Stateful Depletion and Scheduling of Containers on Cloud Nodes for Efficient Resource Usage

Amirali Amiri[1,2,*], Uwe Zdun[1], and Konstantinos Plakidas[3]
[1]University of Vienna, Software Architecture Group, Vienna, Austria
[2]University of Vienna, Doctoral School Computer Science DoCS, Vienna, Austria
[3]fiskaly GmbH, Vienna, Austria
amirali.amiri@univie.ac.at, uwe.zdun@univie.ac.at, konstantinos.plakidas@fiskaly.com
*corresponding author

***Abstract*—Container scheduling is a fundamental part of today's service and cloud-based applications. Schedulers operate at different levels depending on how much control the system developers have. On the one hand, container orchestration managers such as Google Kubernetes manage the scheduling of containers to different nodes. On the other hand, serverless managers, such as Google Autopilot, take care of the underlying infrastructure automatically, and developers do not need to manage the nodes. However, when it comes to container depletion, i.e., removing the assigned cloud resources to an idle container, current scheduling technologies have limitations. In this paper, we propose our approach to managing cloud resource usage when containers are idle efficiently. For this purpose, we deplete idle containers statefully, i.e., propose a novel manager that monitors idle containers, saves their state, and efficiently depletes them. This manager reconstructs a depleted container using the saved state when reconstruction is needed. In our approach, we suggest an Infrastructure as Code component to automate the creation of new nodes if a depleted container cannot be scheduled on the same node, e.g., because of being overloaded. We provide an analytical model for the stateful depletion of containers and their rescheduling and empirically evaluate the accuracy of our model. For this purpose, we ran an experiment on a private cloud infrastructure and Google Cloud Platform. Our model has a low error rate of 4.28% averaged over public and private clouds.**

Keywords*—Container Scheduling; Container Depletion; Cloud Resource Management; Infrastructure as Code amplifiers

I. Introduction

Many different container orchestrating technologies are available. These technologies operate at different levels, e.g., Google Kubernetes[1] and Docker Swarm[2] are container orchestration managers. In contrast, technologies such as Google Autopilot[3] and Microsoft Azure Container Instances[4] are serverless managers that automatically take care of the underlying infrastructure. These technologies are essential for the dynamic behavior of cloud-based applications; however, current technologies take container depletion into account only to a limited extent. As a result, when a container has been scheduled to a node but is idle, the assigned cloud resources, e.g., vCPUs and memory, are not used efficiently. In this case, existing container schedulers would provision more resources than needed resulting in a cost increase of cloud resources.

This problem is highly relevant in cases such as the paper's real-world industrial case study. In it, an extremely high number of containers is needed to process the fiscal transactions of users. For example, there can be (at least) one container per customer that is instantiated when the first fiscal transaction of the customer occurs. These containers have limited cloud resource demand but cannot be shared across multiple customers for privacy and security-related issues. There can be various load profiles for different customers. On the one hand, a type of customer can have an extremely high number of transactions per minute, e.g., a bank-related institute. On the other hand, other customers can have transactions sporadically, e.g., in a small retail shop. In the latter case, the containers are idle, waiting to process the subsequent transactions. We set out to answer the research questions:

RQ1: *How can idle containers be depleted in a stateful manner and rescheduled on a cloud node at run-time?*

RQ2: *How much improvement does this stateful depletion result in with regards to efficient resource management?*

RQ3: *Can we find a decision point, based on the number of containers, to automatically create a new machine and migrate the depleted containers?*

In our prior work [4], [5], we proposed a self-adapting architecture that automatically reconfigures the services of a cloud-based system. In this paper, we propose a new concept that adapts this architecture to the problem of stateful depletion and scheduling of containers on cloud nodes. For this, we present the updated metamodel of our architecture and propose a new self-adapting approach. We focus on dynamically reconfiguring services and modeling container scheduling concerns considering compliance rules. Additionally, we automate the deployment of containers using an Infrastructure as Code (IaC) component, e.g., for tasks such as adding virtual machines to the scheduler node pool at runtime.

We present an analytical model of stateful depletion of idle containers to predict the improvements in terms of the number of processed requests. To evaluate our proposed approach, we designed an experiment according to our studied industrial use case (see Section II-A for details). We performed the

[1] https://kubernetes.io

[2] https://docs.docker.com/engine/swarm/

[3] https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview

[4] https://azure.microsoft.com/en-us/services/container-instances/

experiment on private as well as public cloud infrastructures. Moreover, we empirically found the decision point (based on the number of containers) to automatically change the infrastructure using the IaC component. We calculated the prediction accuracy of our model as 4.28% averaged over private and public clouds. Given that the target prediction accuracy is commonly used as 30.0% in the cloud quality-of-service field [19], this error rate is more than reasonable.

The structure of the article is as follows. In Section II, we give the background of our study. Section III presents the overview of our approach, and Section IV explains our approach details. Our analytical model is parameterized in Section V and evaluated in Section VI. In Section VII, we discuss the prediction error of our model and present the threats to validity of our study. The related work of our paper is presented in Section VIII. Finally, we conclude in Section IX.

II. Background

In this section, the background of our study is presented.

A. Real-World Industrial Case Study

A case study with particular complexity is encountered by fiskaly GmbH, a provider of cloud-based Certified Technical Security Systems (CTSS) used to combat tax fraud. According to German legislation[5], every electronic cash register or Point of Sale (PoS) must be associated with a CTSS instance. This instance is responsible for tracking PoS business cases, recording them as event logs, and digitally signing and storing these logs for a future audit by the tax authorities. Each CTSS comprises two main components: A protocol unit, known as Security Module Application for Electronic Record-keeping Systems (SMAERS), and a cryptography unit, known as the cryptographic service provider. While multiple SMAERS instances can share the latter, each SMAERS is required to be assigned to a customer organization or, in practice, to a specific PoS or set of PoS of such an organization. In other words, the process data of each customer entity must be kept separate from all other customers.

As a result, at least one but potentially up to thousands of SMAERS instances, each running on its own container, must be managed for each customer. Consequently, hundreds of thousands of containers for the German market alone exist. These containers are clustered together in groups of more than 100 containers on virtual machines in the cloud. Depending on the customer's identity and the location of the associated PoS, a SMAERS instance may be highly active, processing thousands of transactions each day or only sporadically active, with a handful of transactions per week. Larger organizations tend to batch-create SMAERS instances, meaning that some clusters may feature many high-load instances belonging to the same customer, while others have more mixed demography. Fiskaly needs to regulate resource consumption to ensure very high availability of the CTSS components per its service level agreements and low-latency servicing of customer requests (typically under 250 msec). This regulation is necessary because cash register transactions cannot be delayed for long. Furthermore, all signed logs generated by the CTSS have to be persisted and be constantly available for immediate export for auditing purposes by the tax authorities. This export includes formally decommissioned SMAERS instances, which must be kept active and accessible indefinitely.

B. Existing Solutions

1) Container Schedulers: Different business-grade container schedulers are available, e.g., Google Kubernetes[1] or Docker Swarm[2]. These tools usually work with constraints, with which a system designer controls on which cloud node a container is scheduled and deployed. However, these schedulers are mostly static: A designer must usually define the node pools and the constraints in advance. If a reconfiguration is needed, this information must be updated manually. Moreover, a container orchestration tool would provision more resources than needed when containers are idle, resulting in a cloud cost increase. This increase is because the depletion of containers cannot be done generically and must be tailored for each application separately.

