	0.18	0.03	0.06	0.25	3	1.97
	QN-CTRL	**0.11**	**0.02**	**0.06**	**0.18**	**0**	**2.43**
RP3	*Static (1)*	0.74	0.51	0.06	1.47	351	1.00
	Rule-based	0.28	1.38	0.06	24.34	64	2.76
	Rule-based (+3)	0.49	1.14	0.06	13.11	211	3.55
	Step	0.16	0.04	0.06	0.31	32	2.34
	Target	0.19	0.09	0.06	0.41	98	2.47
	TargetFast	0.20	0.09	0.06	0.36	120	6.62
	ScaleX	**0.18**	**0.03**	**0.06**	**0.24**	**0**	**2.59**
	QN-CTRL	0.10	0.02	0.06	0.18	0	3.21
TW	*Static (1)*	0.27	0.14	0.06	0.66	206	1.00
	Rule-based	0.18	0.07	0.06	0.44	63	1.35
	Rule-based (+3)	0.18	0.09	0.06	0.52	80	1.96
	Step	0.11	0.05	0.06	0.30	15	1.90
	Target	0.16	0.06	0.06	0.37	37	1.47
	TargetFast	0.15	0.06	0.06	0.27	19	2.63
	ScaleX	0.18	0.03	0.06	0.26	1	1.68
	QN-CTRL	**0.11**	**0.02**	**0.06**	**0.18**	**0**	**2.04**
WK	*Static (1)*	0.34	0.05	0.06	0.44	570	1.00
	Rule-based	0.20	0.08	0.06	0.40	178	1.57
	Rule-based (+3)	0.30	0.75	0.06	9.60	167	2.86
	Step	0.10	0.04	0.06	0.29	11	1.96
	Target	0.18	0.08	0.06	0.39	138	1.66
	TargetFast	0.16	0.06	0.06	0.25	8	3.21
	ScaleX	0.19	0.04	0.07	0.26	7	2.15
	QN-CTRL	**0.11**	**0.02**	**0.06**	**0.15**	**0**	**2.43**

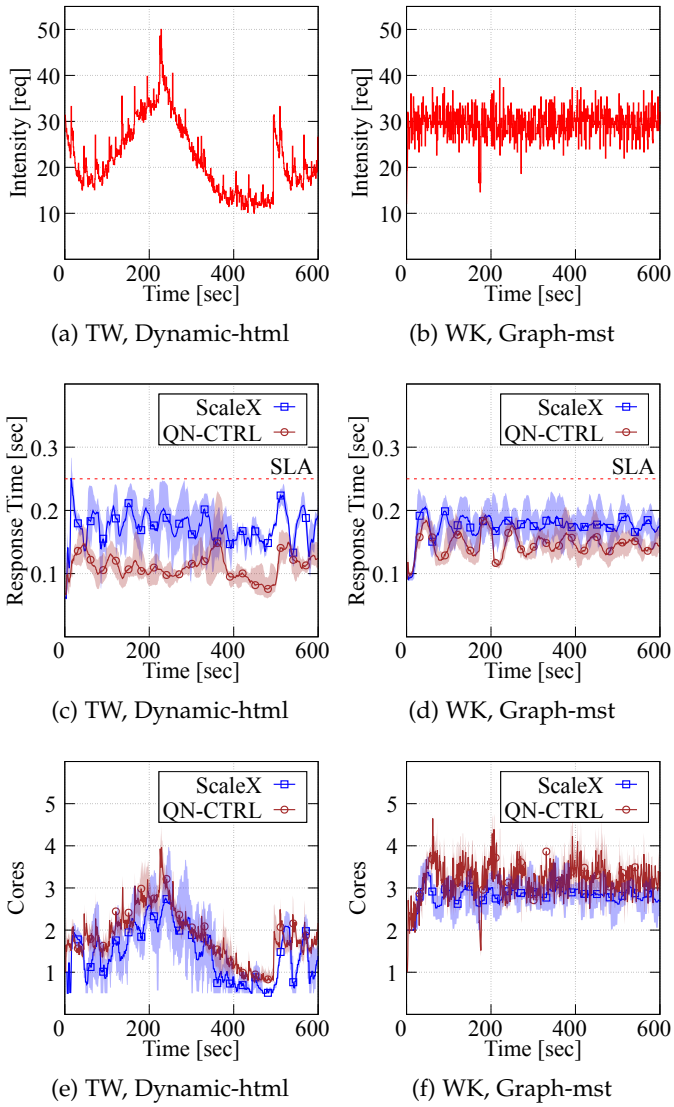
TABLE 9: Autoscaling solutions for a cloud system under different workloads to control the response time of *dynamic-html*.

cated cores, respectively) of *ScaleX* and *QN-CTRL* under workload TW. Solid lines depict average values, whereas shaded areas show the performance distribution. While *QN-CTRL* keeps the response time always far from the provided SLA, requests controlled by *ScaleX* occasionally violate the given objective. This means that *QN-CTRL* is more conservative: it allocates a (slightly) larger number of cores compared to *ScaleX* to safely keep the response time under the SLA.

Graph-mst. Similar results are observed with application graph-mst, as shown in Table 10. Industrial approaches obtain overall worse performance compared to our solutions. *QN-CTRL* never violates the SLA no matter the considered application but allocates more cores compared to *ScaleX*. *ScaleX* obtains zero SLA violations by using fewer resources compared to *QN-CTRL* with workloads RP3 and WK.

