Journal of Grid Computing (2023) 21:80
https://doi.org/10.1007/s10723-023-09718-2

RESEARCH

Cost-Availability Aware Scaling: Towards Optimal Scaling of Cloud Services

Andre Bento · **Filipe Araujo** · **Raul Barbosa**

Received: 24 May 2023 / Accepted: 14 November 2023 / Published online: 7 December 2023

Abstract Cloud services have become increasingly popular for developing large-scale applications due to the abundance of resources they offer. The scalability and accessibility of these resources have made it easier for organizations of all sizes to develop and implement sophisticated and demanding applications to meet demand instantly. As monetary fees are involved in the use of the cloud, one of the challenges for application developers and operators is to balance their budget constraints with crucial quality attributes, such as availability. Industry standards usually default to simplified solutions that cannot simultaneously consider competing objectives. Our research addresses this challenge by proposing a Cost-Availability Aware Scaling (CAAS) approach that uses multi-objective optimization of availability and cost. We evaluate CAAS using two open-source microservices applications, yielding improved results compared to the industry standard CPU-based Autoscaler (AS). CAAS can find optimal system configurations with higher availability, between 1 and 2 nines on average, and reduced costs, 6% on average, with the first application, and 1 nine of availability on average, and reduced costs up to 18% on average, with the second application. The gap in the results between our model and the default AS suggests that operators can significantly improve the operation of their applications.

Keywords Cloud services · Microservices · Availability modeling · Cost-effectiveness · Multi-objective optimization · Autoscaling

A. Bento (✉) · F. Araujo · R. Barbosa
Department of Informatics Engineering, Centre for Informatics and Systems of the University of Coimbra, 3030-290 Coimbra, Portugal
e-mail: apbento@dei.uc.pt

F. Araujo
e-mail: filipius@dei.uc.pt

R. Barbosa
e-mail: rbarbosa@dei.uc.pt

1 Introduction

Organizations are increasingly deploying their software on containerized environments, such as Docker [1] and others [2], because these can easily run on available, on-demand, resources, e.g., directly on a public cloud or via Kubernetes clusters [3]. Replicating application components becomes much simpler, meeting the demand instantly or cutting unnecessary resources, rather than undergoing a complex and expensive update to the infrastructure [4]. In addition, cloud-based environments provide on-demand resources to ease the execution of workloads with nearly uninterrupted access and high availability, usually a crucial request for applications [5].

To take advantage of this paradigm, organizations shifted the development of applications from the traditional monolith, a singular, large computing component with one code base that couples all the business logic, to an architectural style known as microservices [6,7],

where the system is decomposed into self-contained standalone components that interact over the network, through well-defined Application Programming Interfaces (APIs). The simplicity of each component entails a better division of work among teams, while also bringing some management advantages, as each component is easier to develop, configure, integrate, and deploy in Continuous Integration/Continuous Delivery cycles. Nonetheless, this architectural style results in more services, more connections between them, and more connections to external services [8]. The resulting distributed systems are complex and more difficult to manage and monitor.

These systems evolve organically and control is usually decentralized [9]. Since services can scale in and out, automatic mechanisms continuously monitor some specific metric to meet diverse service requirements. As an example, Amazon Web Services Auto Scaling [10] employ metrics that may encompass a range of factors such as the average Central Processing Unit (CPU) utilization, the average memory utilization, network in and network out, request count, and other custom metrics. Typically, the metric of primary concern is the CPU load of the replicas, to check if the load exceeds or falls below a given threshold. Should that happen, the autoscaler will either spawn a new replica or remove an existing one, to adjust the computational resources and ensure that the CPU load stays within predetermined boundaries [11]. Even though autoscalers may also resort to other metrics, such as memory, network Input/Output, or latency, for example, a decentralized approach will always have many independent sources of control that may drive the system to inefficient points of operation. The overall result may show limited consideration for the overall cost involved in running whatever number of replicas emerges from sequential, separate decisions, or even for system availability if it runs out of resources. While individual services can indeed prioritize their availability, overseeing the overall cost is beyond their ability to control.

The main problem with this decentralized approach is that it is: 1) delayed, as they just apply configurations when the system is already failing or not meeting the objective, thus not enabling continuous control; and 2) local, as they provide configurations looking only at one service at a time, thus offering cost-inefficient alternatives for a local section of the entire system and do not conduct a comprehensive analytical inspection of all components and the complete system.

Passing from an uncoordinated control effort to a centralized one that considers load and optimizes the system for availability and cost is, therefore, a goal worthwhile pursuing. Hence, choosing the optimal configuration for all services, to simultaneously optimize the two objectives of availability and cost remains an open challenge, as the system resulting from numerous components and respective interactions is usually beyond the cognitive capacity of operators, the application administrators responsible for managing and ensuring the proper functioning of the applications, who are unable to find some optimal configuration for a given load [12].

In this paper, we propose to do exactly this, by formalizing an optimization problem based on the model of a microservice application. To evaluate our method, Cost-Availability Aware Scaling (CAAS)[1], we monitor and control two microservice-based applications, 1) Robot Shop [13] and 2) Sock Shop [14]. Against the most widely used approach in the industry, autoscaling based on CPU utilization using the Kubernetes Horizontal Pod Autoscaler (HPA) [15], we can considerably cut the cost of running the application while keeping the availability within strict boundaries. Additionally, finding a solution for the optimization problem we propose is fast. These results are encouraging and show that a centralized approach, even with a simple model like the one we use, can result in significant gains for the operators.

In this work, we make the following contributions:

- We model the availability of microservices workflows. Specifically, we merge Amdahl's law [16] into the Generalized Logistic Function (GLF), enabling the modeling of non-linear variations as the means to represent availability for a given load and the number of replicas.
- Based on this model, we formalize an optimization problem that incorporates the objectives of availability and cost. The experimental evaluation shows that CAAS achieves better results than the industry default autoscaling based on CPU utilization in two case studies, Robot Shop and Sock Shop.

[1] https://github.com/SysOBs/caas

The remainder of the paper is organized as follows. Section 2 provides an overview of existing work related to our research topic. Section 3 depicts the problem statement for this research. Section 4 outlines the methodology we have developed for optimizing availability and cost in cloud services through a multi-objective approach. Section 5 presents our case study using two microservice-based applications as examples. Section 6 provides a comparison of results between CAAS and the CPU-based Kubernetes HPA, along with a discussion and observations. Section 7 presents the limitations and potential threats to the validity of our study. Section 8 summarizes the key findings of our research and discusses potential future work in this area.

2 Related Work

Cloud services have become a popular choice for developing large-scale applications due to the on-demand resources and pay-per-use business models. However, as the use of cloud services incurs monetary fees, one of the key challenges for application developers and operators is to balance their budget constraints with crucial non-functional requirements such as availability. In the field of autoscaling approaches for cloud services, there are three major groups: 1) Statistical-based autoscaling, 2) Multi-objective optimization autoscaling and 3) Machine learning-based autoscaling [17].

Statistical-based autoscaling leverages statistical methods to determine the optimal moments for resource allocation adjustments, whether through rule-based or threshold-based strategies, to scale up or down the resources effectively. Several works try to do this, such as Bauer et al. present Chamulteon, an approach to scale applications consisting of multiple services in a coordinated manner [18], Srirama et al. present a strategy to deploy the requested applications on the best-fit lightweight containers, with minimum deployment time, based on the resource requirements [19]. Singh et al. propose Robust Hybrid Auto-Scaler, an auto-scaling technique designed with threshold-based rules [20].

Multi-objective optimization-based autoscaling involves using optimization techniques to find the best configuration for the system while considering multiple objectives such as performance and cost. The number of potentially related works is not high at the moment of writing; however, they present promising results and are motivated by the good results obtained in other resource management optimization problems. Guerrero et al. propose a genetic algorithm approach for multi-objective optimization of container allocation in cloud architecture [21], and Ali et al. propose a multi-objective task-scheduling optimization problem that minimizes both the makespans and total costs in a fog-cloud environment [22].