2) Elastic Containers: Many cloud providers offer a standard service to automate this process, i.e., to run containers without considering the underlying cloud nodes. For instance, Google Autopilot[3], Microsoft Azure Container Instances[4] and Amazon Elastic Container Service[6] on Fargate[7], all of which free a designer from the container scheduling and deployment decisions. However, these solutions are not always applicable when a degree of control is needed over the underlying cloud nodes, e.g., following a compliance rule that dictates on which servers customer data should be stored. So the containers might violate deployment regulations in a fiscal application.

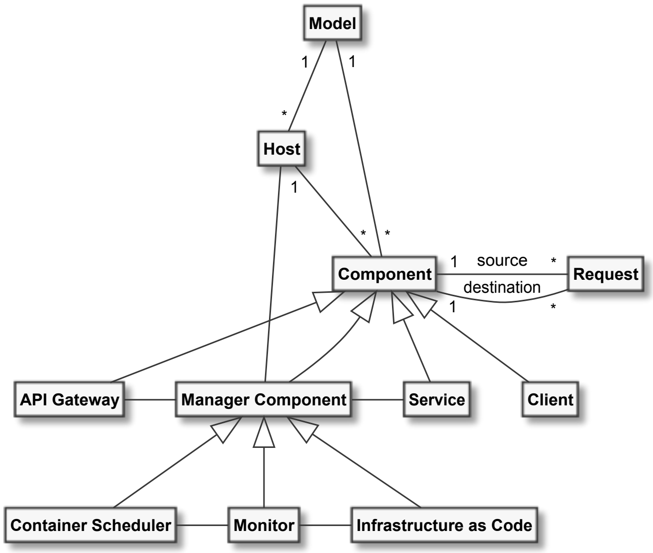


Figure 1: Metamodel

[5] https://kassensichv.com

[6] https://aws.amazon.com/ecs/

[7] https://aws.amazon.com/fargate/

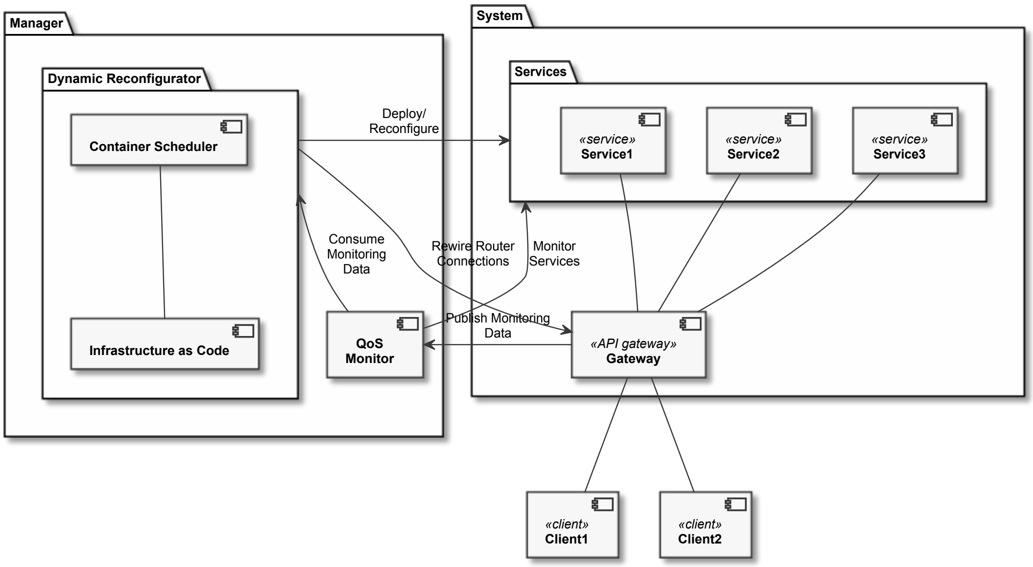


Figure 2: Component Diagram of the Proposed Approach

3) Controllers: Different container scheduling technologies provide controllers that monitor the state of a cloud-based application. For example, using a controller in Kubernetes[8], the number of pods[9] can be changed automatically to achieve the desired state of an application. However, these controllers do not usually focus on a specific container but adjust a cluster of containers to achieve a goal for that cluster. Using a specific controller for each container requires additional effort. Furthermore, extending a controller from a specific technology demands detailed knowledge of the used scheduler and cannot be easily applied to another technology.

III. Approach Overview

To answer **RQ1**, we study the scenario where we can deplete containers in a stateful fashion, i.e., save their state for later reconstruction and free their resources. When a new request for a depleted container is made, we reconstruct the container. Our approach either schedules this container on an existing node or creates a new node. Our generic approach can be used with different container orchestration technologies.

A. Metamodel

In our prior work [4], [5], we introduced a self-adaptive architecture that took QoS metrics, e.g., reliability and performance, into consideration when reconfiguring services of a cloud-based system. Figure 1 presents a simplified version of our metamodel focusing on container scheduling. A *model* has *hosts* and *components*. A specific *component* is an *api gateway*. The *api gateways* forward *requests* received from *clients* to *services*. These are monitored and reconfigured by the *manager components*.

B. Component Diagram

Figure 2 shows the component diagram of our proposed approach that is based on the extensively studied Map, Analyze, Plan, Execute, Knowledge (MAPE-K) loops [6], [7], [15]. This solution automatically updates the scheduler settings based on the monitored data at runtime. The *Gateway* published the monitoring data to the *Quality of Service (QoS) Monitor* that analyzes if a reconfiguration is needed. For example, if degradation of metrics has been observed, the monitor triggers the *Dynamic Reconfigurator* components, i.e., the *Container Scheduler* and the *Infrastructure as Code (IaC)* components.

On the one hand, the proposed solution can be used as a stand-alone tool using a proprietary container scheduler and an IaC manager, e.g., manual scripts working with proprietary cloud technologies. On the other hand, the presented component diagram can be seen as a high-level component diagram used on top of the existing container schedulers, e.g., Google Kubernetes[1] and IaC tools such as Ansible[10].

IV. Approach Details

We analytically model the improvements of the number of processed requests when statefully depleting idle containers.

A. Definition of Container Types

Based on our studied industrial use case (see Section II-A), we define *busy containers* as the ones that are active most of the time processing requests with a high frequency, i.e., thousands of requests per day. On the other hand, some containers receive requests with a lower frequency and are idle between incoming calls. In this paper, we call these sporadically active containers *sporadical containers*.

[8] https://kubernetes.io/docs/concepts/architecture/controller/
[9] https://cloud.google.com/kubernetes-engine/docs/concepts/pod
[10] https://www.ansible.com

B. Reconfiguration Algorithms

Algorithm 1 proposes a simple depletion strategy that is used by the *Manager* component in Figure 2. We monitor sporadical containers for idleness. When depletion is triggered for a container, we replace it with an extra container, i.e., a new container that has yet to be deployed. However, deploying an extra container only happens if there are more than twice as many depleted containers as extra containers in the system. This policy ensures that we do not overload the cloud nodes by deploying extra containers after each depletion.

