TERA-Scaler for a Proactive Auto-Scaling of e-Business Microservices

Souheir Merkouche[a] and Chafia Bouanaka[b]
LIRE Laboratory, University of Constantine 2-Abdelhamid Mehri, Constantine, Algeria
{souheir.merkouche, chafia.bouanaka}@univ-constantine2.dz

Keywords: Microservice Architectures, Cloud Computing, Auto-Scaling, e-Business, Kubernetes.

Abstract: In this research work, we present a novel multicriteria auto-scaling strategy aiming at reducing the operational costs of microservice-based e-business systems in the cloud. Our proposed solution, TERA-Scaler for instance, is designed to be aware of dependencies and to minimize resource consumption while maximizing system performance. To achieve these objectives, we adopt a proactive formal approach that leverages predictive techniques to anticipate the future state of the system components, enabling earlier scaling of the system to handle future loads. We implement the proposed auto-scaling process for e-business microservices using Kubernetes, and conduct experiments to evaluate the performance of our approach. The results show that TERA-Scaler outperforms the Kubernetes horizontal pod autoscaler, achieving a 39.5% reduction in response time and demonstrating the effectiveness of our proposed strategy.

1 INTRODUCTION

The microservice-based architecture is widely utilized in the design of online applications due to its ability to divide the system into small, loosely-coupled components, which can be developed and updated independently of one another, resulting in a faster development cycle. Additionally, the pay-per-use model of cloud resources has led companies to deploy their systems on the cloud. However, a supplementary effort is required to orchestrate these microservices to ensure proper functioning and optimal resource usage.

Microservices architecture has shown promise in several e-business domains, such as e-commerce and e-learning, as it provides flexibility, reliability, reusability, and scalability. Large e-business companies, including Netflix, eBay, and Amazon, have migrated their monolithic architecture to microservices. The e-learning industry has also considered several scenarios to adopt this architecture, including improving existing e-learning platform architectures(Bauer et al., 2018), migrating custom Learning Management System (LMS) to microservices architectures(Niemelä and Hyyrö, 2019), and developing their own platform(Segura-Torres and Forero-García, 2019). Nonetheless, it is crucial to define orchestration policies dedicated to this domain to ensure the expected quality of each microservice.

Microservices orchestration is a crucial task, as it handles the deployment, auto-scaling, and scheduling of various instances. A proper orchestration guarantees a higher performance of the system with a lower cost. Therefore, orchestration tools development is capturing growing interest, resulting in tools such as Kubernetes(Carrión, 2022), OpenShift(Palumbo, 2022), and Docker Swarm(Marathe et al., 2019). Orchestration tools are not only dedicated to adapt the system to the current workloads but also to predict future workloads and adjust the system accordingly.

Efficient orchestration of e-business applications requires the definition of adequate deployment, scheduling, and auto-scaling policies that preserve their quality of service (QoS). In this work, we propose a dependency-based orchestration strategy based on a proactive approach to ensure service availability for these systems but still maintaining optimal costs.

We present TERA-Scaler, a multicriteria auto-scaling strategy based on the weak and strong dependencies concept introduced in (Bravetti et al., 2019), which allows for proactive auto-scaling without requiring any expert systems or prior knowledge, as it is the case for existing solutions (Rossi et al., 2020)(Niemelä and Hyyrö, 2019) and what actually limits their adoption. We adopt a formal approach due to the fact that auto-scaling is a fault-sensitive process. Inappropriate auto-scaling causes the QoS decrease or extra costs.

[a] https://orcid.org/0000-0003-2250-3947
[b] https://orcid.org/0000-0003-1746-834X

Merkouche, S. and Bouanaka, C.
TERA-Scaler for a Proactive Auto-Scaling of e-Business Microservices.
DOI: 10.5220/0012093500003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 448-455
ISBN: 978-989-758-665-1; ISSN: 2184-2833

The rest of the paper is organized as follows: In section 2, we explain the TERA-Solution philosophy and how it influences the quality in e-business applications. Section 3 discusses existing auto-scaling approaches and recalls the main concepts adopted in our approach. In section 4, we present our auto-scaling approach and the proposed strategy, and we also discuss the Horizontal Pod Autoscaler (HPA) of Kubernetes. In section 5, we implement TERA-Scaler, apply it to the e-commerce application, evaluate its performance and efficiency compared to HPA policy of Kubernetes. Finally, we round up the paper with some perspectives of the actual work.