Machine learning-based autoscaling, on the other hand, uses machine learning algorithms to learn from historical performance data to make predictions about future resource needs [23]. Prachitmutita et al. proposed an autoscaling framework using artificial recurrent neural networks for workload prediction [24]. Yu et al. presented Microscaler, which combines online learning and heuristics to identify scaling-needed services and meet Service Level Agreements [25]. Other works by Hanqing et al. [26] and Rzadca et al. [27] use neural networks for resource requirement prediction and employ reinforcement learning to optimize resource allocation. Coulson et al. designed a prototype auto-scaling system that learns from past service experiences and uses a hybrid model to identify microservices to scale [28]. Marie-Magdelaine et al. proposed a proactive autoscaling framework using a Long Short-Term Memory-based learning model to dynamically adjust the resource pool, improving latency for cloud-native applications [29]. Khaleq et al. introduced an intelligent autonomous microservice autoscaling system for Google Kubernetes Engine, achieving up to a 20% enhancement in microservice response time compared to default autoscaling [30]. Horn et al. introduced a hybrid auto-scaling mechanism that utilizes machine learning to mitigate resource waste and maintain desired response times for microservices in Kubernetes [31]. However, machine learning-based autoscaling, while promising, poses disadvantages such as 1) Limited explainability, 2) Increased demand for computational resources compared to other approaches, and 3) Continuous model retraining, making it a more complex and resource-intensive solution, particularly in early stages of development. Hence, these solutions are typically not deployed in production environments.

These works can be categorized into three approaches: 1) Delayed autoscaling responds to events like high CPU load, 2) Immediate autoscaling takes action before reaching thresholds, and 3) Hybrid autoscaling com-

bines delayed and immediate strategies. Common industry standards often default to simplified solutions. For instance, they might monitor the average CPU utilization of a group of services and scale the number of replicas up or down based on this metric. However, it is essential to note that solutions such as Amazon Web Services Auto Scaling, provide a wide range of metrics beyond just average CPU utilization. These include metrics like average memory utilization, network in and network out, request count, and even the flexibility to define and incorporate custom metrics. Despite this rich array of metrics, the average CPU utilization of resources typically remains the primary target for scaling out replicas [32,33].

In this complex landscape, balancing competing objectives while minimizing resource waste, remains an open challenge and offers a potential avenue to address these challenges while optimizing the overall system behavior. In this context, our research aims to address this challenge by proposing a multi-objective optimization approach to model system availability and cost globally, ensuring immediate and continuous optimization for all services, and enabling it to take action before reaching the thresholds.

3 Problem Statement

We consider an application comprised of multiple services. One may replicate each one of these services separately for improved availability, but each replica entails a monetary cost. The application may consist of various workflows that interact with different services. Our primary focus is on one of these workflows, which we aim to ensure is always available. We will subsequently explore strategies for supporting multiple workflows. We define one workflow of the application to be available if it responds to the client within a predetermined time limit, t_{lim}. In addition to the response time, the response must also be correct. Since we focus on Hypertext Transfer Protocol (HTTP), we simplify and assume that the request is successful if its response time, t_r, is faster than a threshold, t_{lim}, i.e., $t_r < t_{lim}$ *and* the response error status is not in the 5xx class; otherwise, the response is unsuccessful.

Since an unavailable microservice application is useless [34], ensuring that services are available is very important. We measure availability (A) as the ratio of successful requests to total requests, as per equation (1):

$$A = \frac{Successful\ requests}{Total\ requests} \tag{1}$$

We aim to achieve maximum availability while simultaneously minimizing costs. While defining a multi-objective optimization problem is a possibility, an approach we followed in [35], one needs an additional policy to select one of the solutions in the Pareto front. To simplify this process, we take a more direct approach by combining the cost and availability objectives into a single objective function. This simplifies the optimization problem and yields a single optimal solution, provided that the Pareto front is convex. A convex function has a unique global minimum, which ensures that the optimal solution balances both objectives. By combining the objectives into a convex function, we can avoid the need for additional policies and streamline the decision-making process. We combine availability and cost into one metric, by assigning equal weights to *unavailability* and cost, where unavailability, U, is $1 - A$ (refer to (2)), being A the *availability* of the application. Unavailability can be caused by a lack of resources or a faulty service, leading to an inability to meet the Service-Level Objectives (SLOs) [36].

$$U = 1 - A \tag{2}$$

The cost to run an application on the cloud can be complex and influenced by various factors. Nonetheless, two of the dominating factors are the amount of CPU and memory resources used [37]. Hence, our cost model considers CPU and memory used across all services, according to (3):

$$C_{total} = \sum_{i=1}^{n} (c_c \times cpu_i + c_m \times mem_i) \times x_i \quad \text{where} \quad x_i \in \mathbb{N}_0 \tag{3}$$

The overall cost, C_{total}, is the sum of the costs for all replicas of all services. It comprises the overall number of Virtual Central Processing Units (vCPUs), cpu_i, in units, for service i, the overall memory usage mem_i, in GibiBytes, for service i, two coefficients, c_c and c_m,

which are, respectively, the cost per vCPU and the cost per GibiByte of memory, and the number of replicas x_i for each service i. We assume that all replicas in the same service have similar configurations concerning the number of vCPUs and memory.

Since we aim at combining unavailability and cost in the same objective function, with similar weights, to perform computations on the optimization step, we normalize the value of cost to be between 0 and 1, according to the expression $C = C_{total}/C_{max}$, where C_{max} represents the maximum cost of the infrastructure. The value of C_{max} is computed based on the maximum CPU and memory allowed by the infrastructure sized by the application administrators responsible for managing and ensuring the proper functioning of the applications, i.e., the operators.

Simple approaches to determining the number of replicas per service, x_i, may be based on metrics that are local to each service. These may fall onto local optima, e.g., due to the use of some greedy approach that only looks at occupied vCPU percentage. To overcome this limitation, a global policy may compute the overall best number of replicas for all services of the entire application. Such a policy may consider the available resources and the impact that each service has on the global latency.

Given an application with n services, our goal is to determine the number of replicas, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, for each of these services that minimizes application unavailability and cost. Since hiring vCPU and memory entails a cost, the budget will limit the overall replication of the application services.

We assume that the application operators will also impose constraints on the SLOs and, therefore, we add another constraint for the unavailability to be at most $1-A_{SLO}$. The challenge is, therefore, to find an efficient configuration, represented by a vector of replicas $\mathbf{x}$, that balances unavailability and cost. We formalize the problem in equation (4):

$$\begin{aligned} \text{Minimize} \quad & U(\mathbf{x}) + C(\mathbf{x}) \\ \text{s.t.} \quad & 0 \leq U(\mathbf{x}) \leq 1 - A_{SLO} \\ & 0 \leq C(\mathbf{x}) \leq 1 \\ & \mathbf{x} = [x_1\ x_2\ \ldots\ x_n],\ x_i \in \mathbb{N}_0 \\ \text{where} \quad & C(\mathbf{x}) = \frac{C_{total}(\mathbf{x})}{C_{max}} \\ & U(\mathbf{x}) = 1 - A(\mathbf{x}) \end{aligned} \tag{4}$$

A linear combination of unavailability and cost offers a versatile and intuitive approach to effectively balance multiple conflicting objectives. This method works by assigning weights to each objective and combining them into a single objective function allowing us to trade off one objective against the other, making it possible to find a compromise solution that meets both objectives to some degree. A common practice for prioritizing different objective contributions is to introduce constant weight factors. Hence, we can multiply U by ω_1 and C by ω_2 resulting in the following expression: Minimize $\omega_1 \times U + \omega_2 \times C$. Using this approach, it is possible to consider cost alone setting $\omega_1 = 0$ and $\omega_2 = 1$, or, minimizing unavailability alone with $\omega_1 = 1$ and $\omega_2 = 0$. In this paper, we use equal weights for both objective functions by defining $\omega_1 = \omega_2 = 1$. In future work, exploring different weight configurations and their effects on the optimization process would be an interesting direction for research.

The constraints put a threshold both on U and C, which prevents a solution that: 1) minimizes the unavailability without considering an availability threshold, and 2) exceeds the cost normalization threshold of 1. We leave the formalization of U outside the problem statement. We do this definition in Section 4.1.

4 Method

Software systems are constantly exposed to performance degradation, failures, and load variations, which can negatively impact the quality of service. These issues can lead to resource exhaustion, resulting in decreased availability and, in extreme cases, service-level outages [38]. To optimize equation (4) and prevent service outages from overloaded services while managing costs, we propose a concrete approach for computing the unavailability term, U, before using a heuristic method for the optimization.

4.1 Availability Function

To model the availability of a workflow of a microservices architecture, we consider that services execute in sequence and that each one of them may have more than a single server running in parallel and receiving requests according to a round-robin discipline. As shown in Fig. 1, this system represents a scenario where users

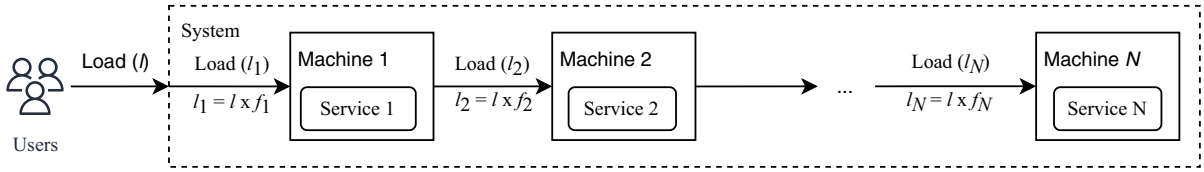


Fig. 1 Series system workflow

generate load with their requests, and each request runs through the entire system from end to end. The failure of any component leads to a service failure, hence, all components are strictly necessary for a given workflow.