Algorithm 1: Reconfiguration Algorithm to Statefully Deplete and Deploy Extra Containers

```
input $n_{depl}$                                      // number of depleted containers
input $n_{extra}$                                     // number of extra containers

foreach (c : containers)                              // parallel for loop — all containers
begin
    while(true)                                       // waiting for idle signal
    begin
        if (idle)
            deplete()                                 // stateful depletion
            if ($n_{depl} >= 2 \cdot n_{extra} + 1$)
                replace()                             // replace with extra containers
                $n_{extra} \leftarrow n_{extra} + 1$
        $n_{depl} \leftarrow n_{depl} + 1$
    end
end

return $n_{extra}$                                    // used in our analytical model
```

Algorithm 2 shows a simple scheduling algorithm when a request for a depleted container arrives. In this case, the container is scheduled on an existing node. Alternatively, if the current nodes are overloaded, the Infrastructure as Code component creates a new node, on which the container is scheduled. Section VII presents an analysis to recognize and predict an overloaded node. Note that this algorithm affects the number of depleted containers, i.e., n_{depl}. This is used in Algorithm 1 to calculate the number of extra containers, i.e., n_{extr}, used in our analytical model.

Algorithm 2: Reconfiguration Algorithm to Schedule a Depleted Container

```
input $n_{depl}$                                      // number of depleted containers

foreach (c : depleted_containers) // parallel for loop — depleted containers
begin
    while(true)                                       // waiting for requests
    begin
        if (request)
            if (nodes are not overloaded)
                deploy_on_current_nodes()
            else
                deploy_on_another_node()              // can create a new node using IaC
        $n_{depl} \leftarrow n_{depl} - 1$
    end
end

return $n_{depl}$
```

C. Analytical Model

Here, we present the analytical model of our approach.

1) Definition of Depletion Events: To calculate the achieved improvements (in terms of the number of processed requests when statefully depleting idle containers), we need some base measurements to compare. That is the number of requests that are processed without any depletion of containers. For this purpose, we use a combination of busy and sporadical services. We calculate the number of requests processed during a fixed observed time. We analytically model this measurement of requests without depletion in the following sections. Moreover, we model the depletion event of sporadical containers (in terms of the number of processed requests) when they are idle and busy containers are replaced.

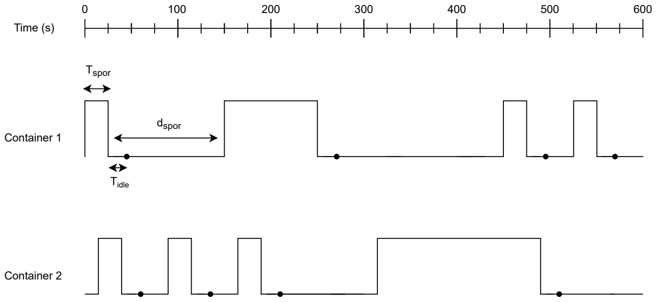


Figure 3: The Sporadical Load Profiles of Two Containers (dots represent depletion)

2) Number of Requests without Depletion Events: Let T be the observed system time in seconds, during which busy and sporadical containers are processing incoming requests. Based on our industrial use case (see Section II-A), we define a busy load profile as constantly feeding a busy container with a call frequency, i.e., f_{busy} without stopping during T. Let n_{busy} be the number of busy containers in a system. A sporadical load profile is defined as a call frequency f_{spor} for a short time T_{spor} followed by a delay d_{spor} of no incoming requests. As shown in Figure 3, a sporadical load is repeated with different values of time and delay. Let n_{spor} be the number of sporadical containers in an application and $\sum_{act}$ be the summation of the periods a container is active. The number of processed requests without depletion, i.e., R, can be calculated as the total number of requests processed by the busy containers plus the requests processed by the sporadical containers when active:

$$R = \sum_{c=1}^{n_{busy}} f_{busy}^{c} \cdot T + \sum_{c=1}^{n_{spor}} \sum_{act} f_{spor}^{c} \cdot T_{spor}^{c} \tag{1}$$

3) Number of Requests with Depletion Events: We deplete sporadical containers when idle (not processing requests) for a period of time T_{idle}. In Figure 3, containers are depleted (represented by dots) after T_{idle} of inactivity. We deploy extra containers instead according to Algorithm 1 (see Section IV-B for explanation). Let n_{extr} be the number of extra containers, f_{extr}^{c} the call frequency of extra container c, and T_{extr}^{c} the time period an extra container c is active. The number of processed requests with depletion, i.e., R_{depl}, is:

$$R_{depl} = R + \sum_{c=1}^{n_{extr}} \sum_{act} f_{extr}^{c} \cdot T_{extr}^{c} \tag{2}$$

We calculate the percentage improvement of the processed requests as follows using Equation (2):

$$\Delta R = \frac{100\%}{R} \cdot (R_{depl} - R) \tag{3}$$

$$\Delta R = \frac{100\%}{R} \cdot \sum_{c=1}^{n_{extr}} \sum_{act} f_{extr}^{c} \cdot T_{extr}^{c} \tag{4}$$

V. Parameterization of Model Elements

Our analytical model can be applied to different scenarios of multiple load profiles with various numbers of containers. Architects must parameterize our model to their specific use case at hand. In this section, we introduce an illustrative sample case and explain how this parameterization of our model elements can be performed.

A. Illustrative Sample Case

We consider a scenario where the load profiles of multiple sporadical containers are so that these containers can be swapped without losing any requests. In this case, when a container is idle, another container receives incoming calls, as shown in Figure 4 for an example. The main benefit of this case is that the resources of the node, e.g., vCPUs, are not reserved for an idle container and can be used efficiently for a busy container resulting in faster response time. This efficient usage can also reduce costs depending on the cloud cost profile a user opts for. If customers are billed per resource usage (e.g., see Google Autopilot pricing[11]), the cost reduction can be significant.

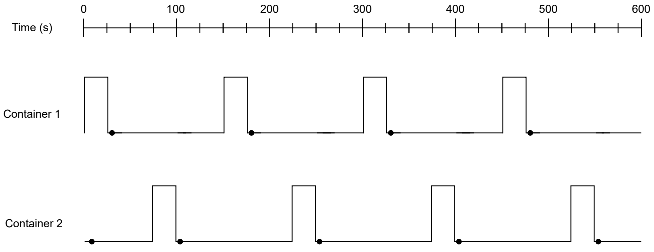


Figure 4: The Sporadical Load Profiles of Two Containers in the Illustrative Sample Case (dots represent depletion)

B. Model Parameterization for the Sample Case

In this case, the sporadical load is repeated homogeneously. Therefore, we can calculate the total requests processed by busy containers and requests processed by the sporadical containers over their active periods as a fraction of T. We rewrite Equation (1):

$$R = n_{busy} \cdot f_{busy} \cdot T + n_{spor} \cdot f_{spor} \cdot T \cdot \frac{T_{spor}}{T_{spor} + d_{spor}} \tag{5}$$

Let n_{extr}, f_{extr} and T_{extr} be the number of extra containers, frequency and active time of extra containers, respectively. As a result of homogeneous load, we can rewrite Equation (2) for the illustrative sample case as:

$$R_{depl} = R + n_{extr} \cdot f_{extr} \cdot T_{extr} \tag{6}$$

Following Algorithm 1, the n_{extr}, in this case, is half the number of sporadical containers. The reasoning behind this is that two containers are swapped repeatedly, as it can be seen in Figure 4, and one container is replaced in this case:

$$n_{extr} = \frac{n_{spor}}{2} \tag{7}$$

We mentioned that when we deplete idle containers, we replace them with busy containers as extra, i.e., we feed them with a busy load profile to maximize the number of processed requests, therefore:

$$f_{extr} = f_{busy} \tag{8}$$

In this illustrative sample case, the extra containers are deployed after the first depletion happens, i.e., after T_{idle}, and are active for the rest of the experiment:

$$T_{extr} = T - T_{idle} \tag{9}$$

Finally, the percentage improvement of the processed requests ΔR presented in Equation (4) for this scenario is:

$$\Delta R = \frac{(n_{spor}/2) \cdot f_{busy} \cdot (T - T_{idle})}{R} \cdot 100\% \tag{10}$$

VI. Evaluation

To answer **RQ2** and to evaluate our approach, we designed an experiment on cloud settings that are representative of our industrial case study (see Section II-A).