2 TERA-Solution AND e-BUSINESS

Orchestrating a microservice-based application on the cloud follows the process depicted in Figure 1. Initially, microservice instances are allocated to pods that are responsible of equipping them with the required resources. Subsequently, the monitoring task tracks changes in the workload and adjusts the number of microservice instances or the amount of resources allocated accordingly, depending on whether horizontal or vertical auto-scaling is being performed. Consequently, new instances are planned, deployed, and monitored to ensure optimal system performance.

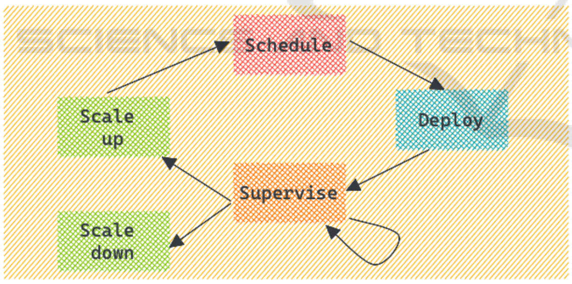

Figure 1: The orchestration process.

Auto-scaling is a critical operation that depends heavily on the nature and requirements of the underlying application, various solutions have been proposed for automating this process. Some solutions focus on latency-sensitive applications (Rossi et al., 2020)(Crankshaw et al., 2020), while others aim to optimize costs(Yang et al., 2014), maintain SLAs(Souza and Netto, 2015), or optimize QoS(Goswami(Mukherjee) et al., 2019). TERA-Solution is a dependency-aware orchestration strategy that includes TERA-Scheduler (Merkouche and Bouanaka, 2022b) and TERA-Scaler. TERA-Scheduler is responsible for the deployment and scheduling phase, while TERA-Scaler is the auto-scaling component. In a previous work, we presented and evaluated the efficiency of TERA-Scheduler by applying it to an e-learning system. In this paper, we introduce TERA-Scaler and evaluate its efficiency by applying it to an e-commerce microservice-based application.

Our proposed solution adopts the auto-scaling approach presented in (Merkouche and Bouanaka, 2022a), which enables proactive auto-scaling based on dependencies between application components, thereby avoiding resource-intensive and time-consuming solutions. The main idea behind our approach is that a microservice future state is predicted from updates on the dependent one state. Whenever a microservice is scaled up/down, its dependencies are also scaled up/down. However, the number of replicas to be added/removed is computed independently for each microservice according to its actual workload.

This approach is well-suited for pipeline or multi-pipeline systems, such as e-mail processing systems. E-business applications may also be considered as multi-pipeline systems, including e-commerce, e-learning, e-search and other systems where a customer firstly visits the items list page, then passes through the order page and the payment one, with each page defined in a separate microservice. In an e-learning example, if a student wants to download a course chapter, he passes through several pages, each with different microservices. We believe that the dependency-based approach is ideal for e-business systems and significantly improves their performance. In the following section, we provide an overview of existing approaches and recall basic concepts necessary to understand the adopted approach and the associated auto-scaling policy.

3 BACKGROUND

Numerous studies have been conducted to automate the auto-scaling process, resulting in various solutions that focus on different aspects of the system and follow different approaches. Existing solutions primarily use reactive approaches (Alexander et al., 2020)(Tsagkaropoulos et al., 2021)(Souza and Netto, 2015), where the amount of allocated resources is increased or decreased based on system metrics monitoring such as CPU and memory usage or input data rate, depending on the current workload. However, this approach remains time-consuming despite its effectiveness. In contrast, proactive approaches(Rossi et al., 2020)(Bauer et al., 2019)(Imdoukh et al., 2020) utilize machine learning to predict the future workload of the system and adapt the amount of allocated resources accordingly. However, machine learning techniques are costly and time consuming, and the predictions may not always be entirely accurate, resulting in lower performance or additional costs. Hence, we adopt a dependency-based approach to predict the future workload and thus scaling the underlying microservices.

Specifically, we consider the workload of the microservice at the entry point of the pipeline, as the key element of the rest of the pipeline. Some concepts need to be recalled before detailing our approach.

3.1 Identifying Microservice Dependencies

A microservice dependencies are identified based on the definitions provided in (Bravetti et al., 2019). Strong dependencies are the microservices that are mandatory for the proper functioning of the considered microservice, while weak dependencies are the ones that must be deployed before the end of its deployment. These dependencies are defined in the deployment file of each microservice.