In cases where a service features multiple endpoints, our approach addresses this considering that each request follows a sequential path through the entire system; however, this approach does not consider workflow cycles where services call themselves using different endpoints.

Equation (5) shows the availability function for a series system composed of n services, where A is the availability of the system computed by multiplying the availability of each service, A_i, and the availability of each machine (virtual or otherwise), A_{Mi}, which is agreed with the cloud provider and is usually documented in Service-Level Agreements (SLAs), and for which we consider a constant value of 0.9999 [39] in our evaluation:

$$A = A_1 \times A_{M1} \times A_2 \times A_{M2} \times ... \times A_n \times A_{Mn} \quad (5)$$

Availability is highly dependent on the load, as a system may struggle to respond within useful time limits if the load is too high. Furthermore, services must scale in and out to accommodate the existing load. Hence, the availability of equation (5) is a function of the ingress load of the workflow, in requests per second. To maximize availability, we need to have some means to compute the dependency of this metric with the number of replicas of the internal services. Hence, we need to first model the availability of the internal services as a function of their replication.

To model the relationship between the load and the availability of one service, A_i in equation (5), we use the GLF, which we define in equation (6), a mathematical function often used to model growth and decay over time [40]:

$$y(\lambda) = a + \frac{k-a}{\left(1 + qe^{-b\lambda}\right)^{1/v}} \quad (6)$$

Equation (6) comprises six parameters: 1) λ, which represents the load, 2) a, the left horizontal asymptote, 3) k, the right horizontal asymptote, 4) b, the growth rate, 5) q, is related to the value $y(0)$, and 6) v, the maximum growth near the asymptotes. Figure 2 shows an example of a GLF with the parameters $a = 0.999999$, $k = 0$, $b = 0.8$, $q = 1$, and $v = 0.0005$.

To demonstrate the suitability of the GLF, we applied synthetic load and conducted a benchmark on multiple services of the Robot Shop, one of our case study microservice applications (refer to Section 5). These benchmarks were executed on a single system service with the remaining services scaled out to the maximum number of replicas (to avoid bottlenecks) and varying the service replication factor, r, between 1 and 5. For this case, each replica was allocated 1 vCPU and 2 GiB of memory to ensure accurate results. We collected results of availability (i.e., response status and response times) and requests per second, to create a profile of the behavior of replicated services under different load conditions, allowing us to select the optimal number of replicas for a given service under a given load.

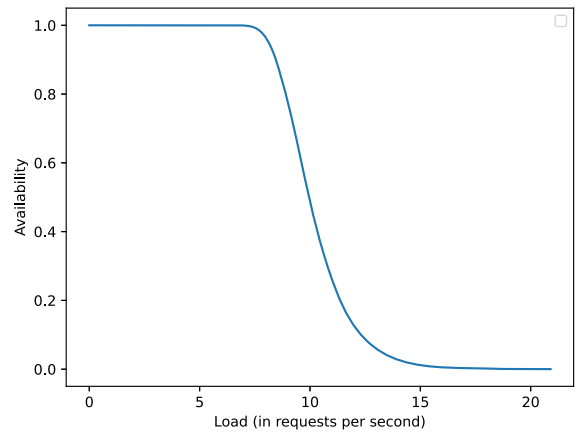


Fig. 2 GLF ($a = 0.999999$, $k = 0$, $b = 0.8$, $q = 1$, and $v = 0.0005$)

Figure 3 presents the results of an experiment to evaluate the relationship between the availability, the number of replicas, and the load (in requests per second) using the catalogue service of our case study microservice application. The availability is measured on a range [0, 1], where 1 represents a fully operational service and 0 represents a completely unavailable service.

Availability is quite high up to a certain load level, where it degrades very quickly before reaching 0. As more computational resources are available to attend to the load, the availability of the service increases as well, a clear indication that as the load on the service increases, it becomes increasingly important to have multiple replicas to maintain a higher level of availability. As we show experimentally, the GLF enables the prediction of the behavior of the service under different loads (refer to Fig. 2), thus providing important insights for determining the optimal number of replicas for a given service.

Using an $M/M/c$ queue, also known as Erlang-C model, could also be a possibility for the model. $M/M/c$ refers to arrivals generated by a Poisson process, exponential job service time distributions, and c replicas. The availability of the service would be the probability of having a response within some time constraint t. With only one server, i.e., $c = 1$, the availability would be as follows [41]:

$$y(\lambda) = P(W \leq t) = 1 - e^{-(\mu-\lambda)t}$$
$$\text{where } \lambda \leq \mu \text{ and } y(\lambda) = 0, \text{ for } \lambda > \mu \quad (7)$$

Here, W is the sojourn time (time waiting in queue plus time in the server), μ is the service rate, and λ is the arrival rate of requests. This availability function requires two branches and is not differentiable at $\lambda = \mu$. Figure 4 illustrates an M/M/c queue across a range of scenarios, varying c from 1 to 4 servers (or replicas). In each scenario, the arrival rate (λ) spans from 1 to 20, while maintaining a constant service rate ($\mu = 5$), for the time limit of 1 ($t = 1$).

Moreover, this queuing-based approach presents a fall of availability near $\lambda = \mu$ that is even sharper than the one presented by the Generalized Logistic Function (GLF). The latter seems to be more in line with our observations of Fig. 3. Finally, since supporting multiple replicas, i.e., $c > 1$, involves a considerably more complex expression when compared to the GLF, we opted for this latter.

4.2 Modeling the Availability of Replicated Services

In our study of availability modeling for cloud services, we observed a non-linear evolution of service response time as the number of replicas increased. To analyze the distribution of response time by the number of replicas, we performed experiments where we applied load to one service for 5 minutes and collected the response times of each request. The load was constant during each experiment and varied from very low, at a rate of 1 request per second, to very high, at a rate of 5000 requests per second. The number of replicas varied from 1 to 5 with a step of 1.

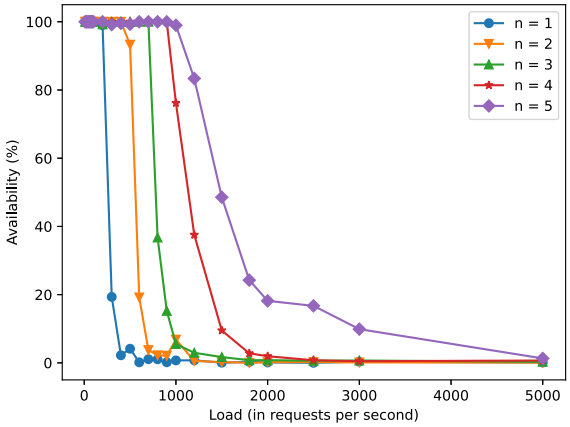


Fig. 3 Empirical observations of availability, the number of replicas, and the load on the service catalogue of Robot Shop

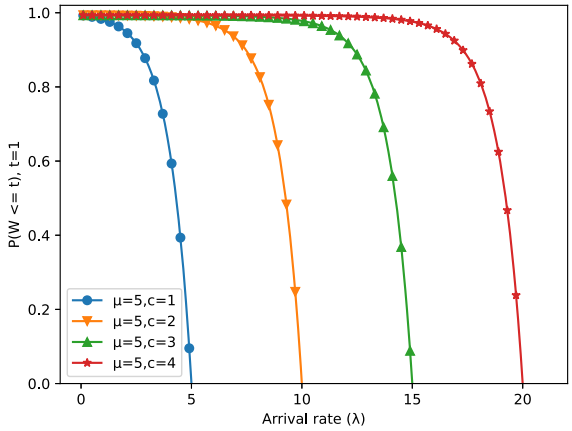


Fig. 4 M/M/c curves

Figure 5a presents the distribution of the response time, in milliseconds, by the number of replicas for one service. As more replicas are added to a service, the response time decreases, reflecting the increased capacity to handle requests; however, as the number of replicas increases, the response time gain becomes less pronounced. This non-linear behavior is likely due to the complexity and overhead of coordinating requests and responses among multiple replicas. Additionally, contention for shared resources such as network bandwidth and storage can also contribute to diminishing returns in response time as the number of replicas increases.