Overall, *Step* and *TargetFast* are the best industrial approaches, but they never outperform our solutions except for *TargetFast* that violates fewer times the SLA compared to *ScaleX* with workload SN3. As with application *dynamic-html*, the average response times and standard deviations obtained by *ScaleX* and *QN-CTRL* are lower compared to those of industrial solutions in most of the cases.

Figures 5(d) and 5(f) depict the response time and the number of cores used by *ScaleX* and *QN-CTRL* with workload WK. Our solutions are able to keep the response time always under the SLAs considering both the average and the maximum values. *QN-CTRL* tends to allocate more cores than *ScaleX* (3.20 cores against 2.85) with a resulting faster average response time (0.15s vs 0.18s).



(a) TW, Dynamic-html (b) WK, Graph-mst

(c) TW, Dynamic-html (d) WK, Graph-mst

(e) TW, Dynamic-html (f) WK, Graph-mst

Fig. 5: Workload (1st row), response time (2nd row), and number of allocated cores (3rd row), for *ScaleX* and *QN-CTRL*.

5.3 Cost Analysis

While *ScaleX* and *QN-CTRL* appear to clearly outperform industrial solutions, understanding the trade-off between the two approaches requires further analysis. Sections 5.1 and 5.2 show that *ScaleX* outperforms *QN-CTRL* with more deterministic applications and that, in most cases, it allocates fewer cores than *QN-CTRL*. In contrast, *QN-CTRL* performed better with more "noisy" applications and tended to produce fewer SLA violations.

Besides allocated resources and violations, we present a cost analysis to deepen the comparison. We adopt the model proposed by Kumar et al. [60], which formalizes the cost of running a system by means of two terms: (i) when the system behaves above agreed expectations (SLA), namely resource over-provisioning, and (ii) when the system is below the desired performance (SLA violations), i.e., resource under-provisioning. The cost is defined as follows:

		RT					
W	**Approach**	μ	σ	m	M	V	A_μ
SN3	*Static (1)*	0.85	0.52	0.09	1.66	358	1.00
	Rule-based	0.29	0.17	0.08	0.70	187	2.29
	Rule-based (+3)	0.34	0.27	0.08	1.34	210	2.92
	Step	0.19	0.16	0.08	0.71	64	2.84
	Target	0.24	0.13	0.08	0.66	135	2.58
	TargetFast	0.17	0.04	0.08	0.29	11	3.46
	ScaleX	0.18	0.04	0.09	0.29	22	2.81
	QN-CTRL	**0.14**	**0.02**	**0.09**	**0.20**	**0**	**3.11**
RP3	*Static (1)*	1.36	0.98	0.09	2.74	398	1.00
	Rule-based	0.20	0.07	0.09	0.53	80	3.16
	Rule-based (+3)	0.28	0.27	0.08	1.80	143	3.41
	Step	0.19	0.04	0.09	0.33	50	3.15
	Target	0.19	0.06	0.09	0.43	59	3.39
	TargetFast	0.20	0.07	0.08	0.32	126	7.93
	ScaleX	**0.18**	**0.02**	**0.09**	**0.24**	**0**	**3.78**
	QN-CTRL	0.13	0.02	0.09	0.29	0	4.33
TW	*Static (1)*	0.58	0.37	0.09	1.56	486	1.00
	Rule-based	0.20	0.08	0.09	0.50	111	2.08
	Rule-based (+3)	0.28	0.19	0.09	1.10	242	2.77
	Step	0.39	1.23	0.09	12.37	55	3.89
	Target	0.17	0.07	0.09	0.43	71	2.23
	TargetFast	0.17	0.05	0.09	0.29	24	3.77
	ScaleX	0.18	0.03	0.09	0.26	9	2.46
	QN-CTRL	**0.15**	**0.03**	**0.09**	**0.21**	**0**	**2.65**
WK	*Static (1)*	0.92	0.16	0.09	1.17	585	1.00
	Rule-based	0.20	0.10	0.09	0.65	98	2.59
	Rule-based (+3)	0.59	1.40	0.09	12.83	358	4.10
	Step	0.13	0.09	0.09	0.58	47	2.90
	Target	0.19	0.08	0.09	0.60	59	2.46
	TargetFast	0.17	0.05	0.09	0.30	54	4.30
	ScaleX	**0.18**	**0.02**	**0.09**	**0.23**	**0**	**2.85**
	QN-CTRL	0.15	0.02	0.09	0.20	0	3.20

TABLE 10: Autoscaling solutions for a cloud system under different workloads to control the response time of *graph-mst*.

$$COST = \sum_{t}^{T} \begin{cases} P * (\tau_t - \tau^\circ), & \text{if } \tau_t > \tau^\circ, \\ C * (\tau^\circ - \tau_t), & \text{otherwise.} \end{cases} \tag{12}$$

where τ_t is the monitored response time at time t, T is the total considered time, τ° is the set point on the response time (i.e., desired performance), and P and C are two constants, defined in $[0, \infty]$, that represent the cost of penalties and over-allocation, respectively. In essence, each contribution is proportional to the distance from the set point, and the ratio $R = \frac{P}{C}$ defines the weight of penalty costs compared to the ones caused by over-allocation.