A. Experiment Planning

1) Goal: We aim to empirically evaluate the improvement in efficient resource usage when idle containers are statefully depleted and reconstructed at a later time.

2) Method: We containerize multiples of the number of services representative of our industrial case study and deploy these containers on a virtual node in public and private cloud infrastructures (see below for details). Then, requests with different levels of frequencies are sent to these containerized services. We deplete containers that are idle for a period of time and measure the difference in the number of processed requests with and without depletion. We follow the illustrative sample case presented in Section V-A for our experiment planning.

3) Experiment Cases: As in our prior work [3], we take the experiment duration, i.e., the observed time, of $T = 600$ seconds. We define two load profiles, i.e., *busy* and *sporadical profiles* based on our studied industrial case study (see Section II-A). A busy load profile is active during the entire experiment run. In our industrial case, busy containers can process thousands of requests daily. Therefore, we define a representative call frequency of busy containers:

$$f_{busy} = 5\ r/s \tag{11}$$

[11] https://cloud.google.com/kubernetes-engine/pricing

Note that these frequencies result in 432000 requests per day.

Presenting our case study, we mentioned that a sporadical load could be as low as a handful of weekly requests. As this is not predictive and might result in no requests per $T = 600$ seconds of experiment time, we follow the sporadical load profiles presented in Figure 4. To observe some requests during our experiment time (and for the cases to be comparable), we give the same call frequency for sporadical and extra containers as for the busy containers (i.e., $5\ r/s$):

$$f_{spor} = f_{extr} = 5\ r/s \tag{12}$$

However, as mentioned before, the sporadical load is active for a time period T_{spor} and inactive for a short delay d_{spor}. To more closely resemble our industrial case study and cover multiple sporadical load profiles, we use two levels for T_{spor} and d_{spor}:

$$(T_{spor}, d_{spor}) \in \{\ (25, 125)\,, (50, 150)\ \} \tag{13}$$

Containers are depleted after a T_{idle} of inactivity. To study the effects of this model element, we also use three levels:

$$T_{idle} \in \{5, 25, 45\}\ \ s \tag{14}$$

These levels are chosen so we have a very short delay ($5\,s$) as well as the longest delay that still results in homogeneous load ($45\,s$) as used in our illustrative sample case (see Section V-A). Moreover, we study a middle case ($25\,s$) to be complete.

Following our industrial use case, we use different numbers of busy and sporadical containers. Remember that the number of extra containers in the illustrative sample case is half the number of sporadical containers according to Equation (7) (see section Section V-B for explanation):

$$\begin{aligned}(n_{busy}, n_{spor}, n_{extr}) \in \{\ & (50, 50, 25), \\ & (80, 20, 10), \\ & (100, 20, 10), \\ & (100, 50, 25)\ \} \end{aligned} \tag{15}$$

We have 96 evaluation cases in total. These include 32 experiment cases (with and without depletion) validated on three cloud infrastructures. Each evaluation case ran for $T = 600$ seconds. Moreover, each case was repeated five times and averaged over to mitigate the influence of other factors on the measurements, such as other workloads in public clouds (see threats to validity Section VII-C). Overall, our experiment had a run-time of 80 hours (excluding setup and processing time). Details of each evaluation case are provided in the online artifact of this study[12].

4) Technical Details: We used private and public cloud infrastructures to validate the accuracy of our model.

Private Cloud Infrastructure We used a physical server having two identical CPUs. The server hosts an Intel® Xeon® E5-2680 v4 @ 2.40 GHz[13] CPU. The processor has 14 cores and two physical threads per core (56 in total). We installed a virtual node on the server using VMware ESXi version 6.7.0 u2 hypervisor. This virtual node has eight vCPU cores, 60 GB system memory, and runs Ubuntu Server 18.04.01 LTS[14]. Docker technology[15] is used to containerize services implemented in Node.js[16]. Each service listens for a request and performs a dummy operation, i.e., a delay of 1000 loops.

Validation Experiment on Public and Private Clouds We used our private cloud to have control over the infrastructure. On a public cloud, other factors, such as the parallel workload of other applications, can influence the results. To show that our approach can be used on other infrastructures as well, we empirically validate the analysis of our proposed model on our private cloud infrastructure and Google Cloud Platform (GCP)[17]. On GCP, we use two machine types, i.e., general-purpose E2 machine instance[18] with two vCPUs and 8 GB of memory, and compute-optimized C2 machine instance[19] with four vCPUs and 16 GB of memory. We duplicated our private cloud infrastructure on these machines and repeated the whole experiment on them. Overall three repetition that we call: **Private**, **GCP2** (two vCPUs) and **GCP4** (four vCPUs).

Load Generation For load generation, we utilized a MacBook Pro with an Apple M1 Pro chip and 16 GB of system memory that runs macOS Monterey version 12.2.1. It generates load using Apache JMeter[20] that sends hypertext transfer protocol version 1.1[21] requests to the virtual nodes.

5) Methodological Principles of Reproducibility: We followed the eight principles of reproducibility [22]:

- *Repeated experiments:* Each experiment case is repeated precisely on the same infrastructure with precise values.
- *Workload and configuration coverage:* We covered multiple experiment cases according to a real-world industrial use case (see Section VI-A3).
- *Experimental setup description:* Our experimental setup is reported in Section VI-A.
- *Open access artifact:* The code, data, and evaluation log of this study is published as an open access data set to support replicability[12].
- *Result description of measured performance:* We described our results in Section VI-B and reported the error measurements in Section VII-B.
- *Statistical evaluation:* See this section.
- *Measurement units:* We reported all units.
- *Cost:* We used the free trial offered by GCP[22].