Thus, our model needs to take into account the non-linear response time evolution of service replicas to accurately predict availability. Amdahl's law states that the speedup of execution time depends on the amount of code that cannot be parallelized [16], and is computed in equation (8), which comprises two parameters: 1) n, the number of processors, or service replicas, and 2) p, the percentage of code that can be made parallel:

$$S(n, p) = \frac{1}{1 - p + \frac{p}{n}} \tag{8}$$

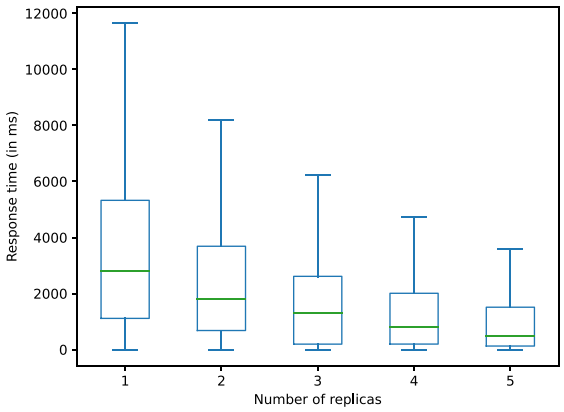
Figure 5b shows the non-linear relationship between the number of replicas and the resulting response time speedup of the service catalogue from Robot Shop, one of our case study applications. The x-axis represents the number of replicas and the y-axis represents the speedup.

To calculate the speedup for each number of replicas, we used the average response times for each number of replicas and divided it by the average response time with only 1 replica. The p value of 0.8525 was obtained by fitting equation (8) to the data points of the response time distribution of Fig. 5a.

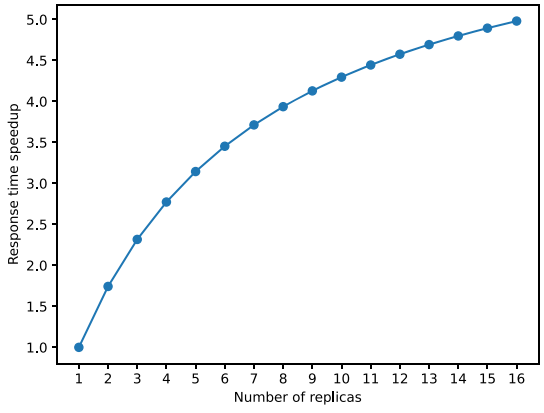
By merging (8) with the GLF, λ is replaced by $\lambda/S(n, p)$, and we obtain the following availability function:

$$y(\lambda, n, p) = a + \frac{k - a}{(1 + qe^{-b(\lambda/S(n,p))})^{1/v}} \tag{9}$$

The results from the availability function approximations using services from our case study applications are presented in Section 5.2.



(a) Response time distribution.



(b) Response time speedup, with ($p = 0.8525$).

Fig. 5 Non-linear relationship of service response time and resulting speedup for the service catalogue of Robot Shop. (a) Response time distribution. (b) Response time speedup, with ($p = 0.8525$)

5 Case Study

To evaluate the efficiency of CAAS, we performed experiments with two case study applications, 1) Robot Shop [13] and 2) Sock Shop [14], and compared results of availability and cost with the widely used Kubernetes CPU-based Horizontal Pod Autoscaler (HPA). Robot Shop microservices application is an online application selling robots and products with artificial intelligence, composed of 12 microservices. Sock Shop is a microservices demo application to simulate a user-facing part of an online shop that sells socks, composed of 14 microservices.

For the first application, Robot Shop, we selected three workflows: 1) *List products*, 2) *Add item to the cart*, and 3) *Rate item*, and for the second application,

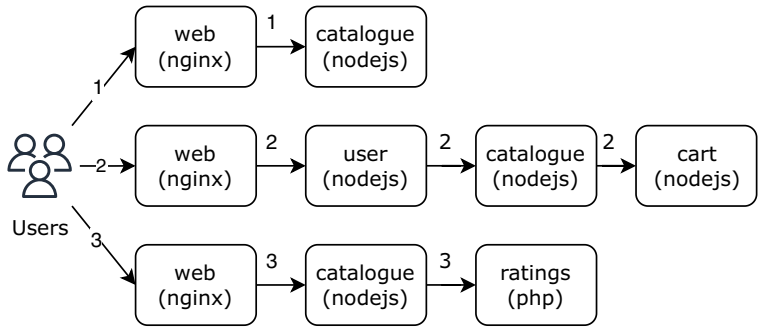


Fig. 6 Workflows of Robot Shop microservices application: 1) *List products*, 2) *Add item to the cart*, and 3) *Rate item*

Sock Shop, we selected three workflows: 1) *List products*, 2) *Add item to the cart*, and 3) *Order an item*. Our solution optimizes one workflow at a time. Figures 6 and 7 depict these workflows for Robot Shop and Sock Shop respectively.

Regarding Robot Shop, the first workflow, *List products*, is used to list all products available in the shop and uses web and catalogue services; The second workflow, *Add item to the cart*, is used to add an item to the cart of a user and uses web, user, catalogue and cart services; and the third workflow, *Rate item*, is used to rate an item in the shop and uses web, catalogue, and ratings services.

Regarding Sock Shop, the first workflow, *List products*, is used to list all products available in the shop and uses front-end and catalogue services; The second workflow, *Add item to the cart*, is used to add an item to the cart of a user and uses front-end, user, catalogue and carts services; the third workflow, *Order an item*, is used to execute the whole process to order and buy an item in the shop and uses front-end, user, catalogue, carts, orders, shipping and payment services.

In our experiments, we used Locust [42], a load generator, to produce constant loads, in terms of requests per second (rps), and emulate user behavior, to simulate the actions of real users browsing, listing, rating and purchasing products on the platforms, as defined in the workflows of Figs. 6 and 7. The experiments were conducted in a local node running a Kubernetes cluster with an Intel Xeon Gold CPU 6226R with a clock speed of 2.90 GHz with 16 vCPUs and 128 GiB of RAM.

5.1 Implementation Details

To implement the method described in the previous section, we need to code the problem as defined in equation (4). For this, we resorted to Pymoo [43], a framework that offers state-of-the-art optimization algorithms for single, multi and many-objective problems. We selected the single-objective Genetic Algorithm (GA) [44] which is suitable for our problem having one objective and two constraints. However, our problem involves discrete variables represented by the number of replicas of services, with valid values ranging from 0 to n (the maximum amount of replicas). We were able to leverage the extensibility of the GA class in the Pymoo framework and modify the sampling, crossover and mutation operators to work with integer values. This allowed us to tailor the algorithm to our specific needs, ensuring that it handles the dis-

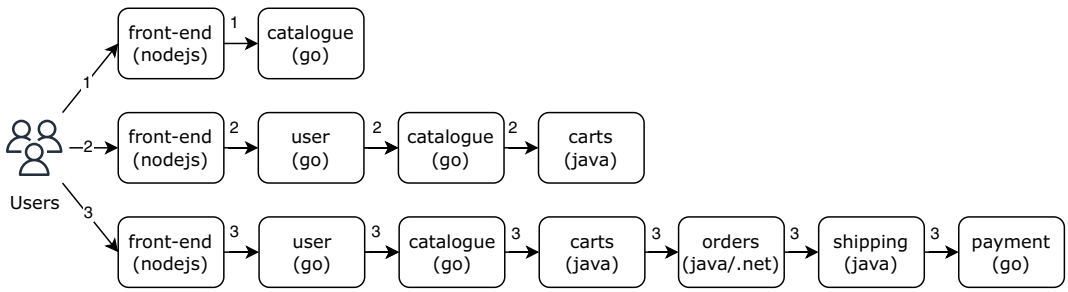


Fig. 7 Workflows of Sock Shop microservices application: 1) *List products*, 2) *Add item to the cart*, and 3) *Order an item*

Table 1 GA algorithm parameters

Parameter	Value
Population size	100
Number of offsprings	10
Sampling	Random Integers $\in \mathbb{N}_0$
Crossover probability	SBX with 0.9 [45]
Mutation probability	PM with 1.0 [45]
Eliminate duplicates	True
Return Least feasible solution	True

crete nature of the replicas configuration output. Table 1 summarizes the parameters for the execution of the GA.