We compute the cost of each experiment in Section 5.2 using Equation 12 with $C = 0.0015$ as in [60], and $R = P/C = [1, 2, 3, 4]$, thus understanding the impact of this ratio on the cost-effectiveness of each approach. This way we obtained a total of 320 data points.

The comparison metric, named $COMP$, is then given by the following formula:

$$COMP = \frac{(COST_{\textit{ScaleX}} - COST_{\textit{QN-CTRL}})}{min(COST_{\textit{ScaleX}}, COST_{\textit{QN-CTRL}})} \tag{13}$$

$COMP$ quantifies which approach performs better and by how much. If the result is negative, *ScaleX* is more cost-effective; otherwise, *QN-CTRL* outperforms the cost-effectiveness. The numerical values indicate the percentage of the comparison, e.g., a $COMP$ value of 0.30 indicates that *QN-CTRL* is 30% more cost-effective (i.e., cheaper) than *ScaleX*. Similarly, a value of -0.50 means that *ScaleX* is 50% more cost-effective than *QN-CTRL*.

		$COMP$			
App	**W**	$R = 1$	$R = 2$	$R = 3$	$R = 4$
dynamic	*SN3*	-2.17	-1.09	-0.56	-0.25
	RP3	-3.02	-1.67	-1.00	-0.60
	TW	-1.66	-0.68	-0.23	0.03
	WK	-1.20	-0.31	0.07	0.37
graph	*SN3*	-0.05	0.53	1.09	1.64
	RP3	-1.64	-0.56	-0.12	0.13
	TW	-0.28	0.13	0.43	0.68
	WK	-0.67	-0.12	0.16	0.42

TABLE 11: *ScaleX* and *QN-CTRL*: cost comparison.

Table 11 presents the values (averaged over 5 repetitions run for each experiment and approach) of $COMP$ in the 32 cases taken into account: 2 applications $\times$ 4 workloads $\times$ 4 values for R. For application *dynamic-html*, all values, no matter the workload, are negative with $R = 1, 2$, which suggests a higher cost-effectiveness for *ScaleX*. However, with $R > 2$, the value becomes positive for some workloads (TW and WK), which indicates a shift in cost-effectiveness in favor of *QN-CTRL*. Similarly, with application *graph-mst*, with low $R's$, the values are generally negative or close to zero, which suggests that *ScaleX* exhibits similar or superior cost-effectiveness. However, as R increases (i.e., $R > 1$), the values become consistently positive, that is, they witness that *QN-CTRL* is more cost-effective.

Summarizing, as expected, R plays a key role in defining the cost-effectiveness of a solution. With low values of $R's$, that is, when SLA violations and over-allocations have the same impact, *ScaleX* tends to be more cost-effective. This means that users may prefer very efficient core allocations at the cost of occasional violations. Higher values of $R's$ mean that SLA violations are considered more costly (important) than allocated resources, and this leads *QN-CTRL* outperforming *ScaleX*. This is the case when a more conservative resource allocation aims to minimize SLA violations.

5.4 Lessons Learned

The experiments based on both simulated and cloud executions allowed us to extensively compare six industrial autoscaling solutions, *ScaleX* and *QN-CTRL*.

One may think that the industrial solutions available in public cloud platforms are optimized to reduce SLA violations while providing a conservative resource allocation (higher billing but better performance). However, our results show that none of the industrial approaches we tested can consistently minimize response time violations no matter the type and shape of the workload, the target application, and the execution environment (simulation or cloud-based deployments). Some heuristics appear to be better than others in some cases, but their simple design makes them not flexible enough to perform well in different scenarios. In contrast, both *ScaleX* and *QN-CTRL* obtain significantly fewer SLA violations. Thanks to their theoretical

foundation, they can properly adapt to different workloads and, on average, their response times are closer to the set points with small standard deviations, indicating that the controlled system is kept responsive and stable.

If we consider simulation-based approaches, when requests are served with a deterministic service time, *ScaleX* generally reduces the number of SLA violations and allocated cores. *ScaleX* copes well with deterministic requests since it is feedback-based. Since the *QN-CTRL* assumption (i.e., exponential service time) deviates from the attributes of the controlled system (i.e., deterministic service time), *QN-CTRL* does not perform as well as *ScaleX*. The only exception is observed with the Stair workload (i.e., ST2), where the best autoscaling approach is *QN-CTRL*. This may be due to sudden changes in the workload intensity that mostly keep large values and facilitate the learning process on which *QN-CTRL* relies.

When the service time follows an exponential distribution (i.e., a common practice when considering cloud computing applications [61], [62], [63]), *QN-CTRL* is the autoscaling approach that provides the best performance (i.e., the smallest number of SLA violations and allocated cores). This is due to *QN-CTRL* assuming exponentially distributed service time for the controlled application.

Cloud-based experiments assess the performance of *ScaleX* and *QN-CTRL* with respect to industrial controllers and show possible directions to improve both approaches. The response time of real cloud applications is rarely deterministic due to application interference [64] and other noise. Hence, *ScaleX* may underestimate the application requirements and violate the given SLA since it leverages a feedback-based control mechanism. *QN-CTRL* may allocate a large number of cores when the service time does not follow an exponential distribution. This is due to the QN estimator that should be extended, as part of our future work, to subsume generic distributed service times.