B. Experiment Results

Table I presents our experiment's model predictions and empirical measurements, and Figure 5 visualizes the data. We

[12] https://zenodo.org/record/7334238 DOI: 10.5281/zenodo.7334238

[13] https://www.intel.com/content/www/us/en/homepage.html

[14] https://www.ubuntu.com

[15] https://www.docker.com

[16] https://nodejs.org/en/

[17] https://cloud.google.com

[18] https://cloud.google.com/compute/docs/general-purpose-machines

[19] https://cloud.google.com/compute/docs/compute-optimized-machines

[20] https://jmeter.apache.org

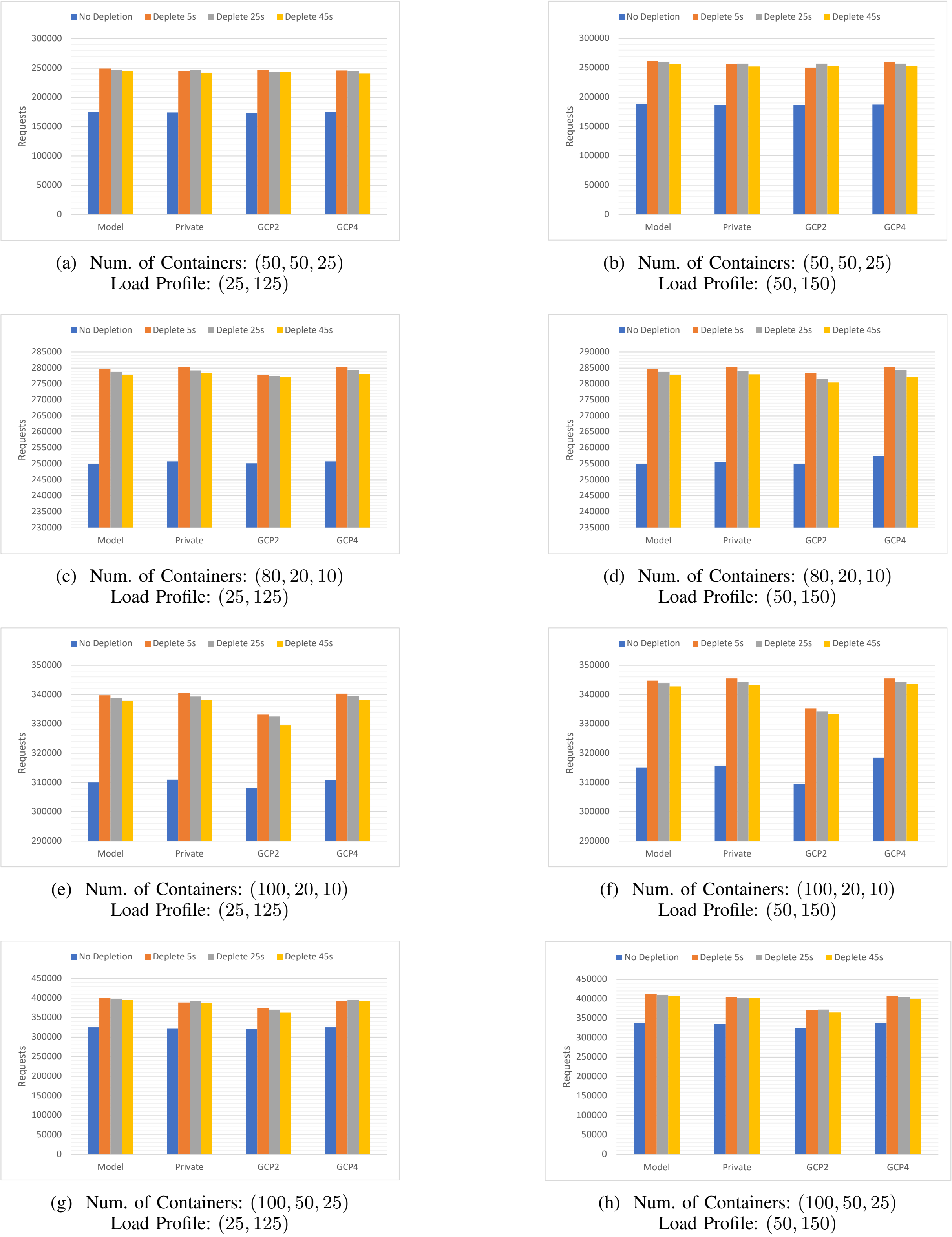
[21] https://tools.ietf.org/html/rfc7230

[22] https://cloud.google.com/free/docs/free-cloud-features

TABLE I: Model Predictions of Experiment Cases and Empirical Results

$T_{idle}(s)$	**Num. of Containers** $(n_{busy}, n_{spor}, n_{extr})$	**Load Profile** (T_{spor}, d_{spor})		R	R_{depl}	$\Delta R(\%)$		R	R_{depl}	$\Delta R(\%)$
				Model				**Private**		
5	$(50, 50, 25)$	$(25, 125)$		175000.00	249375.00	42.50		174180.00	245020.00	40.67
		$(50, 150)$		187500.00	261875.00	39.87		186810.00	256480.00	37.29
	$(80, 20, 10)$	$(25, 125)$		250000.00	279750.00	11.90		250756.00	280348.00	11.80
		$(50, 150)$		255000.00	284750.00	11.67		255580.00	285244.00	11.61
	$(100, 20, 10)$	$(25, 125)$		310000.00	339750.00	9.60		310988.00	340580.00	9.51
		$(50, 150)$		315000.00	344750.00	9.44		315780.00	345462.00	9.40
	$(100, 50, 25)$	$(25, 125)$		325000.00	399375.00	22.88		322420.00	388299.00	20.43
		$(50, 150)$		337500.00	411875.00	22.04		335210.00	404290.00	20.61
25	$(50, 50, 25)$	$(25, 125)$		175000.00	246875.00	41.07		174180.00	246335.00	41.42
		$(50, 150)$		187500.00	259375.00	38.33		186810.00	257460.00	37.82
	$(80, 20, 10)$	$(25, 125)$		250000.00	278750.00	11.50		250756.00	279234.00	11.36
		$(50, 150)$		255000.00	283750.00	11.27		255580.00	284204.00	11.20
	$(100, 20, 10)$	$(25, 125)$		310000.00	338750.00	9.27		310988.00	339314.00	9.11
		$(50, 150)$		315000.00	343750.00	9.13		315780.00	344224.00	9.01
	$(100, 50, 25)$	$(25, 125)$		325000.00	396875.00	22.12		322420.00	392450.00	21.72
		$(50, 150)$		337500.00	409375.00	21.30		335210.00	402045.00	19.94
45	$(50, 50, 25)$	$(25, 125)$		175000.00	244375.00	39.64		174180.00	242355.00	39.14
		$(50, 150)$		187500.00	256875.00	37.00		186810.00	252295.00	35.05
	$(80, 20, 10)$	$(25, 125)$		250000.00	277750.00	11.10		250756.00	278364.00	11.01
		$(50, 150)$		255000.00	282750.00	10.88		255580.00	283048.00	10.75
	$(100, 20, 10)$	$(25, 125)$		310000.00	337750.00	8.95		310988.00	338080.00	8.71
		$(50, 150)$		315000.00	342750.00	8.81		315780.00	343366.00	8.74
	$(100, 50, 25)$	$(25, 125)$		325000.00	394375.00	21.35		322420.00	387565.00	20.20
		$(50, 150)$		337500.00	406875.00	20.55		335210.00	401460.00	19.76