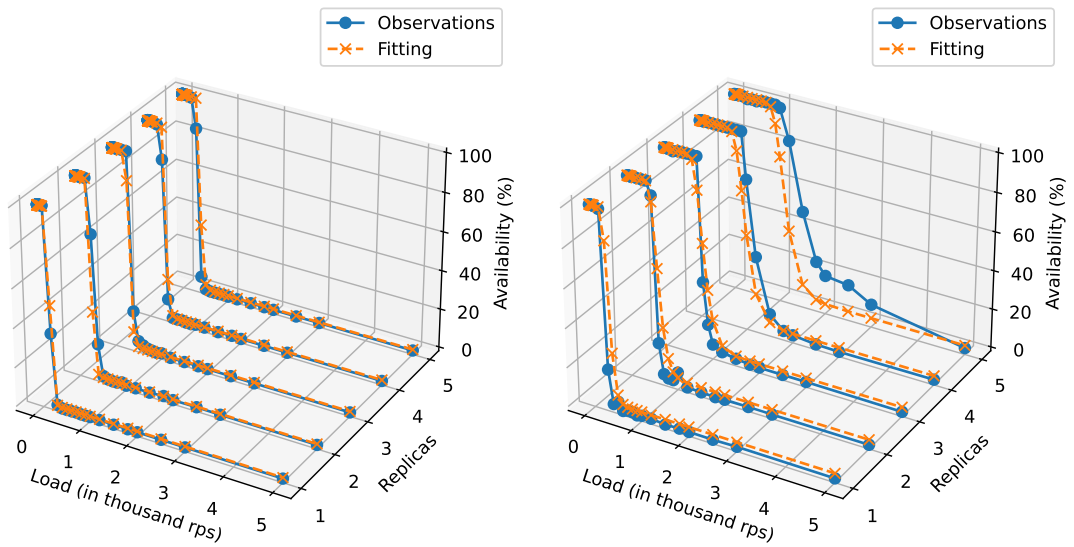
We defined the *Problem*, an object-oriented definition from Pymoo that allows one to specify details such as the number of objectives, the number of constraints, and the number of variables of an optimization problem and implement a method to evaluate a set of solutions. The problem consists of one objective, $n\_obj = 1$, where we want to minimize the linear combination of unavailability and cost. Regarding constraints, the optimization is subject to two inequality constraints, $n\_constr = 2$. To achieve our optimization goals, we minimize the problem using the built-in Pymoo method *minimize* with the problem and the aforementioned defined algorithm to compute the optimal solution.

In our implementation, the connection between CAAS and the microservices is established through the use of the Kubernetes command-line tool to define the number of replicas for each microservice. This is accomplished by adjusting the size of the deployment, which subsequently defines the number of microservice replicas. The microservice architecture and the resources allocated for one replica of each microservice are defined statically as a configuration within CAAS.

5.2 Availability Results

To evaluate our availability model, we profiled the endpoints of the 5 services of Robot Shop required for the 3 workflows individually: cart, catalogue, ratings, user, and web, and the 7 services of the Sock Shop required for the 3 workflows individually: catalogue, carts, front-end, user, orders, shipping and payment.

To profile Robot Shop, we conducted 1 run for each service (5 services) and 1 run for each replica (ranging from 1 to 5 with a step of 1) at each load level (40 load levels from 1 request per second to 5000), resulting in a total of 40 runs for each load level. In total, we executed 1000 runs (5 services $\times$ 5 replicas $\times$ 40 load levels). To profile Sock Shop, we conducted 1 run for each



(a) Cart service. (b) Catalogue service.

Fig. 8 Availability function fitting of two services of Robot Shop. (a) Cart service. (b) Catalogue service

service (7 services) and 1 run for each replica (ranging from 1 to 5 with a step of 1) at each load level (65 load levels from 1 request per second to 5000). In total, we executed 2275 runs (7 services $\times$ 5 replicas $\times$ 65 load levels).

Each run to profile one service had a duration of 10 minutes, 5 minutes of execution, and 5 minutes of intervals between experiments to let the system pause, restart all configurations, i.e., delete every deployment, re-deploy and start a new run. Taking this into account, the total duration of the profiling of each service of Robot Shop was around 7 days and around 15 days for Sock Shop.

During the profiling of each service, the replication factor was set to 1 and all the remaining components were scaled out to the maximum allowed by the infrastructure so that they would not influence the results. Regarding resources, the replication factor set to 1 states that each service of Robot Shop is configured with the limit of 1 vCPU for the CPU and 2 GiB for

Table 2 Robot Shop: Availability estimation results using R-squared, MAE, and MSE

Service	p	n	R-squared	MAE	MSE
cart	0.5932	1	0.9976	0.0074	0.0006
		2	0.9774	0.0188	0.0052
		3	0.9957	0.0115	0.0010
		4	0.9953	0.0109	0.0011
		5	0.9876	0.0163	0.0029
Average			0.9907	0.0130	0.0022
catalogue	0.8525	1	0.9187	0.0478	0.0191
		2	0.9632	0.0389	0.0086
		3	0.9778	0.0332	0.0047
		4	0.9489	0.0393	0.0087
		5	0.8266	0.0700	0.0204
Average			0.9270	0.0458	0.0123
ratings	0.3764	1	0.7985	0.0967	0.0409
		2	0.9821	0.0396	0.0039
		3	0.8418	0.0797	0.0338
		4	0.9542	0.0476	0.0097
		5	0.9739	0.0443	0.0057
Average			0.9101	0.0616	0.0188
user	0.6274	1	0.8856	0.0681	0.0274
		2	0.9745	0.0404	0.0053
		3	0.9483	0.0471	0.0092
		4	0.9341	0.0462	0.0112
		5	0.9629	0.0434	0.0074
Average			0.9411	0.0490	0.0121
web	0.8334	1	0.9337	0.0574	0.0107
		2	0.9475	0.0447	0.0072
		3	0.8746	0.0607	0.0268
		4	0.9291	0.0516	0.0163
		5	0.8774	0.0399	0.0091
Average			0.9083	0.0509	0.0140

the memory by enforcement, i.e., the system prevents the service within the container from ever exceeding the configuration limits, and that each service of Sock Shop is configured with the limit of 0.5 vCPU for the CPU and 1 GiB for the memory.

This profiling allowed us to extract the value of the variables for equation (9) and generate the model representation of the availability of each service. With the availability of each service modeled, we compared the availability results with the model estimates and the real system using the following metrics [46]: coefficient of determination, R-squared, the Mean Absolute Error (MAE) and the Mean Squared Error (MSE) to assess the results of the availability model estimates.

Figure 8a and b show the observed availability data, marked with dots, and the model estimates, marked with crosses, for two services, cart and catalogue of Robot Shop. In these figures, the proximity of the crosses to the dots illustrates the degree to which the model aligns with the actual availability trends, providing a visual representation of the model's fitting performance.

Furthermore, Tables 2 and 3 show all results of availability model estimates for the 5 services of the Robot Shop and the 7 services of the Sock Shop case study applications. Each service has a p, which lists the percentage of code that can be made parallel and, in the case of Table 2, an n, which lists the number of replicas for each service used in the study, as well as averages of R-squared, MAE, and MSE results. In the latter table, we omitted the n column and computed averages of R-squared, MAE, and MSE results for brevity.

R-squared lists the coefficient of determination which is a measure of the goodness of fit of the model, with a value of 1 indicating a perfect fit. MAE lists the mean absolute error, a measure of the average magnitude of the errors in a set of predictions. MSE lists the mean squared error, a measure of the average of the squares of the errors. Additionally, the consistently low MAE and MSE values across all replicas for all services, even negligible in some cases, further indicate the models' accuracy; however, some services may exhibit lower R-squared values, implying a less precise fit.

The reasons behind these variations could be multifaceted. Factors like the complexity of the service, the nature of the workload, or the availability of historical data may influence the model's performance. Services with inherently unpredictable or sporadic behavior might naturally yield lower R-squared values. As such, future refinements in the modeling approach may be tailored to specific service characteristics, enabling even better predictability in scenarios where the models exhibit lower R-squared values. Overall, results for all services of both applications show that the models have a good fit and high accuracy regardless of the percentage of code that can be made parallel and the number of replicas.

In summary, our analysis demonstrates that while some services exhibit exceptional fit with our availability model, others may present challenges due to their unique characteristics. However, the overall results indicate that our model is capable of accurately predicting the availability of services for the two case study applications.