Technically, strong dependencies represent microservices that the given microservice will certainly call to accomplish its task, while weak dependencies are optional. As an example, the order microservice in an e-commerce application is optional and only called by the items list microservice if the user decides to view and order a specific item, while the items database microservice is mandatory to show the items list and related information. Therefore, the workload scaling up/down of a microservice directly influences its strong/weak dependencies and implies proactively adapting the microservice's strong dependencies, and computing auto-scaling plans for its weak dependencies.

3.2 Auto-Scaling Policy

The auto-scaling policy implemented in our solution is based on the multicriteria policy described and formally defined in (Merkouche and Bouanaka, 2022a). The approach considers the input data rate, CPU, and memory usage of each microservice as supervised metrics. For each supervised metric, a MAPE loop (Lemos et al., 2013) is defined to determine the corresponding adaptation plans. The set of MAPE loops shares a centralized monitor that supervises all the metrics. When a metric is violated, an adaptation is triggered in the corresponding loop. An extended plan is also shared between the loops to compute compromise plans when several metrics are violated. The auto-scaling policy can be summarized in the following steps:

- The Shared Monitor collects metrics of each microservice from the environment layer and verifies if an adaptation is needed.
- When an adaptation is needed for a given microservice, i.e., when one of the microservice's metrics is violated (not within the thresholds interval), the Analyze and Plan components of the corresponding loop define the adaptation plan from the violated metric viewpoint.
- The Extended Plan computes a compromise adaptation plan if several metrics were violated. It also proactively computes proactive adaptation plans for the strong dependencies and recommended adaptation plans for the weak dependencies.
- The Execute element applies the adaptation plan on the microservice and the proactive adaptation plans on the strong dependencies, then it notifies the weak dependencies by their recommended adaptation plans. Figure 2 illustrates this process.

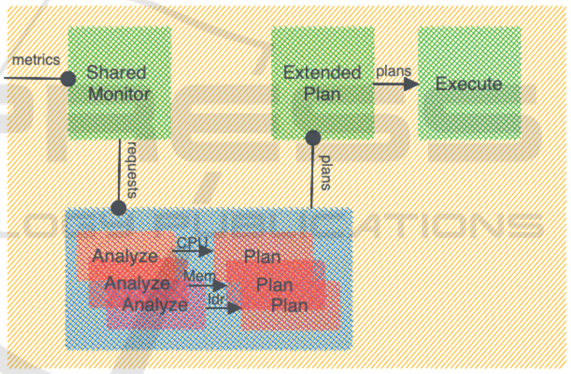

Figure 2: The proactive auto-scaling process.

4 TERA-Scaler

Auto-scaling is a critical task when deploying a system in the cloud, as it has a significant impact on system performance and cost. It is essential to adopt an efficient and suitable auto-scaling approach for the considered system to ensure that it meets the desired system qualities, enhances performance, and reduces costs. However, considering a single metric during auto-scaling is insufficient to guarantee that the microservice is provided with the necessary resources to meet the current workload. Therefore, a multi-criteria policy that includes CPU and memory usage, as well as input data rate, needs to be considered. The solution is not limited to these three metrics, other metrics can be easily introduced since each metric is adapted independently. Kubernetes and several other tools enable the monitoring of a broad set of metrics, and custom metrics can even be defined.

The auto-scaler is responsible for adjusting the amount of resources according to the workload variations, it monitors each microservice metrics, detects metric violations, and computes the required amount of replicas to add or remove to adapt to the violated metrics. When a certain metric is violated, the auto-scaler triggers an adaptation computing process that takes a processing time, which affects the system's performance; the faster this process is, the less waste of performance (in the case of scale up) and costs (in the case of a scale down). Therefore, a proactive method is needed to allow workload prediction. However, these methods involve learning techniques that are costly.

To address the issues mentioned above, we propose TERA-Scaler, an efficient microservice auto-scaling strategy based on a proactive approach that relies on the dependencies between microservices to predict future workload. The efficiency of the TERA-Scaler autoscaling strategy is evaluated compared to Kubertenes and it horizontal Pod Autoscaler(HPA).

4.1 TERA-Scaler Strategy

In order to achieve a proactive auto-scaling, TERA-Scaler computes 03 types of adaptation plans: a compromise adaptation plan, a proactive adaptation plan and a recommended adaptation plan.