				GCP2				**GCP4**		
5	$(50, 50, 25)$	$(25, 125)$		173330.00	246935.00	42.47		174650.00	245995.00	40.85
		$(50, 150)$		187060.00	249585.00	33.43		187260.00	259580.00	38.62
	$(80, 20, 10)$	$(25, 125)$		250200.00	277796.00	11.03		250764.00	280276.00	11.77
		$(50, 150)$		254924.00	283392.00	11.17		257532.00	285234.00	10.76
	$(100, 20, 10)$	$(25, 125)$		308048.00	333166.00	8.15		310904.00	340314.00	9.46
		$(50, 150)$		309600.00	335316.00	8.31		318492.00	345512.00	8.48
	$(100, 50, 25)$	$(25, 125)$		320720.00	374910.00	16.90		324540.00	393010.00	22.00
		$(50, 150)$		325050.00	370295.00	13.92		336530.00	407845.00	21.19
25	$(50, 50, 25)$	$(25, 125)$		173330.00	243690.00	40.59		174650.00	245295.00	40.44
		$(50, 150)$		187060.00	257365.00	37.58		187260.00	257140.00	37.32
	$(80, 20, 10)$	$(25, 125)$		250200.00	277412.00	10.88		250764.00	279374.00	11.41
		$(50, 150)$		254924.00	281530.00	10.44		257532.00	284286.00	10.39
	$(100, 20, 10)$	$(25, 125)$		308048.00	332460.00	7.92		310904.00	339402.00	9.17
		$(50, 150)$		309600.00	334226.00	7.95		318492.00	344340.00	8.12
	$(100, 50, 25)$	$(25, 125)$		320720.00	369435.00	15.19		324540.00	395080.00	21.74
		$(50, 150)$		325050.00	372150.00	14.49		336530.00	404780.00	20.28
45	$(50, 50, 25)$	$(25, 125)$		173330.00	243240.00	40.33		174650.00	240820.00	37.89
		$(50, 150)$		187060.00	253500.00	35.52		187260.00	253210.00	35.22
	$(80, 20, 10)$	$(25, 125)$		250200.00	277128.00	10.76		250764.00	278156.00	10.92
		$(50, 150)$		254924.00	280468.00	10.02		257532.00	282176.00	9.57
	$(100, 20, 10)$	$(25, 125)$		308048.00	329408.00	6.93		310904.00	338094.00	8.75
		$(50, 150)$		309600.00	333302.00	7.66		318492.00	343506.00	7.85
	$(100, 50, 25)$	$(25, 125)$		320720.00	362720.00	13.10		324540.00	392650.00	20.99
		$(50, 150)$		325050.00	364810.00	12.23		336530.00	398930.00	18.54



(a) Num. of Containers: $(50, 50, 25)$
Load Profile: $(25, 125)$

(b) Num. of Containers: $(50, 50, 25)$
Load Profile: $(50, 150)$

(c) Num. of Containers: $(80, 20, 10)$
Load Profile: $(25, 125)$

(d) Num. of Containers: $(80, 20, 10)$
Load Profile: $(50, 150)$

(e) Num. of Containers: $(100, 20, 10)$
Load Profile: $(25, 125)$

(f) Num. of Containers: $(100, 20, 10)$
Load Profile: $(50, 150)$

(g) Num. of Containers: $(100, 50, 25)$
Load Profile: $(25, 125)$

(h) Num. of Containers: $(100, 50, 25)$
Load Profile: $(50, 150)$

Figure 5: Plots of All Experiment Cases
without Depletion, and Depletion with T_{idle} seconds

analyze the results separately for private and public cloud infrastructures.

1) Private Cloud Infrastructure: The experiment results are very close to our analytical model predictions in all cases of our private cloud infrastructure. Our model predicts, and our experiment confirms that having the same number of containers and feeding them with the same load profile, a shorter T_{idle} (the time period a container is idle before depletion) results in a higher R_{depl} (number of processed requests with depletion). This improvement is because we quickly identify idle containers and replace them with busy ones. Therefore, more requests are processed during the same period of time, i.e., resources are used more efficiently.

Moreover, when sporadical load profiles have a longer T_{spor} (the time period a sporadical container is active), we have a higher R_{depl} in all cases when we keep other model elements constant. This improvement is because when sporadical containers are swapped, they stay active for longer, processing more requests. Another interesting observation is that the ratio of busy to sporadical containers directly impacts ΔR, i.e., the percentage improvement of the processed requests. In all experiment cases with $(n_{busy}, n_{spor}, n_{extr})$ of $(80, 20, 10)$ and $(100, 20, 10)$, we have around 10% increase in the number of the processed requests. As we increase the number of sporadical containers (and consequently the number of extra containers) in the experiment case of $(100, 50, 25)$, the ΔR also rises to a value close to 20%. Having a one-to-one of busy and sporadical containers results in the highest ΔR as predicted by our models and confirmed by our experiment of $(50, 50, 25)$ containers.

2) Public Cloud Infrastructure: The experiment results on GCP follow the data trend of the private cloud. However, as seen in Figure 5, as the number of containers goes higher, GCP2 has lower processed requests compared to our model predictions and other experiment infrastructures. This deterioration is because GCP2 has an E2 machine instance with two vCPUs. This machine can handle a lower number of containers, i.e., $(n_{busy}, n_{spor}, n_{extr})$ of $(50, 50, 25)$ and $(80, 20, 10)$, closely to the other experiment infrastructures. Nonetheless, when a high number of containers are deployed on this machine, i.e., $(100, 20, 10)$ and $(100, 50, 25)$ containers, the E2 machine is saturated and results in fewer processed requests. As seen in Table I and Figure 5, a more powerful C2 machine in GCP4 that has four vCPUs can handle all our experiment cases, and the experiment results are close to our model predictions.

VII. Discussion

This section discusses our findings to answer **RQ3**. Moreover, we calculate the prediction accuracy of our model. Finally, we present the threats to the validity of our study.

A. Container Migration

As we studied in Section VI-B, the GCP2 infrastructure becomes saturated when the number of containers increases. As shown in Figure 2, our approach includes an Infrastructure as Code (IaC) component. IaC can automatically start a new machine to migrate the depleted containers, i.e., schedule the containers on a newly-created machine. In our experiment, the GCP2 infrastructure performed close to our model predictions with $(n_{busy}, n_{spor}, n_{extr})$ of $(50, 50, 25)$ and $(80, 20, 10)$. That is when there are up to 100 deployed containers in a system, i.e., busy and sporadical containers.

$$n_{busy} + n_{spor} \leq 100 \tag{16}$$

However, with more than 100 deployed containers, i.e., $(n_{busy}, n_{spor}, n_{extr})$ of $(100, 20, 10)$ and $(100, 50, 25)$, the GCP4 infrastructure gave values close to our model predictions. Therefore, we can empirically conclude that for our experiment cases, the decision point to start a new machine is when we have 100 deployed containers. We call this **GCP Mixed**. The IaC component uses the following formula to change the infrastructure and schedules a container using Algorithm 2.

$$\text{GCP Mixed} = \begin{cases} \text{GCP2} & \text{if } (n_{busy} + n_{spor} \leq 100) \\ \text{GCP4} & \text{otherwise} \end{cases}$$

Table II presents the model predictions of our experiment cases and the empirical measurements on this infrastructure.