5.3 Cost Results

To determine the coefficients for the model and minimize the cost, we analyzed Amazon Web Services (AWS) data tables for on-demand hourly rate instance pricing of C5/R5 (compute optimized/memory opti-

Table 3 Sock Shop: Availability estimation results using averages of R-squared, MAE, and MSE

Service	p	R-squared	MAE	MSE
catalogue	0.7762	0.9378	0.0498	0.0829
carts	0.8525	0.9174	0.0484	0.0165
front-end	0.9017	0.9178	0.0487	0.0164
payment	0.0437	0.8409	0.0649	0.0172
orders	0.9462	0.8939	0.1041	0.0427
shipping	0.8862	0.9184	0.0480	0.0164
user	0.2189	0.9031	0.1158	0.0516

mized) instances of Elastic Compute Cloud (EC2). We performed multiple linear regression on the relationship between the dependent variable (hourly rate) and two independent variables (vCPU and memory). The results of the multiple linear regression on the AWS data showed a regression statistic value of 1 for R^2 and 3.07^{-7} for standard error, allowing for accurate predictions with a small error. The ANOVA significance F value was approximately equal to zero, indicating significant results. The coefficients found were 0.0427 cents per vCPU and 0.0039 cents per GiB of memory, which replace c_c and c_m in the proposed cost equation (refer to equation (3)). The cost equation was then utilized to compute the monetary cost of each configuration for every experiment in both case study applications, allowing for result comparisons (refer to Section 6).

5.4 Performance Evaluation

The performance, i.e., execution time, of an algorithm to control the number of replicas of a microservices' application is important as it needs to be able to suggest a configuration within an acceptable time. To evaluate the performance of CAAS, we measure the results of the execution time of the optimization step for each experiment.

Table 4 presents the results of execution time for CAAS for each application. The optimization was computed in one node with an Intel Core i5-8279U CPU with a clock speed of 2.40GHz with 4 vCPUs and 8 GiB of memory. The execution time to compute the optimal configuration averaged 2.048 seconds for Robot Shop and 2.297 seconds for Sock Shop; however, Sock Shop exhibited longer times, with its 75th percentile and maximum execution times being higher, owing to its more extensive workflows, which led to a larger objective space to explore. By default, monitoring applications installed in popular container management systems, e.g., Kubernetes and OpenShift, are configured to collect time series data every 15 seconds. The autoscaler used by these management systems uses the information collected by the monitoring mechanisms and is thus limited to adapting the number of nodes every 15 seconds. Taking this into account, we can state that CAAS can suggest an optimal configuration below the collection time of monitoring mechanisms used in container management systems.

Table 4 Execution time (in milliseconds) of the optimization step for each application

	Robot Shop	Sock Shop
mean	2048	2297
min	1488	1799
25%	1896	1928
75%	2202	2876
max	2636	3786

6 Comparative Analysis of Results

In this subsection, we present a comparison of our method, CAAS, an optimization approach for cloud services autoscaling, with the industry standard CPU-based Kubernetes Horizontal Pod Autoscaler (HPA) used in containerized applications, hereafter referred to as Autoscaler (AS) for brevity.

To evaluate them, we obtained availability and cost data and compared the results. For CAAS, we computed the optimization using the system's model for each workflow of each application, conducting a total of 120 experiments (40 load levels $\times$ 3 workflows) for the Robot Shop and 195 experiments (65 load levels $\times$ 3 workflows) for the Sock Shop, to produce the optimal system configurations selected by the model. From these configurations, we calculated the cost and availability for each experiment. For the AS approach, we defined and applied the autoscaler policies at the application level using the Kubernetes HPA, and we conducted an identical set of 120 experiments for the Robot Shop and 195 experiments for the Sock Shop. The AS was configured with a threshold of 80% for each service, which means that the Central Processing Unit (CPU) utilization is monitored and if it exceeds the threshold of 80%, a new replica of the service is initiated [47]. The maximum number of replicas was set to the maximum allowed by the infrastructure for both approaches, i.e., 16 vCPUs and 128 GiB of RAM.

Regarding the CPU threshold, we have chosen 80% threshold for the Kubernetes HPA to have a balance between cost and performance. Additionally, we have experimented with 90, 70, and 20%. Regarding the 90% threshold, the availability was very low, maybe due to the services being close to the very limit of 100%. On the contrary, when we tested with the 70% threshold,

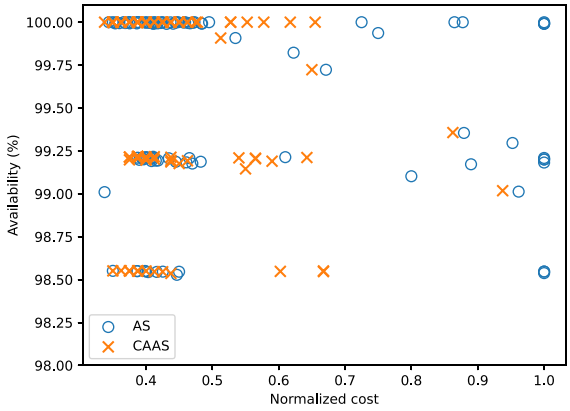


Fig. 9 Availability and Cost of both approaches, the AS and CAAS

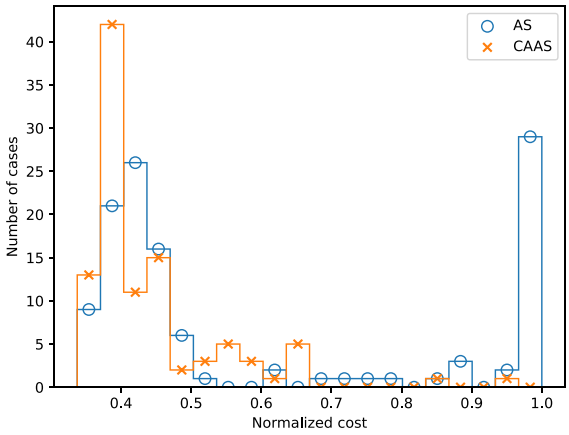


Fig. 10 Cost comparison between the AS and CAAS

the costs started to increase notably, outweighing the marginal benefits in terms of availability and worsening the case for the Kubernetes HPA. Finally, we explored a very low threshold of 20%; however, in this scenario, the system availability was very low, and costs were consistently maximized, i.e., the normalized cost was 1.0.

6.1 Availability and Cost Comparisons

Figure 9 depicts a comprehensive overview of the results obtained for the Robot Shop application, comparing both the AS approach, denoted in blue dots, and our proposed CAAS approach, denoted by crosses. Our analysis revealed several key findings regarding the effectiveness of these approaches. We found that, regarding availability, both approaches delivered similar results. This suggests that the various configurations and approaches tested were all successful in achieving and maintaining high availability levels.

Consequently, it implies that further fine-tuning in this area may not yield significant improvements; however, a more detailed examination regarding cost, uncovered instances where the AS approach encountered limitations (refer to Fig. 10 where we present a more in-depth overview of the cost comparison between both approaches). Notably, in certain scenarios, the AS algorithm fully allocated all available resources, resulting in configurations that were not well-suited to handle the imposed load level. These points are represented on the far right side of the plot, where the values of the normalized cost are high. This disparity highlights the potential risks of relying solely on an autoscaler, as it may lead to unexpected and costly resource usage occupying the available resources while only considering the CPU metric. In contrast, CAAS achieves consistently lower costs while maintaining high levels of availability.

Despite the similar availability values, our CAAS approach consistently outperformed AS in terms of cost-effectiveness achieving significantly cheaper configurations, as indicated by the leftward shift of CAAS data points.

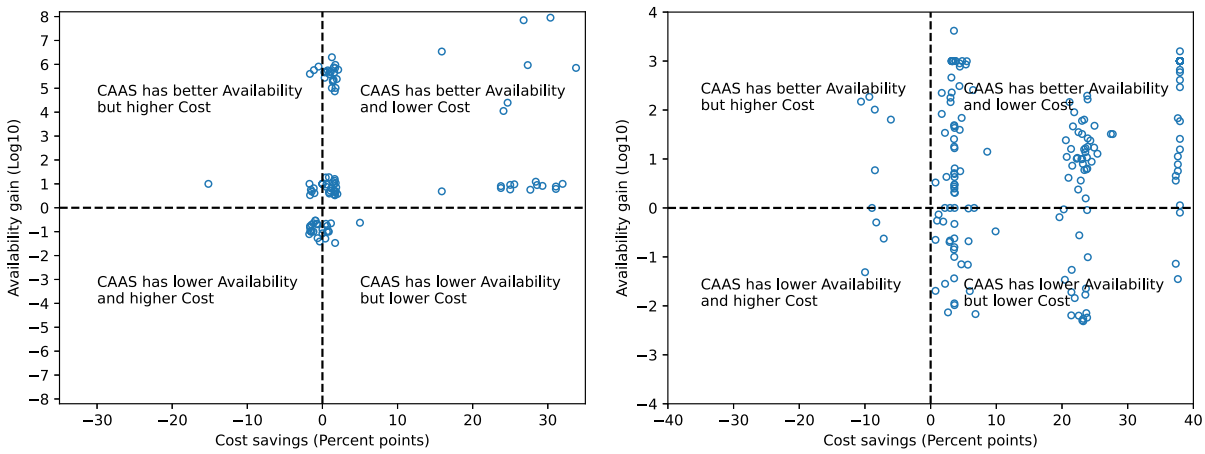
This cost-efficiency advantage of CAAS is particularly noteworthy since it demonstrates that it not only achieves similar levels of availability but does so while utilizing resources more effectively. The implication here is that CAAS optimizes the allocation of resources, preventing over-provisioning and, consequently, reducing operational expenses.