All approaches need to comply with user-defined SLAs, a common practice in robust control is to define a target by scaling down the given SLA to facilitate controllers in their task [52], see Eq. (11). For this purpose, a scaling parameter is adopted, i.e., α in Eq. (11). Correctly tuning this parameter based on the response time model and workload enables enhanced performance and further savings.

Our evaluation shows that both *ScaleX* and *QN-CTRL* outperform all the examined industrial solutions. Moreover, our analysis indicates the choice between our two approaches depends on the criticality of the controlled application and the user's requirements. From an architectural standpoint, *ScaleX* is more lightweight and scalable than *QN-CTRL* due to its hierarchical approach. Conversely, *QN-CTRL* consistently demonstrates fewer SLA violations and a generally higher core allocation. The cost analysis detailed in Section 5.3 aims to focus on this aspect, evaluating the two autoscalers across increasing levels of criticality. The findings emphasize that if the objective is to minimize core allocation, even if it entails occasional SLA violations, then *ScaleX* is likely the more fitting choice. On the contrary, *QN-CTRL* is the preferable solution if the primary aim is to reduce SLA violations through a more conservative allocation strategy. As a result, the decision hinges on the application's criticality and, by extension, the acceptable balance between allocated resources and violations.

5.5 Threats to Validity

We conducted the experiments using two different applications exposed to a variegate set of seven representative workloads. Even if *ScaleX* and *QN-CTRL* outperform industrial solutions with respect to some metrics, in the following we argue on threats that may constrain the validity of obtained results [65].

Internal threats. The presented autoscaling techniques aim to control the average value of the performance metrics of interest (e.g., the system response time), not the distribution of values (e.g., tail latency) that instead may trigger further fluctuations to be managed. Both *ScaleX* and *QN-CTRL* require to set a percentage (i.e., α) that provides some flexibility from the user-defined SLA. This is intended to better manage disturbances in the control objectives and prevent violations. Our experiments demonstrate that setting $\alpha = 0.8$ works reasonably well for the considered techniques, and this setting is motivated through experimenting with other values for sensitivity analysis purposes. Besides, *QN-CTRL* technique requires an estimation window to learn (or update) the application model, hence the control effectiveness is affected by such learning procedure. In our experiments, we observe that a small amount of time (i.e., tens of seconds) is required to learn the initial model (i.e., the slowest one to be learned).

External threats. To mitigate generalization issues, our experiments use two different applications, multiple workloads, and compare a set of relevant approaches. The two applications are characterized by different response time models (already used in the literature [30], [43]) and service times (i.e., deterministic and exponential). We employ "standard" synthetic workload shapes: sine, stair (i.e., step-like), and ramp with different magnitudes obtaining comparable results to the ones measured with a real-world workload. To further improve our tests, we plan in the future to include more real-world workloads and types of applications.

6 Related Work

The problem of autoscaling resources at runtime in cloud-based applications has been widely studied [66], [67], [68], [69]. State-of-the-art approaches provide evidence of three main branches of research that we review in the following.

Control theory autoscaling. Our previous work [70] belongs to this category since we make use of control theory techniques (i.e., Proportional-Integral-Derivative — PID) to scale container-based applications on the basis of envisioned execution times. Al-Dulaimy et al. [67] presents a multi-loop autoscaling approach (including three layers) performing the following tasks: regulate resources' shares, scale resources, and migrate VMs from one host to another. However, the three loops work independently, and the lack of coordination in their actions may represent a threat. This limitation has been recently tackled in [66], where a Monitoring and Measurement Unit (MMU) is introduced to orchestrate the three layers, at the cost of a major complexity and performance overhead. Yu et al. [71] focuses on response times and the goal is to produce close-to-optimal

allocations, however, it does not provide formal guarantees on the control and it does not support vertical scalability. Podolskiy et al. [72] illustrate a comparison among different cloud providers (AWS, GCP, and Azure), but we already showed in [70] to scale faster than the others while targeting containers. Zhang et al. [73] make use of workload usage and duration, along with the cool-down period for scaling the containers in the physical machines, however utilization of CPU is considered only to vary the container size. Srirama et al. [74] present a strategy for scaling out/down the computing resources based on the resource utilization of the physical machines and the resources requested by the non-warm containers for processing the microservices. Salah et al. [75] evaluate the performance of web services focusing on Amazon cloud platform, and the comparison is performed between: (i) AWS EC2 Container Service for container-based deployment, and (ii) AWS Elastic Cloud Compute (EC2) for VM-based deployment. Results show that the performance of container-based web services is more critical, thus motivating our choice of designing experiments on containers only. Padala et al. [76] propose the usage of classical control theory techniques to automatically allocate resources based on the dynamic workload changes; it represents seminal work in the direction of designing adaptive resources that are dynamically adjusted in their resource shares. With respect to state-of-the-art control theory methodologies, our *ScaleX* approach shows the advantage of targeting vertical scaling on containers, and it turns out to provide the best solutions for a large number of different workloads acting on an application regulated by deterministic service times.