B. Evaluation of the Prediction Error

We measure the accuracy of our model predictions compared to the empirical results of our private infrastructure (see Table I) and the GCP Mixed infrastructure (see Table II). We calculate the prediction error by calculating four error measurements commonly used in the cloud research [27]:

- Mean Absolute Percentage Error (MAPE)
- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)

We calculate the error measurements in terms of ΔR and define the common elements used in the error measurements. Let ΔR^c_{model} and $\Delta R^c_{empirical}$ be the result of the model, and the measured empirical data for an experiment case c, and n_c be the number of measured empirical cases. As shown in Table I and Table II, we have $n_c = 24$ ΔR values per each experiment infrastructure. The error measurements are calculated using the following formulae:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c=1}^{n_{case}} \left| \frac{\Delta R^c_{model} - \Delta R^c_{empirical}}{\Delta R^c_{empirical}} \right| \tag{17}$$

$$MAE = \frac{1}{n_c} \cdot \sum_{c=1}^{n_{case}} \left| \Delta R^c_{model} - \Delta R^c_{empirical} \right| \tag{18}$$

$$MSE = \frac{1}{n_c} \cdot \sum_{c=1}^{n_{case}} \left(\Delta R^c_{model} - \Delta R^c_{empirical} \right)^2 \tag{19}$$

$$RMSE = \sqrt{MSE} \tag{20}$$

Table III Presents the prediction error of the proposed model compared to the measured data on different experiment

TABLE II: Model Predictions and Empirical Results on the GCP Mixed Infrastructure

$T_{idle}(s)$	Num. of Containers $(n_{busy}, n_{spor}, n_{extr})$	Load Profile (T_{spor}, d_{spor})	R	R_{depl}	$\Delta R(\%)$	R	R_{depl}	$\Delta R(\%)$
				Model			**GCP Mixed**	
5	$(50, 50, 25)$	$(25, 125)$	175000.00	249375.00	42.50	173330.00	246935.00	42.47
		$(50, 150)$	187500.00	261875.00	39.87	187060.00	249585.00	33.43
	$(80, 20, 10)$	$(25, 125)$	250000.00	279750.00	11.90	250200.00	277796.00	11.03
		$(50, 150)$	255000.00	284750.00	11.67	254924.00	283392.00	11.17
	$(100, 20, 10)$	$(25, 125)$	310000.00	339750.00	9.60	310904.00	340314.00	9.46
		$(50, 150)$	315000.00	344750.00	9.44	318492.00	345512.00	8.48
	$(100, 50, 25)$	$(25, 125)$	325000.00	399375.00	22.88	324540.00	393010.00	22.00
		$(50, 150)$	337500.00	411875.00	22.04	336530.00	407845.00	21.19
25	$(50, 50, 25)$	$(25, 125)$	175000.00	246875.00	41.07	173330.00	243690.00	40.59
		$(50, 150)$	187500.00	259375.00	38.33	187060.00	257365.00	37.58
	$(80, 20, 10)$	$(25, 125)$	250000.00	278750.00	11.50	250200.00	277412.00	10.88
		$(50, 150)$	255000.00	283750.00	11.27	254924.00	281530.00	10.44
	$(100, 20, 10)$	$(25, 125)$	310000.00	338750.00	9.27	310904.00	339402.00	9.17
		$(50, 150)$	315000.00	343750.00	9.13	318492.00	344340.00	8.12
	$(100, 50, 25)$	$(25, 125)$	325000.00	396875.00	22.12	324540.00	395080.00	21.74
		$(50, 150)$	337500.00	409375.00	21.30	336530.00	404780.00	20.28
45	$(50, 50, 25)$	$(25, 125)$	175000.00	244375.00	39.64	173330.00	243240.00	40.33
		$(50, 150)$	187500.00	256875.00	37.00	187060.00	253500.00	35.52
	$(80, 20, 10)$	$(25, 125)$	250000.00	277750.00	11.10	250200.00	277128.00	10.76
		$(50, 150)$	255000.00	282750.00	10.88	254924.00	280468.00	10.02
	$(100, 20, 10)$	$(25, 125)$	310000.00	337750.00	8.95	310904.00	338094.00	8.75
		$(50, 150)$	315000.00	342750.00	8.81	318492.00	343506.00	7.85
	$(100, 50, 25)$	$(25, 125)$	325000.00	394375.00	21.35	324540.00	392650.00	20.99
		$(50, 150)$	337500.00	406875.00	20.55	336530.00	398930.00	18.54

infrastructures. Our model has a MAPE prediction error of 4.28% averaged over Private and GCP infrastructures. Given the 30.0% target prediction accuracy commonly used in the cloud quality-of-service field [19], The prediction error of our approach is more than reasonable. Other low error measurements also confirm the high accuracy of our model.

C. Threats to Validity

As in all empirical research, there are several threats to the validity as well as limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [28].

1) Construct validity: In our study, we modeled the depletion of containers based on the number of requests they are processing at a given time period. This approach is a common criterion in the cloud quality-of-service research (see Section VIII) and in current container scheduling technologies such as Google Kubernetes[1] to define controllers based on the incoming load (see Section II-B). However, a threat remains that other measures, such as CPU usage percentage, might result in more accurate modeling of container depletion. More research with several real-world systems would be needed to cover other measurements and exclude this threat.

TABLE III: Prediction Error of the Proposed Model

Measurement	**Private**	**GCP Mixed**	**Overall**
$MAPE(\%)$	2.96	5.60	4.28
MAE	0.69	0.95	0.82
MSE	1.11	2.40	1.33
$RSME$	1.05	1.55	1.3

2) Internal validity: This concerns factors that affect the independent variables concerning causality. We performed an experiment based on our studied industrial use case to evaluate our proposed model. However, we did so in limited experiment time and had control over the workload on cloud infrastructures. We avoided factors such as other loads on the machines where the experiment ran. To mitigate this threat, we repeated each experiment case five times and averaged the empirical measurements, and much of the related literature takes a similar approach (see Section VIII). However, more research with multiple real-world workloads would be needed to confirm that no other factors influence the measurements.

3) External validity: This concerns threats that limit the ability to generalize the results beyond the experiment. We designed our approach with generality in mind and explained in detail how architects could tailor it to their needs (see Section V). Although we evaluated our approach by designing a representative experiment and measuring empirical data, the threat remains that evaluating based on another infrastructure may lead to different results. To mitigate this thread, we validated our measurements on Google Cloud Platform infrastructure and showed that our results are applicable (see Section VI). Also, we consider multiple load profiles, including a sporadical load profile (see Section IV-C and Figure 3). However, the load was constant regarding the frequency of incoming calls during active periods. We plan to cover a bursty load of different frequencies for future work. A related threat is that we implemented all our services with Node.js[16] and did not use an off-the-shelf implementation, e.g., Envoy[23]. We did so to have a comparable infrastructure and to avoid technological impacts on our results.

4) Conclusion Validity: This concerns factors that affect the ability to draw conclusions about the relations between treatments and study outcomes. As the statistical method to compare our model's predictions to the empirical data, we used the MAPE metric as it is widely used and offers good interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, i.e., MAE, MSE, and RMSE, which confirmed our results reported in Section VII-B.

VIII. Related Work

The proposed the approach in this paper is related to self-adaptive systems that typically use MAPE-K loops [7], and similar approaches to realize adaptations. Moreover, research on efficient resource provisioning, e.g., [11], [17], and cloud elasticity, e.g., [13], [14], are related to our work. Our study extends these approaches by analytically modeling the depletion of idle containers as a reconfiguration measure. Similarly, multidimensional auto-scalers have been studied in the literature for resource provisioning. AutoMAP [8] uses response time triggers to provision resources. AutoMAP finds optimal resources using Virtual Machine (VM) image sizes to support cost efficiency. Nguyen et al. [21] use a forecasting model to predict CPU demand and uses these predictions to start new machines before load peak to increase performance. CloudScale [25] supports scaling of CPU and memory resources when local scaling is possible. Otherwise, it migrates VMs to prevent overloaded hosts. Our work differs from these studies because they consider auto-scaling at the VM level and configure the resources. We proposed an approach that works at the container level by depleting and rescheduling containers on cloud nodes.