In summary, while both approaches exhibited comparable availability results, CAAS demonstrated superior cost-effectiveness by avoiding resource exhaustion and delivering more economically efficient configurations.

6.2 The differences in Availability and Cost

To further compare Autoscaler (AS) to CAAS, we computed differences in both availability and cost for both approaches. For the availability values, we applied the following formula to compute the gain in the number of 9s (nines):

$$A_{\text{gain}} = -\left(\log_{10}(1-A_{\text{CAAS}}) - \log_{10}(1-A_{\text{AS}})\right) \quad (10)$$



(a) Robot Shop. (b) Sock Shop.

Fig. 11 Availability gain and Cost savings of CAAS over the AS approach for both case study applications. (a) Robot Shop. (b) Sock Shop

Equation (10) is used to compute the difference in the number of 9s between the availability values of the two approaches. This way we are able to compare the availability values of both approaches in a more intuitive way, as the number of 9s is a common metric used to measure availability. For example, a difference of 1 means that the availability of the CAAS approach achieved 1 more 9 than the AS approach. A difference of 0 means that both approaches achieved the same availability, and a difference of -1 means that the AS approach achieved 1 more 9 than the CAAS approach.

Figure 11a and b show the observations of the computed differences between the availability and cost results for both approaches for the Robot Shop and Sock Shop respectively, on a plane divided into four quadrants. These quadrants divide results into the following four cases:

- Upper right quadrant: CAAS has better Availability and lower Cost than the AS approach.
- Upper left quadrant: CAAS has better Availability but higher Cost than the AS approach.
- Lower left quadrant: CAAS has lower Availability and higher Cost than the AS approach.
- Lower right quadrant: CAAS has lower Availability but lower Cost than the AS approach.

The best scenario for CAAS is exemplified by observations within the first quadrant, where it achieves both higher availability and lower cost. Following this, observations within the second and fourth quadrants show instances where CAAS is slightly better in either higher availability or lower cost, at the expense of the other objective. Finally, the third quadrant represents the least favorable scenario for CAAS, characterized by observations with lower availability and higher cost, representing the cases where CAAS faces a disadvantage. Table 5 presents a summary of the comparison of the availability and cost differences of both approaches.

For Robot Shop, results show a mean improvement of 1.61 9s of availability and cost savings of 5.84%.

Table 5 Comparison summary of CAAS with the AS

	Robot Shop		Sock Shop	
	Availability gain (Log10)	Cost saving (%)	Availability gain (Log10)	Cost saving (%)
mean	1.61	5.84	1.15	17.97
min	-1.47	-15.18	-2.31	-10.60
max	7.95	33.76	3.62	37.95

For this application, in the worst-case scenarios, we observed an availability detriment of 1.47 9s and an increase in costs up to −15.18%. It is noteworthy that the latter value can be considered an outlier (refer to the upper left quadrant in Fig. 11b) with the majority of worst-case scenarios showing only small differences. Finally, regarding best-case scenarios, we observed an improvement of up to 7.95 9s of availability, and cost savings up to 33.76%.

For Sock Shop, results also reveal a notable increase in availability, with a mean gain of approximately 1.15 9s and cost savings of 17.97%. These findings indicate that CAAS effectively enhances both availability and cost-effectiveness for the average case in this application. In the most challenging scenarios encountered during our experiments, we observed a decrease in availability, with the minimum availability gain reaching as low as −2.31 9s. In these scenarios, cost savings were also negatively impacted, with the minimum cost saving recorded at −10.60%. Conversely, in the best-case scenarios, we observed substantial improvements in availability, with the maximum gain reaching 3.62 9s, while cost savings also demonstrated remarkable performance, with the maximum recorded at 37.95%.

Table 6 shows the counting of the experiment results grouped by quadrants. For Robot Shop, 82 of the 120 cases lie in quadrant 1, where CAAS wins on both criteria. Quadrant 3, where CAAS loses on both criteria has 18 cases. Quadrants 2 and 4, characterized by a draw between both approaches, show 11 and 9 cases, respectively. For Sock Shop, 142 of the 195 cases lie in quadrant 1. Quadrant 3 has 3 cases. Finally, Quadrants 2 and 4, show 5 and 45 cases, respectively.

6.3 Discussion and Observations

Based on the values observed in the experiments for the Robot Shop and Sock Shop applications, there are several noteworthy points to discuss:

Availability improvement The results demonstrate that, on average, CAAS leads to an increase in availability compared to the traditional approach (AS). This suggests that CAAS has the potential to enhance the overall system's availability, which can be particularly important for applications where uptime is critical.

Cost savings The cost savings achieved with CAAS indicate that the approach can also be cost-effective. This means that while maintaining or even improving availability, it can do so at a reduced cost, which is an important factor in optimizing resource allocation and budget management.

Worst-case scenarios In the most challenging scenarios, where system demands may be at their peak or unusual conditions prevail, we observed a decrease in availability when using CAAS. Additionally, in these scenarios, there were instances of negative cost savings. It is important to highlight that these negative values fall outside the typical range of cost savings. This suggests that while CAAS generally performs well, there may be exceptional cases where it could lead to temporary dips in availability and unexpected cost increases.

Best-case scenarios On a more positive note, in the best-case scenarios, CAAS showed substantial improvements. This highlights the potential for significant performance enhancements and cost reductions when the system operates under optimal conditions.

Table 6 Comparison with the AS: Evaluation of scenarios

Quadrant	Outcome	Robot Shop		Sock Shop	
		Count	%	Count	%
Upper right quadrant (Q1)	Win	82	68.33	142	72.82
Upper left quadrant (Q2)	Draw	11	9.17	5	2.56
Lower left quadrant (Q3)	Loose	18	15.00	3	1.54
Lower right quadrant (Q4)	Draw	9	7.50	45	23.08

In summary, the majority of cases show that CAAS achieves better results when compared to the Central Processing Unit (CPU)-based AS, demonstrating the effectiveness in achieving cost-efficient resource usage. These results underscore the value of a cost-aware approach to resource management and show the effectiveness of CAAS approach in achieving this goal by being cost and availability-aware.

7 Limitations and Threats to Validity

In this section, we outline the limitations and potential threats to the validity of CAAS, providing an overview of factors that may impact the interpretation and generalization of our findings.

Instrumentation and monitoring One significant limitation of CAAS is its reliance on extensive application instrumentation and monitoring. This necessity involves metrics such as the average response time for each service and computing availability using these values. However, not all applications are prepared for this level of instrumentation, and some applications may lack any instrumentation entirely. As a consequence, implementing this method can become expensive, both in terms of the initial investment required to instrument the application and the ongoing maintenance costs associated with collecting and analyzing the necessary data.

Load scenario abstraction The load scenarios were designed to mirror real user load patterns and behaviors; however, it is important to note that these tests were executed without real users. While these scenarios aim to simulate real-world conditions as accurately as possible, the absence of real data may introduce a level of abstraction that could impact the accuracy and comprehensiveness of the test results.

Response time sensitivity to parameters Furthermore, it is important to recognize that response times can be significantly influenced by the specific invocation parameters used during testing and variations in these parameters, such as input data or request configurations, may lead to different performance outcomes. Therefore, when interpreting response time data from these tests, one needs to consider the potential impact of parameter variations and acknowledge that real-world usage may exhibit a wider range of response times due to the dynamic nature of user inputs and system interactions.

Service scope We excluded databases and queue services from our analysis. While our focus on services with Hypertext Transfer Protocol (HTTP) endpoints addresses specific aspects of our research objectives, we acknowledge this limitation. Additionally, we do not consider workflow cycles where services call themselves using different endpoints. We intend to address and incorporate these components as part of future research to provide a more comprehensive analysis of system performance.

8 Conclusion

In this paper, we present CAAS, an efficient autoscaling approach for cloud services that significantly improves both availability and cost objectives in the majority of cases, compared to the conventional CPU-based AS. Evaluation results show an improvement between one and two 9s in availability and a reduction of replication costs of about 6% for the first case study application (Robot Shop), and an average improvement of over one 9 while reducing costs by approximately 18% for the second case study application (Sock Shop). Also, the CAAS approach improves both objectives simultaneously in around 68% of cases for Robot Shop, and around 73% of cases for Sock Shop.