Analytical modeling autoscaling. In our previous work [18] we make use of an efficient SMT solver [77] to derive feasible control signals encoded in a QN performance model, but no optimization was considered (as we do in this paper). Funari et al. [78] propose an analytical model for evaluating storage occupancy of clusters hosting containerized applications and various scheduling policies are introduced to prevent the growth of storage utilization as cluster size increases. Tadakamalla et al. [79] present a single-queue multiple-server system model (G/G/c) subject to workload surges and scales the number of servers or their capacity to mitigate the effects of such surges. Cai et al. [80] propose a resource scaling engine that enables provisioning for (non-)periodic workloads of long-running services. It is constituted of three main components: (i) identification of periodic patterns in history arrivals; (ii) continuous tracking of request arrivals of services and prediction of future resource demands; (iii) continuous monitoring of services latency and re-provisioning of resources if a violation is imminent. Khamse-Ashari et al. [81] make use of an optimization problem relying on a system model that includes (i) servers distributed over multiple geographically distributed clusters, and (ii) active users/tenants. The goal is to minimize the operational costs of servers, while guaranteeing fairness (in terms of a minimum share of service) across different users. Ali et al. [82] present an analytical autoscaling approach based on quantile regression. Different from the solutions described in this paper, such an approach focuses on (burstable) VMs. Biswas et al. [83] propose an autoscaling technique based on broker profit (that is modeled analytically as the difference between user and broker costs), specifically resources are acquired on demand from a public cloud to service requests; however the broker itself may become a system bottleneck in computing required resources. With respect to state-of-the-art analytical modeling techniques, our *QN-CTRL* approach shows the advantage of exploiting an optimization formulation of the problem to minimize the number of servers, differently from [81] that instead focuses on resource sharing among services. Our methodology turns out to provide the best solutions for all the variegate workloads acting on an application regulated by the MVA algorithm.

Dynamic learning autoscaling. In our previous work [84] we propose a machine-learning approach to derive QNs from data and we found a maximum discrepancy of 10% between learned models and ground truth. Sun et al. [85] foster a technique grouping nodes with similar performance to reduce the waiting time for faster workers and improve the efficiency of computing resources, using distributed deep learning. Abdullah et al. [56] define the provisioning of resources as a learning process based on the historical autoscaling performance traces; the predicted workload is used to infer the number of containers needed to satisfy the set response time. Forecasting workload has been recently pursued by Feng et al. [86], where online regression is adopted relying on a sliding window method that takes into account trend and time correlations, along with random fluctuations of workload. Osypanka et al. [87] target cloud-based resource usage optimization through the discovery and learning of historical data used to decide when scaling system resources, specifically by comparing a pre-defined allocation plan with current resources. Makridis et al. [88] propose dynamic CPU allocation to address resource provisioning in virtualized servers by means of controllers that adjust the resources by computing variances with respect to previous CPU utilization.

Fangming Liu et al. [89] present *MarVeLScaler*, a prototype system that provides a preliminary estimation, through multi-view neural networks, of the required cluster size, adjusted at runtime due to cluster's real-time status. The learning process is performed considering homogeneous VMs only, heterogeneity is left as part of future work. Iqbal et al. [90] target the autoscaling of multi-tier applications with the goal of minimizing the usage of resources to handle dynamically increasing workloads and satisfy the response time requirements. A supervised learning method is adopted to identify the appropriate resources based on the prediction of the application response time and the request arrival rate. Farokhi et al. [91] relies on a fuzzy controller that builds upon known knowledge-based control, i.e., nonlinear functions implicitly constructed through fuzzy rules and fuzzy inference by imitating human/learned control. Compared to these approaches, our *QN-CTRL* methodology relies on learning a QN model that is later used as a software-hardware abstraction to define an efficient optimization problem. This way, *QN-CTRL* shows the benefit of providing optimal control signals that are efficiently deduced from the analysis of the learned performance model.

7 Conclusions

Autoscaling solutions are used by cloud providers to help containerized applications cope with fluctuating workloads. Several industrial solutions have been proposed, ranging from ones based on simple rules to others that employ more sophisticated heuristics. This paper analyzes the most important autoscaling solutions available on the major cloud providers and compares them with two systems, that we developed, based on control and queue theory, respectively. Unlike industrial approaches that focus on horizontal scalability, our solutions exploit vertical scaling that allows for faster control periods. Using a comprehensive empirical evaluation based on both simulated and cloud executions, we demonstrate how our approaches outperform the industrial competitors in almost all scenarios, and under dynamic workloads. A cost analysis highlights that the choice between *ScaleX* and *QN-CTRL* depends on the users' priorities in the trade-off between maximizing core allocation efficiency and minimizing SLA violations.

In future work, we plan to combine our two autoscaling techniques in a single controller that can adapt to different execution environments by efficiently alternating different control strategies.

8 Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. This work has been partially funded by the national funding for MUR-PRIN projects EMELIOT ($2020W3A5FY$) and DREAM ($20228FT78M$), the MUR-PRO3 project on Software Quality, and the MUR-PNRR project VITALITY ($ECS00000041$).

References

[1] W. Zhang, S. Zeadally, W. Li, H. Zhang, J. Hou, and V. C. Leung, "Edge AI as a Service: Configurable Model Deployment and Delay-energy Optimization with Result Quality Constraints," *IEEE Transactions Cloud Computing*, 2022.