4.1.1 The Adaptation Plans

An adaptation plan represents the required amount of replicas to ensure a well functioning of the microservice. This step consists in computing the amount of replicas needed to adapt the system, with regard to each metric by increasing/decreasing the replicas number and computing the new value of the metric. If the new value ranges within the desired thresholds the corresponding replicas number is returned, otherwise the process is repeated(see algorithm 1).

4.1.2 The Compromise Adaptation Plan

Each of the adaptation plans already identified is pondered to define a compromise plan that fulfills all the violated metrics. Adaptation plan weighting depends on the which is a percentage that defines the priority of each metric over the other metrics for the microservice. In our case, we considered 03 metrics, namely the CPU usage, Memory usage and the input data rate, other metrics can be considered according to the system's nature.

Algorithm 1: Adaptation plan computing.

```
Input: metric, $min_{threshold}$ , $max_{threshold}$, $R_{old}$
   Output: $R_{new}$
 $R_{new} \Leftarrow R_{old}$
 $R_{new} \Leftarrow R_{old}$
 while metric $> max_{threshold}$ do
     metric $\Leftarrow$ ( metric $\times R_{old}$)/($R_{new}$ + 1)
     $R_{new} \Leftarrow R_{new}$+1
 end while
 while metric $< min_{threshold}$ do
     metric $\Leftarrow$ ( metric $\times R_{old}$)/($R_{new}$ - 1)
     $R_{new} \Leftarrow R_{new}$-1
 end while
       return $R_{new}$
```

4.1.3 The Proactive Adaptation Plans

The proactive adaptation plans are used to adapt strong dependencies of the microservice. To do so, for each microservice in the strong dependencies list a plan is computed by multiplying the replicas number obtained by the compromise adaptation plan by a factor R_{ms} where R_{ms} is the number of replicas needed from a strong dependency to deal with a replica of the microservice. For example, considering ms_i a strong dependency of ms_j, where a replica of ms_j needs 02 replicas of ms_i to run, if the compromise plan of ms_j is 04 then the proactive plan of ms_i is 08.

4.1.4 The Recommended Adaptation Plans

It is computed as the proactive adaptation plan, expect that it is dedicated to the microservice weak dependencies, and instead of being applied, it is only computed and suggested to those microservices so that when the workload changes it is applied instantly. Algorithm 2 illustrates the computing of these 03 plans.

Algorithm 2: Final plans computing.

```
Input: $P_{CPU}$, $cpu_{priority}$, $P_{Mem}$, $mem_{priority}$, $P_{idr}$, $idr_{priority}$, $dep_{strong}$, $dep_{weak}$
   Output: $P_{compromise}$, $Ps_{proactive}$, $Ps_{recommended}$
 $P_{compromise} \Leftarrow ((P_{CPU} \times cpu_{priority}) + (P_{Mem} \times mem_{priority}) + (P_{idr} \times idr_{priority}))/3$
 for (ms, $R_{ms} \in dep_{strong}$ do
     $Ps_{proactive} \Leftarrow Ps_{proactive} + (P_{compromise} \times R_{ms})$
 end for
 for (ms, $R_{ms}$) $\in dep_{weak}$ do
     $Ps_{recommended} \Leftarrow Ps_{recommended} + (P_{compromise} \times R_{ms})$
 end for
       return $P_{compromise}$, $Ps_{proactive}$, $Ps_{recommended}$
```

Following this strategy, TERA-Scaler anticipates adaptations according to workload variations for a given microservice. TERA-Scaler not only ensures adapting the considered microservice but also its dependencies and hence proactively prepares other microservices for the upcoming workload.

4.2 Kubernetes and the Horizontal Pod Autoscaler

K8s[1] is an open-source orchestration tool introduced by Google in 2014, designed to simplify the packaging and execution of containerized applications, workloads, and services. It provides a platform to deploy application services across multiple containers in a cluster, enabling their scaling and ensuring their integrity over time. K8s supports various container runtime engines, including Docker[2] containerd, CRI-O, and other K8s Container Runtime Interface (CRI) implementations. Pods are the basic units for running containers in a K8s cluster, representing an instance of a microservice and always belonging to a namespace.Figure 3 shows a simple K8s cluster composed of five nodes using Docker as the container runtime. In a K8s cluster, containers are run on elementary