Moreover, in our evaluation, CAAS operates at a faster pace than the frequently used collection period of the monitoring system, allowing it to be applied to scale cloud services. Our study suggests that CAAS can be useful in practice for autoscaling cloud services. Further research, including prioritizing objective functions using weights, optimizing multiple workflows concurrently, and considering additional metrics such as queue lengths and custom application metrics, might enhance its performance in specific scenarios.

Acknowledgements The authors would like to thank the Centre for Informatics and Systems of the University of Coimbra (CISUC) for providing the conditions for this work. This work is funded in part by the Portuguese Foundation for Science and Technology (FCT) through Doctoral Grant No. BD.06012.2021, and in part by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020.

Author contributions Andre Bento, Filipe Araujo and Raul Barbosa conceptualized the solution proposed in the paper and designed the experimental evaluation. While all have contributed to the experimental design, Andre Bento executed the experiments and obtained and processed the results. All authors discussed the objectives and the evaluation of results, the observations and the conclusions. Andre Bento drafted the complete paper and Filipe Araujo and Raul Barbosa revised and updated it. All authors approved the final manuscript.

Funding Open access funding provided by FCT|FCCN (b-on). This work is funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

Data Availability The datasets generated during the current study are available from the corresponding author upon reasonable request.

Declarations

Consent for Publication All authors gave their consent for this publication.

Competing Interests The authors declare no competing interests.

References

1. Nickoloff, J., Kuenzli, S.: Docker in Action. Simon and Schuster (2019)
2. Senthil Kumaran, S.: Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. Springer (2017)
3. Luksa, M.: Kubernetes in Action. Simon and Schuster (2017)
4. Qian, L., Luo, Z., Du, Y., Guo, L.: Cloud computing: an overview. In: IEEE international conference on cloud computing, pp. 626–631. Springer (2009)
5. Low, C., Chen, Y., Wu, M.: Understanding the determinants of cloud computing adoption. Ind. Manag, Data Syst (2011)
6. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. https://martinfowler.com/ (2014)
7. Newman, S.: Building Microservices. O'Reilly Media, Inc. (2021)
8. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and ulterior software engineering, pp. 195–216 (2017)
9. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: how to make your application scale. In: Perspectives of system informatics: 11th international Andrei P. Ershov informatics conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11, pp. 95–104. Springer (2018)
10. Amazon Web Services, Inc.: Amazon cloudwatch metrics for Amazon EC2 auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-metrics.html
11. Chen, T., Bahsoon, R., Yao, X.: A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. ACM Comput. Surv. (CSUR) **51**(3), 1–40 (2018)
12. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Softw. **35**(3), 24–35 (2018)
13. Instana: Robot shop: sample microservice application. https://github.com/instana/robot-shop (2018). Accessed: 10 Feb 2023
14. Weaveworks: Sock shop: a microservice demo application. https://microservices-demo.github.io (2017). Accessed: 17 Aug 2023
15. Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., Kim, S.: Horizontal pod autoscaling in Kubernetes for elastic container orchestration. Sensors **20**(16), 4621 (2020)
16. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference AFIPS '67 (Spring), pp. 483–485. Association for Computing Machinery (1967)
17. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**, 559–592 (2014)
18. Bauer, A., Lesch, V., Versluis, L., Ilyushkin, A., Herbst, N., Kounev, S.: Chamulteon: coordinated auto-scaling of micro-services. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 2015–2025. IEEE (2019)
19. Srirama, S.N., Adhikari, M., Paul, S.: Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. **160**, 102629 (2020)
20. Singh, P., Kaur, A., Gupta, P., Gill, S.S., Jyoti, K.: Rhas: robust hybrid auto-scaling for web applications in cloud computing. Clust. Comput. **24**(2), 717–737 (2021)
21. Guerrero, C., Lera, I., Juiz, C.: Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. J. Grid Comput. **16**, 113–135 (2018)
22. Ali, I.M., Sallam, K.M., Moustafa, N., Chakraborty, R., Ryan, M., Choo, K.-K.R.: An automated task scheduling model using non-dominated sorting genetic algorithm II for fog-cloud systems. IEEE Trans. Cloud Comput. **10**(4), 2294–2308 (2020)

23. Imdoukh, M., Ahmad, I., Alfailakawi, M.G.: Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. **32**(13), 9745–9760 (2020)
24. Prachitmutita, I., Aittinonmongkol, W., Pojjanasuksakul, N., Supattatham, M., Padungweang, P.: Auto-scaling microservices on IaaS under SLA with cost-effective framework. In: 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), pp. 583–588. IEEE (2018)
25. Yu, G., Chen, P., Zheng, Z.: Microscaler: Automatic scaling for microservices with an online learning approach. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 68–75. IEEE (2019)
26. Zhao, H., Lim, H., Hanif, M., Lee, C.: Predictive container auto-scaling for cloud-native applications. In: 2019 International Conference on Information and Communication Technology Convergence (ICTC), pp. 1280–1282 (2019)
27. Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., et al: Autopilot: workload autoscaling at google. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16 (2020)
28. Coulson, N.C., Sotiriadis, S., Bessis, N.: Adaptive microservice scaling for elastic applications. IEEE Internet Things J. **7**(5), 4195–4202 (2020)
29. Marie-Magdelaine, N., Ahmed, T.: Proactive autoscaling for cloud-native applications using machine learning. In: GLOBECOM 2020-2020 IEEE Global Communications Conference, pp. 1–7 (2020)
30. Khaleq, A.A., Ra, I.: Intelligent autoscaling of microservices in the cloud for real-time applications. IEEE Access **9**, 35464–35476 (2021)
31. Horn, A., Fard, H.M., Wolf, F.: Multi-objective hybrid autoscaling of microservices in Kubernetes clusters. In: European Conference on Parallel Processing, pp. 233–250. Springer (2022)
32. Qu, C., Calheiros, R.N., Buyya, R.: Auto-scaling web applications in clouds: a taxonomy and survey. ACM Comput. Surv. (CSUR) **51**(4), 1–33 (2018)
33. Singh, P., Gupta, P., Jyoti, K., Nayyar, A.: Research on auto-scaling of web applications in cloud: survey, trends and future directions. Scalable Comput. Pract. Exp. **20**(2), 399–432 (2019)
34. Bauer, E., Adams, R.: Reliability and Availability of Cloud Computing. Wiley (2012)
35. Bento, A., Soares, J., Ferreira, A., Duraes, J., Ferreira, J., Carreira, R., Araujo, F., Barbosa, R.: Bi-objective optimization of availability and cost for cloud services. In: 2022 IEEE 21st International Symposium on Network Computing and Applications (NCA), vol. 21, pp. 45–53 (2022)
36. Ding, J., Cao, R., Saravanan, I., Morris, N., Stewart, C.: Characterizing service level objectives for cloud services: realities and myths. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 200–206. IEEE (2019)
37. Bello, S.A., Oyedele, L.O., Akinade, O.O., Bilal, M., Delgado, J.M.D., Akanbi, L.A., Ajayi, A.O., Owolabi, H.A.: Cloud computing in construction industry: use cases, benefits and challenges. Autom. Constr. **122** (2021)
38. Kumar, M., Sharma, S.C., Goel, A., Singh, S.P.: A comprehensive survey for scheduling techniques in cloud computing. J. Netw. Comput. Appl. **143**, 1–33 (2019)
39. Amazon Web Services, Inc.: Amazon compute service level agreement. https://aws.amazon.com/compute/sla/ (2022)
40. Richards, F.J.: A flexible growth function for empirical use. J. Exp. Bot. **10**(2), 290–301 (1959)
41. Adan, I., Resing, J.: Queueing theory. Eindhoven University of Technology Eindhoven (2002)
42. Locust.io: Locust: what is locust. https://docs.locust.io/en/stable/what-is-locust.html
43. Blank, J., Deb, K.: Pymoo: multi-objective optimization in python. IEEE Access **8**, 89497–89509 (2020)
44. Konak, A., Coit, D.W., Smith, A.E.: Multi-objective optimization using genetic algorithms: a tutorial. Reliab. Eng. Syst. Saf. **91**(9), 992–1007 (2006)
45. Deb, K., Sindhya, K., Okabe, T.: Self-adaptive simulated binary crossover for real-parameter optimization. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07), pp. 1187–1194. Association for Computing Machinery (2007)
46. Chicco, D., Warrens, M.J., Jurman, G.: The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation. PeerJ Comput. Sci. **7**, 623 (2021)
47. Casalicchio, E., Perciballi, V.: Auto-scaling of containers: the impact of relative and absolute metrics. In: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W), pp. 207–214. IEEE (2017)