[2] N. Mahmoudi and H. Khazaei, "Performance Modeling of Metric-Based Serverless Computing Platforms," *IEEE Transactions Cloud Computing*, 2022.

[3] L. Ruan, Y. Bai, S. Li, J. Lv, T. Zhang, L. Xiao, H. Fang, C. Wang, and Y. Xue, "Cloud Workload Turning Points Prediction via Cloud Feature-Enhanced Deep Learning," *IEEE Transactions Cloud Computing*, 2022.

[4] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic Multi-metric Thresholds for Scaling Applications Using Reinforcement Learning," *IEEE Transactions Cloud Computing*, 2022.

[5] D. Cheng, Y. Wang, and D. Dai, "Dynamic Resource Provisioning for Iterative Workloads on Apache Spark," *IEEE Transactions Cloud Computing*, 2021.

[6] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

[7] N. Sfondrini, G. Motta, and L. You, "Service level agreement (SLA) in Public Cloud environments: A Survey on the current enterprises adoption," in *International Conference on Information Science and Technology*. IEEE, 2015, pp. 181–185.

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

[9] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[10] V. Millnert and J. Eker, "HoloScale: horizontal and vertical scaling of cloud resources," in *International Conference on Utility and Cloud Computing*. IEEE, 2020, pp. 196–205.

[11] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, 2014.

[12] E. Lakew, A. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth, "KPI-Agnostic Control for Fine-Grained Vertical Elasticity," in *International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 589–598.

[13] L. Baresi and G. Quattrocchi, "Towards vertically scalable spark applications," in *Europ. Conference on Parallel Processing*. Springer, 2018, pp. 106–118.

[14] Kubernetes, "Production-Grade Container Orchestration," https://kubernetes.io, 2022.

[15] C. Barna, M. Fokaefs, M. Litoiu, M. Shtern, and J. Wigglesworth, "Cloud adaptation with control theory in industrial clouds," in *Proceedings of International Conference on Cloud Engineering Workshop (IC2EW)*, 2016, pp. 231–238.

[16] L. Baresi and G. Quattrocchi, "COCOS: A Scalable Architecture for Containerized Heterogeneous Systems," in *International Conference on Software Architecture*. IEEE, 2020, pp. 103–113.

[17] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems*. John Wiley & Sons, 2004.

[18] E. Incerto, M. Tribastone, and C. Trubiani, "Symbolic performance adaptation," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2016, pp. 140–150.

[19] E. D. Lazowska, J. Zahorjan, G. Scott Graham, and K. C. Sevcik, *Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, 1984.

[20] Amazon Web Services, "Step and simple scaling policies for Amazon EC2 Auto Scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html, 2022.

[21] Microsoft Azure, "AutoScale," https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview#what-is-autoscale, 2022.

[22] Amazon Web Services, "Target tracking scaling policies for Amazon EC2 Auto Scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html, [Online; 2022-03-16].

[23] Google Cloud Platform, "Autoscale to maintain a metric at a target value," https://cloud.google.com/compute/docs/autoscaler/scaling-cloud-monitoring-metrics#configure_utilization_target, 2022.

[24] Kubernetes, "Horizontal Pod Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022.

[25] Google Kubernetes Eninge, "Vertical Pod Autoscaler," https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler, 2022.

[26] Amazon Web Services, "Scheduled Scaling for Amazon EC2 Auto Scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-scheduled-scaling.html, 2022.

[27] Microsoft Azure, "Azure AutoScale based on Schedule," https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/tutorial-autoscale-performance-schedule#create-recurrence-profile, 2022.

[28] Google Cloud Platform, "Schedules in GCP," https://cloud.google.com/compute/docs/autoscaler#schedules, 2023.

[29] Amazon Web Services, "Predictive Scaling for Amazon EC2 Auto Scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html, 2022.

[30] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A Discrete-Time Feedback Controller for Containerized Cloud Applications," in *International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 217–228.

[31] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "NEPTUNE: Network- and GPU-aware Management of Serverless Functions at the Edge," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM/IEEE, 2022, pp. 144–155.

[32] I. Kumara, G. Quattrocchi, D. Tamburri, and W.-J. V. D. Heuvel, "Quality assurance of heterogeneous applications: The sodalite approach," in *Europ. Conference on Service-Oriented and Cloud Computing*. Springer, 2020, pp. 173–178.

[33] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review,"

IEEE Transactions Software Engineering, vol. 44, no. 8, pp. 784–810, 2017.

[34] L. T. Biegler and V. M. Zavala, "Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization," *Computers & Chemical Engineering*, vol. 33, no. 3, pp. 575–582, 2009.

[35] M. Kowal, M. Tschaikowski, M. Tribastone, and I. Schaefer, "Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models," in *International Conference on Automated Software Engineering*. IEEE, 2015, pp. 407–417.

[36] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[37] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[38] W. J. Stewart, *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009.

[39] E. Incerto, M. Tribastone, and C. Trubiani, "Software performance self-adaptation through efficient model predictive control," in *Proceedings of the International Conference on Automated Software Engineering, (ASE)*, 2017, pp. 485–496.

[40] ——, "Combined vertical and horizontal autoscaling through model predictive control," in *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par)*, vol. 11014. Springer, 2018, pp. 147–159.