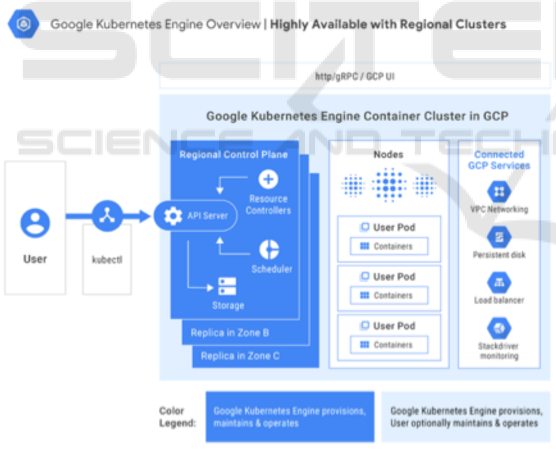

Figure 3: Google K8s Engine Architecture.

units called pods, each one representing an instance of a microservice and always belongs to a namespace (Nguyen et al., 2020). Pods representing instances of the same microservice are identified by similar specifications.

A K8s cluster comprises worker nodes for deploying pods and a master node for orchestrating them. The master node consists of an API server, a controller manager, etcd, and kube-scheduler, which is K8s's default scheduler.Figure 4 illustrates the architecture of K8s.

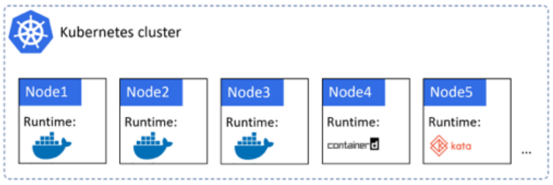

Figure 4: Google K8s Engine Architecture.

Horizontal scaling in K8s involves deploying more pods in response to increased load, while vertical scaling involves assigning more resources, such as memory or CPU, to the pods already running for the workload. The Horizontal Pod Autoscaler (HPA) is a Kubernetes API resource and controller that scales the number of replicas of a target (e.g., a Deployment) based on observed metrics such as average CPU and memory utilization. The HPA periodically adjusts the desired number of replicas to match the workload's needs and ensures that the number of pods is above the minimal threshold. Figure 5 depicts the HPA concept. Additional details about K8s's functionalities and components can be found in the official documentation.

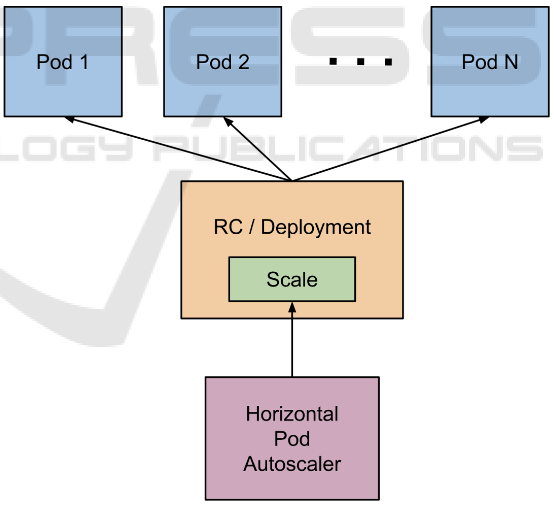

Figure 5: The Horizontal Pod Autoscaler concept.

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we aim to evaluate the efficiency of the proposed approach in auto-scaling microservice-based systems and its impact on their performance. Therefore, we implement TERA-Scaler and test it on a simple e-commerce application as shown in figure 6,