Moreover, our study provides a novel approach for container depletion and scheduling on cloud nodes, especially considering potentially high numbers of sporadically active containers, as in the considered industry case (see Section II-A). While another existing approach has yet to study this particular research problem, there is a rich literature on container scheduling in a more general sense. For example, Stratus [10] is a dynamically allocating cluster scheduler orchestrating batch job execution on virtual clusters of virtual machine instances on public IaaS platforms. KCSS [20] offers a novel container scheduling strategy for Kubernetes. Kaewkasi and Chuenmuneewong [16] use the Ant Colony Optimization methods to implement a new scheduler for Docker, whereas Liu et al. [18] provide a new container scheduling approach based on multi-objective optimization. Cérin et al. [12] introduce a new Docker Swarm scheduler that uses service level agreement information to provision a container that must execute the service based on a dynamic computation of available resources. Sureshkumar and Rajesh [26], in contrast to those other approaches, use load scheduling to optimize Container usage. Our study differs from these works because it is not specific to container scheduling technology. Our approach tackles containers' stateful depletion and rescheduling to cloud nodes from a higher level of abstraction that can be used with different container orchestration technologies.

Finally, our approach is relevant to architecture-based optimization analysis [2]. This analysis builds on top of prediction approaches and uses architectural tactics to search for optimal architectural candidates. Example optimization analysis approaches are ArcheOpterix [1], PerOpteryx [9], and SQuAT [23]. Sharma and Trivedi [24] present an architecture-based unified hierarchical model for software reliability, performance, security, and cache behavior prediction. In our prior work [4], [5], we used multi-criteria optimization analysis to find Pareto optimal solutions [2]. Like our study, those works focus on supporting architectural design or decision-making to facilitate this process. In contrast to our work, they do not focus on the stateful depletion of containers and rescheduling them for efficient resource usage.

IX. Conclusion and Future Work

In this paper, we studied container scheduling with depletion, i.e., freeing cloud resources when containers are idle. We studied the industrial case of fiskaly GmbH, in which many containers that need limited resources run in parallel. This process results in reserving many cloud resources and increasing costs if they are idle. We set out to answer how idle containers can be depleted statefully and rescheduled on a cloud node at run-time when needed (**RQ1**), how much improvement this stateful depletion results in concerning resource management (**RQ2**), and whether we can find a decision point, based on the number of containers, to automatically create a new machine and migrate the depleted containers (**RQ3**).

For **RQ1**, we proposed a novel self-adaptive approach based on MAPE-K [6], [7], [15] loops to monitor containers for idleness and deplete them if necessary. For this purpose, we proposed an analytical model. We explained the details of our approach that can also work with off-the-shelf orchestration solutions adding depletion management capabilities. Moreover, we discussed how architects could parameterize our analytical model to different scenarios by following an illustrative sample case. For **RQ2**, we designed and performed an experiment on a private cloud infrastructure, as well as on Google Cloud Platform (GCP)[17]. Based on the details of our studied industrial use case (see Section II-A), we defined multiple experiment cases and compared our empirical results to our model predictions.

For **RQ3**, we found out empirically that, for our experiment, 100 deployed containers is a decision point to start a

[23] https://www.envoyproxy.io/

new machine automatically using the Infrastructure as Code component of our proposed approach (see Figure 2). We calculated the prediction error of our model as 4.28% based on the widely used Mean Absolute Percentage Error (MAPE) measurement [27] averaged over private and public clouds. Because 30.0% is the common target prediction accuracy in the cloud quality-of-service field [19], the prediction error is reasonable to conclude that our model is highly-accurate and applicable in this domain.

To the best of our knowledge, stateful depletion of containers has yet to be extensively studied in the literature. Moreover, current container orchestration technologies, such as Google Kubernetes[1], consider container depletion minimally. We believe our proposed approach that builds upon our previous work (empirically validated in multiple studies with an open access dataset and code presented in Section VI-A5) can provide a solid base for a set of future work. Therefore, we plan to apply our approach in multiple industrial use cases and cover common cloud scenarios in this field of research. Moreover, as we studied multiple machine types of GCP, we plan to continue our study to provide guidance and empirical measurements on using different machine types.

Acknowledgment

This paper was funded by the Austria Research Promotion Agency (FFG) project Generische eFiskalisierung in der Cloud für unterschiedlichste Organisationen (GECO).

References

[1] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of AADL models. In *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009*, pages 61–71. IEEE, 2009.

[2] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[3] A. Amiri, U. Zdun, and A. van Hoorn. Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures. In *IEEE Transactions on Services Computing (TSC)*, 2021.

[4] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Automatic adaptation of reliability and performance tradeoffs in service- and cloud-based dynamic routing architectures. In *IEEE Transactions on Software Quality, Reliability and Security (QRS)*, 2021.

[5] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Cost-aware multi-dimensional auto-scaling of service- and cloud-based dynamic routing to prevent system overload. In *IEEE International Conference on Web Services (ICWS)*, 2022.

[6] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015.

[7] P. Arcaini, E. Riccobene, and P. Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4):1–35, 2017.

[8] M. Beltrán. Automatic provisioning of multi-tier applications in cloud computing environments. *The Journal of Supercomputing*, 71:2221––2250, 2015.

[9] A. Busch, D. Fuchss, and A. Koziolek. Peropteryx: Automated improvement of software architectures. In *IEEE International Conference on Software Architecture ICSA Companion 2019*, pages 162–165. IEEE, 2019.

[10] A. Chung, J. W. Park, and G. R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[11] J. Comden, S. Yao, N. Chen, H. Xing, and Z. Liu. Online optimization in cloud resource provisioning: Predictions, regrets, and algorithms. In *Publication: Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

[12] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah. A new docker swarm scheduling strategy. In *2017 IEEE 7th international symposium on cloud and service computing (SC2)*, 2017.

[13] G. Galante and L. C. E. de Bona. A survey on cloud computing elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE, 2012.

[14] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 23–27, 2013.

[15] D. G. D. L. Iglesia and D. Weyns. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):1–31, 2015.

[16] C. Kaewkasi and K. Chuenmuneewong. Improvement of container scheduling for docker using ant colony optimization. In *IEEE 10th International Conference on Autonomic Computing*, 2013.

[17] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 261–268. IEEE, 2016.

[18] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang. A new container scheduling algorithm based on multi-objective optimization. In *Soft Computing, 22(23), 7741-7752*, 2018.

[19] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2001.

[20] T. Menouer. Kcss: Kubernetes container scheduling strategy. In *The Journal of Supercomputing, 77(5), 4267-4293*, 2021.

[21] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing*, 2013.

[22] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. In *IEEE Transactions on Software Engineering*. IEEE, 2019.

[23] A. Rago, S. A. Vidal, J. A. Diaz-Pace, S. Frank, and A. van Hoorn. Distributed quality-attribute optimization of software architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2017*, pages 7:1–7:10. ACM, 2017.

[24] V. S. Sharma and K. S. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, page 217–227, New York, NY, USA, 2005. Association for Computing Machinery.

[25] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *9th international conference on knowledge and smart technology (KST)*, pages 254–259, 2017.

[26] M. Sureshkumar and P. Rajesh. Optimizing the docker container usage based on load scheduling. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, 2017.

[27] K. S. Trivedi and A. Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Oxford University Press, 2017.

[28] C. Wohlin, P. Runeson, M. Hoest, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Springer, 2012.