[41] S. Brüggemann and C. Possieri, "On the use of difference of log-sum-exp neural networks to solve data-driven model predictive control tracking problems," *IEEE Control Systems Letters*, vol. 5, no. 4, pp. 1267–1272, 2020.

[42] R. P. Borase, D. Maghade, S. Sondkar, and S. Pawar, "A review of pid control, tuning methods and applications," *International Journal of Dynamics and Control*, vol. 9, no. 2, pp. 818–827, 2021.

[43] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *Journal of the ACM*, vol. 27, no. 2, pp. 313–322, 1980.

[44] D. A. Menascé and S. Bardhan, "TDQN: Trace-driven analytic queuing network modeling of computer systems," *Journal of Systems and Software*, vol. 147, pp. 162–171, 2019.

[45] A. Kattepur, "Towards Structured Performance Analysis of Industry 4.0 Workflow Automation Resources," in *International Conference on Perf. Engineering*. ACM, 2019, pp. 189–196.

[46] S. Dipietro, G. Casale, and G. Serazzi, "A Queueing Network Model for Performance Prediction of Apache Cassandra," in *International Conference on Perf. Evaluation Methodologies and Tools*. ACM, 2016.

[47] G. Casale, J. F. Pérez, and W. Wang, "QD-AMVA: Evaluating systems with queue-dependent service requirements," *Perf. Evaluation*, vol. 91, pp. 80–98, 2015.

[48] C. A. Smith and A. B. Corripio, *Principles and practices of automatic process control*. John wiley & sons, 2005.

[49] M. Maggio, A. V. Papadopoulos, and A. Leva, "On the use of feedback control in the design of computing system components," *Asian Journal of Control*, vol. 15, no. 1, pp. 31–40, 2013.

[50] Archive Team, "Twitter Stream 2021-01," https://archive.org/details/archiveteam-twitter-stream-2021-01, [Online; 2022-09-01].

[51] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing Inference Serving on Serverless Platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[52] K. Zhou and J. C. Doyle, *Essentials of robust control*. Prentice Hall Upper Saddle River, NJ, 1998, vol. 104.

[53] E. Incerto, A. Napolitano, and M. Tribastone, "Moving Horizon Estimation of Service Demands in Queuing Networks," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2018, pp. 348–354.

[54] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[55] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[56] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Transactions on Services Computing*, 2020.

[57] K. Zheng, Y. Bai, and X. Wang, "Fctcon: Dynamic control of flow completion time in data center networks for power efficiency," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1467–1478, 2019.

[58] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[59] J. Heyman, C. Byström, J. Hamrén, and H. Heyman, "Locust: An open source load testing tool," https://web.archive.org/web/20221226091044/https://locust.io/.

[60] S. Kumar, T. Chen, R. Bahsoon, and R. Buyya, "Debtcom: Technical debt-aware service recomposition in saas cloud," *IEEE Transactions on Services Computing*, 2023.

[61] K. Li, "Quantitative Modeling and Analytical Calculation of Elasticity in Cloud Computing," *IEEE Transactions Cloud Computing*, vol. 8, no. 4, pp. 1135–1148, 2020.

[62] P. Ambati, N. Bashir, D. E. Irwin, and P. J. Shenoy, "Modeling and Analyzing Waiting Policies for Cloud-Enabled Schedulers," *IEEE Transactions Parallel Distributed Systems*, vol. 32, no. 12, pp. 3081–3100, 2021.

[63] X. Qin, B. Li, and L. Ying, "Distributed Threshold-based Offloading for Large-Scale Mobile Cloud Computing," in *Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[64] F. Antonelli, V. Cortellessa, M. Gribaudo, R. Pinciroli, K. S. Trivedi, and C. Trubiani, "Analytical modeling of performance indices under epistemic uncertainty applied to cloud computing systems," *Future Generation Computer Systems*, vol. 102, pp. 746–761, 2020.

[65] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[66] A. Al-Dulaimy, J. Taheri, A. V. Papadopoulos, and T. Nolte, "LOOPS: A Holistic Control Approach for Resource Management in Cloud Computing," in *International Conference on Performance Engineering*. ACM, 2021, pp. 117–124.

[67] A. Al-Dulaimy, J. Taheri, A. Kassler, M. R. H. Farahabady, S. Deng, and A. Zomaya, "Multiscaler: A multi-loop auto-scaling approach for cloud-based applications," *IEEE Transactions Cloud Computing*, 2020.

[68] W. Delnat, E. Truyen, A. Rafique, D. Van Landuyt, and W. Joosen, "K8-Scalar: A Workbench to Compare Autoscalers for Container-Orchestrated Database Clusters," in *International Conference on Software Engineering for Adaptive and Self-Managing Systems*, 2018, p. 33–39.

[69] I. Pietri and R. Sakellariou, "Mapping virtual machines onto physical machines in cloud computing: A survey," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–30, 2016.

[70] L. Baresi and G. Quattrocchi, "A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications," in *International Conference on Web Services*. IEEE, 2020, pp. 94–101.

[71] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic Scaling for Microservices with an Online Learning Approach," in *International Conference on Web Services*. IEEE, 2019, pp. 68–75.

[72] V. Podolskiy, A. Jindal, and M. Gerndt, "IaaS Reactive Autoscaling Performance Challenges," in *International Conference on Cloud Computing*. IEEE, 2018, pp. 954–957.