[1]https://K8s.io/docs/
[2]https://hub.docker.com

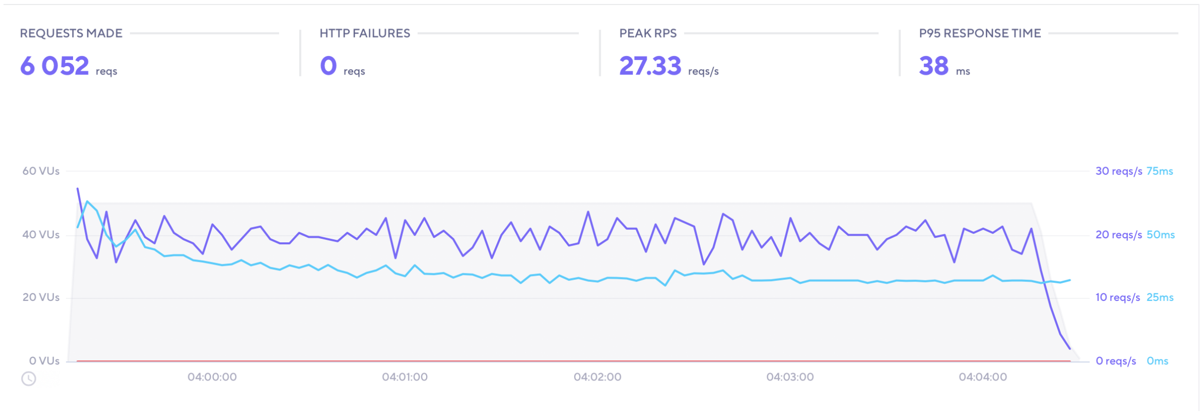

Figure 7: The response time using the Kubernetes HPA policy.

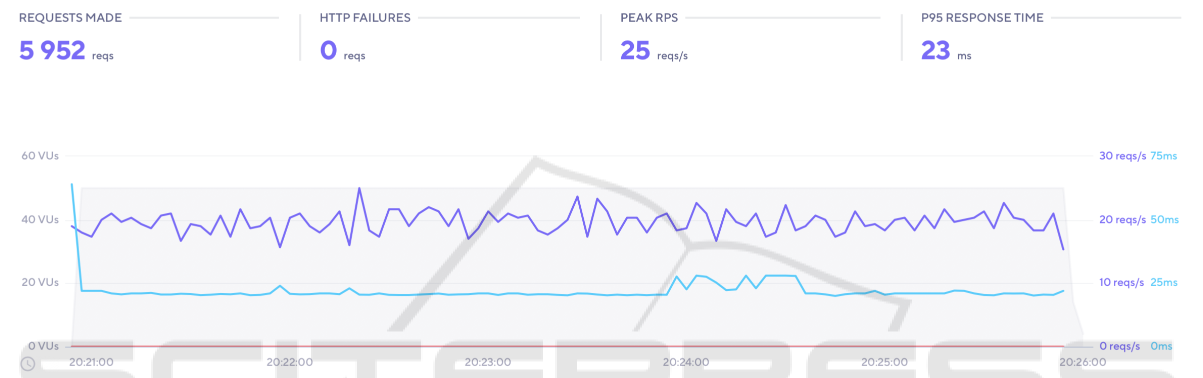

Figure 8: The response time when using TERA-Scaler.

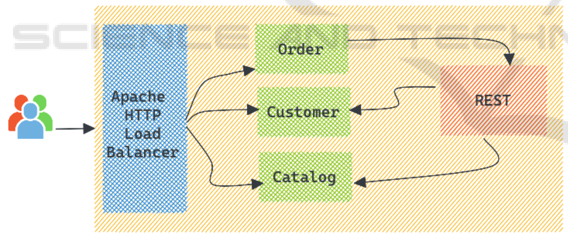

Figure 6: The e-commerce application's architecture.

the considered example is composed of the following microservices:

- Catalog microservice: to show and handle available items in the catalog.
- Customer microservice: to handle the customers data.
- Order microservice: to order items.

The microservices communicate through a REST API and Apache is used as a Load Balancer to forward http requests to the microservices (Load Balancer). The considered example was deployed in a k8s cluster and tested twice, the first time using a set of HPA each applied to a microservice deployment, and the second one using TERA-Scaler.

5.1 Implementing TERA-Scaler

Implementing the HPA is an easy task, we create a deployment file specifying the deployment to scale, the resource to monitor and the target average of this resource. For our test, we deployed 03 HPAs, each targeting one of the application's microservices. We considered the Memory usage as a supervised metric with 30% as a maximal threshold for all of them.

To implement our auto-scaler, we used the custom-pod-autoscaler framework[3](CPA) to define our auto-scaler strategy and implement the presented algorithms. We define the same thresholds as in the case of HPA and use k6[4] to apply a workload on the application in both cases with the same test and duration. Dependencies between microservices were defined as follows:

- Catalog: having Customer as a weak dependency and Order as a strong dependency.
- Order: having Customer as a strong dependency.

To scale them, we deploy 02 CPAs, where:

[3]https://custom-pod-autoscaler
[4]https://k6.io/docs/

- The first CPA scales the Catalog microservice , proactively scales the Order microservice and define scaling plan for the Customer microservice.
- The second CPA scales the Order microservice and proactively scales the Customer microservice.