[73] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems*, vol. 98, pp. 672–681, 2019.

[74] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *Journal of Network and Computer Applications*, vol. 160, p. 102629, 2020.

[75] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "Performance comparison between container-based and VM-based services," in *International Conference on Innovations in Clouds, Internet and Networks*. IEEE, 2017, pp. 185–190.

[76] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple

virtualized resources," in *Europ. Conference on Computer Systems*. ACM, 2009, pp. 13–26.

[77] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[78] L. Funari, L. Petrucci, and A. Detti, "Storage-saving scheduling policies for clusters running containers," *IEEE Transactions Cloud Computing*, 2021.

[79] V. Tadakamalla and D. A. Menascé, "Autonomic Elasticity Control for Multi-Server Queues Under Generic Workload Surges in Cloud Environments," *IEEE Transactions Cloud Computing*, vol. 10, no. 2, pp. 984–995, 2022.

[80] B. Cai, K. Li, Z. Laiping, and R. Zhang, "Less Provisioning: A Hybrid Resource Scaling Engine for Long-running Services with Tail Latency Guarantees," *IEEE Transactions Cloud Computing*, 2020.

[81] J. Khamse-Ashari, I. Lambadaris, G. Kesidis, B. Urgaonkar, and Y. Zhao, "A Cost-Efficient and Fair Multi-Resource Allocation Mechanism for Self-Organizing Servers," in *Global Communications Conference*. IEEE, 2018, pp. 1–7.

[82] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "It's not a Sprint, it's a Marathon: Stretching Multi-resource Burstable Performance in Public Clouds," in *International Middleware Conference Industrial Track*. ACM, 2019, pp. 36–42.

[83] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki, "An Auto-Scaling Framework for Controlling Enterprise Resources on Clouds," in *International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 971–980.

[84] G. Garbi, E. Incerto, and M. Tribastone, "Learning Queuing Networks by Recurrent Neural Networks," in *International Conference on Performance Engineering*. ACM, 2020, pp. 56–66.

[85] H. Sun, Z. Gui, S. Guo, Q. Qi, J. Wang, and J. Liao, "GSSP: Eliminating Stragglers through Grouping Synchronous for Distributed Deep Learning in Heterogeneous Cluster," *IEEE Transactions Cloud Computing*, 2021.

[86] B. Feng, Z. Ding, and C. Jiang, "Fast: A forecasting model with adaptive sliding window and time locality integration for dynamic cloud workloads," *IEEE Transactions on Services Computing*, 2022.

[87] P. Osypanka and P. Nawrocki, "Qos-aware cloud resource prediction for computing services," *IEEE Transactions on Services Computing*, 2022.

[88] E. Makridis, K. Deliparaschos, E. Kalyvianaki, A. Zolotas, and T. Charalambous, "Robust dynamic cpu resource provisioning in virtualized servers," *IEEE Transactions on Services Computing*, 2020.

[89] Y. Li, F. Liu, Q. Chen, Y. Sheng, M. Zhao, and J. Wang, "MarVeLScaler: A Multi-View Learning-Based Auto-Scaling System for MapReduce," *IEEE Transactions Cloud Computing*, vol. 10, no. 1, pp. 506–520, 2022.

[90] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive Auto-Scaling of Multi-Tier Applications Using Performance Varying Cloud Resources," *IEEE Transactions Cloud Computing*, vol. 10, no. 1, pp. 595–607, 2022.

[91] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints," in *International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 69–80.

Giovanni Quattrocchi received his Ph.D. in Computer Engineering in 2018 from Politecnico di Milano, where he is currently a Junior Assistant Professor (RTD-a). He was a visiting researcher at University of California San Diego and Imperial College London. His research interests include self-adaptive systems, software architectures, edge computing, and blockchain-based systems.

Emilio Incerto is a Junior Assistant Professor (RTD-a) in Computer Science within the SySMA research unit of IMT Lucca since 2021. He received his Ph.D. in Computer Science in 2019 from the Gran Sasso Science Institute of L'Aquila, Italy. His research focuses on modeling and controlling quantitative properties (e.g., response time), of software systems subject to stringent extra-functional requirements.

Riccardo Pinciroli received M.S. (2014) and Ph.D. (2018) degrees in computer engineering from Politecnico di Milano. He is currently a Post-doc Fellow in Computer Science at the Gran Sasso Science Institute. His research interests include stochastic modeling, performance evaluation, energy efficiency, and uncertainty propagation applied to cloud computing, data-centers, and cyber-physical systems.

Catia Trubiani is an associate professor at the Gran Sasso Science Institute (GSSI), Italy. Previously she collaborated with various international research institutes like the Karlsruhe Institute of Technology in Germany, and the Imperial College of London in UK. Her research interests include quality-based modeling and analysis of interacting heterogeneous distributed systems, feedback strategies on software architectures, more recently applied to cyber-physical systems.

Luciano Baresi is a full professor at the Politecnico di Milano. Luciano was visiting professor at University of Oregon (USA) and visiting researcher at University of Paderborn (Germany). His research interests are in the broad area of software engineering and include formal approaches for modeling and specification languages, distributed systems, service-based applications and mobile, self-adaptive, and pervasive software systems.