According to the evaluation tests, the response time reduction rate is up to 39%. Figure 7 shows the application's response time when using the HPA and figure 8 illustrates it when using TERA-Scaler.

6 CONCLUSION

The purpose of our contribution is to propose an auto-scaling solution for e-business applications, with the aim of optimizing performance and reducing costs in the cloud environment. Although our proposed solution, TERA-Scaler, is suitable for a range of applications, it is particularly well-suited for pipelines and component-interactive systems.

Our custom auto-scaler employs a dependency-based approach to proactively adjust resource allocation to microservices in the cloud, while considering the dependencies and quality requirements of each microservice through a multicriteria approach.

To implement this strategy, we use the CPA framework to define the TERA-Scaler policy and related functions, which are then executed on the application. However, by leveraging the TERA-Scheduler in the deployment environment, to ensure the deployment of each microservice together with its dependencies on the same node, our policy can be defined with HPA commands and implemented without any intermediary, resulting in faster and more efficient auto-scaling.

As a future direction, we aim to integrate the elements of TERA-Solution into an orchestration tool for microservice-based applications, taking into account dependencies between microservices and the maintenance of quality requirements for each microservice.

REFERENCES

Alexander, K., Hanif, M., Lee, C., Kim, E., and Helal, S. (2020). Cost-aware orchestration of applications over heterogeneous clouds. *PLOS ONE*, 15(2):1–21.

Bauer, A., Lesch, V., Versluis, L., Ilyushkin, A., Herbst, N., and Kounev, S. (2019). Chamulteon: Coordinated auto-scaling of micro-services. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2015–2025.

Bauer, D. A., Bauer, D. A., Penz, B., Mäkiö, J., and Assaad, M. (2018). Improvement of an existing microservices architecture for an e- learning platform. In *STEM Education*.

Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., and Zavattaro, G. (2019). Optimal and automated deployment for microservices.

Carrión, M. (2022). Kubernetes as a standard container orchestrator - a bibliometric analysis. *Journal of Grid Computing*, 20.

Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., and Tumanov, A. (2020). Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491.

Goswami(Mukherjee), B., Sarkar, J., Saha, S., Kar, S., and Sarkar, P. (2019). Alvec: Auto-scaling by lotka volterra elastic cloud: A qos aware non linear dynamical allocation model. *Simulation Modelling Practice and Theory*, 93:262–292. Modeling and Simulation of Cloud Computing and Big Data.

Imdoukh, M., Ahmad, I., and Alfailakawi, M. (2020). Machine learning based auto-scaling for containerized applications. *Neural Computing and Applications*, pages http://link.springer.com/article/10.1007/s00521–019.

Lemos, R., Giese, H., Müller, H., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., and Wuttke, J. (2013). *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32.

Marathe, N., Gandhi, A., and Shah, J. M. (2019). Docker swarm and kubernetes in cloud computing environment. *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 179–184.

Merkouche, S. and Bouanaka, C. (2022a). A proactive formal approach for microservice-based applications auto-scaling. In *proceedings of CEUR Workshop*, volume 3176, pages 15–28.

Merkouche, S. and Bouanaka, C. (2022b). Tera-scheduler for a dependency-based orchestration of microservices. In *2022 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, pages 1–8.

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., and Kim, S. (2020). Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16).

Niemelä, P. and Hyyrö, H. (2019). Migrating learning management systems towards microservice architecture. In *SSSME-2019: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution (pp. 10- 20)*. (CEUR Workshop Proceedings; Vol. 2520). CEUR-WS.

Palumbo, A. (2022). Literature review on kubernetes & red-hat openshift container platform.

Rossi, F., Cardellini, V., and Presti, F. L. (2020). Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37.

Segura-Torres, D. A. and Forero-García, E. F. (2019). Architecture for the management of a remote practical learning platform for engineering education. *IOP Conference Series: Materials Science and Engineering*, 519(1):12–20.

Souza, A. A. D. and Netto, M. A. (2015). Using application data for sla-aware auto-scaling in cloud environments. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 252–255.

Tsagkaropoulos, A., Verginadis, Y., Papageorgiou, N., Paraskevopoulos, F., Apostolou, D., and Mentzas, G. (2021). Severity: a qos-aware approach to cloud application elasticity. *Journal of Cloud Computing*, 10.

Yang, J., Liu, C., Shang, Y., Cheng, B., Mao, Z., Chunhong, L., Niu, L., and Junliang, C. (2014). A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16:7–18.