VS20: Vertical Scale To Zero for Kubernetes Pods

EDOARDO CARLOTTO

ACM Reference Format:
Edoardo Carlotto. 2024. VS20: Vertical Scale To Zero for Kubernetes Pods. 1, 1 (November 2024), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This report provides a comprehensive overview of the VerticalScaleToZero research project, which was developed as part of the Software Engineering 2 course. The primary objective of this project is to present an innovative solution aimed at reducing the resource consumption associated with cloud applications. By optimizing the allocation of resources, VerticalScaleToZero effectively tackles the cold start issue—a prevalent challenge in cloud environments that can hinder performance and responsiveness when scaling applications. The cold start issue often arises when applications need to start up new instances to handle increased demand, leading to delays and inefficiencies. Our solution not only mitigates this problem but also introduces minimal overhead, ensuring that performance remains largely unaffected.

2 PROBLEM STATEMENT

Cloud applications are increasingly executed onto lightweight containers that can be efficiently managed to cope with highly varying and unpredictable workloads. Kubernetes [1] has emerged as the leading container orchestrator in cloud environments. Kubernetes operates across multiple nodes, physical servers, or virtual machines, to schedule containers: this means on each node multiple containers are scheduled, resulting in a compounded loss of resources as the number of containers increases. One of the features offered by Kubernetes is the possibility of managing resources allocated to containers. Specifically, it is possible to set how many resources to allocate (or reserve, dedicated) to a container but also how many resources it can consume at most. If a container uses fewer resources than the amount dedicated to it, the remaining portion can not be used by any other application and therefore is wasted.
As an example, consider the use case of an application offering food delivery services. For this service to function effectively, it needs to manage the surge of user requests during the peak period. If the resources allocated to the application match the amount needed during the peak, there is a risk of over-provisioning since most of the time a lot of resources will be reserved for such application without being utilized. Another illustrative example of the significance of the issue addressed in this research project involves applications that have a pipelined architecture. In such systems, each request is processed in different stages, one after the other, allowing distribution of the pipeline across multiple containers. However, when the traffic is insufficient to fill the pipeline, containers dedicated to certain stages will not utilize all the resources dedicated to them.

[1]https://kubernetes.com

Author's address: Edoardo Carlotto, edoardo.carlotto@mail.polimi.it.

One potential solution to address these limitations is the following: turn off containers when they are not needed an spin up some new containers as the workload increases. This solution can effectively minimize the amount of used resources. However, it introduces the issue of cold starts. When a container is initiated, several operations, such as data structure allocation, need to be performed before it can begin handling requests. As a result, requests that reach the servers when there are no available containers to serve them must wait for the newly launched containers to become operational. This leads to increased response time for some requests. Furthermore, the overhead associated with cold starts implies also that processing power is consumed to initialize containers each time they are activated, ultimately diminishing the efficiency of resource utilization.

3 RELATED WORK

The most efficient way to ensure containers do not waste resources is to turn them off as soon as they are not needed. This means containers need to be activated frequently, leading to the cold start problem. The cold start problem in Kubernetes refers to the delay that occurs when a new pod is created and initialized. This delay can impact the responsiveness and performance of applications, especially in environments where rapid scaling is required. There are multiple solutions to try and mitigate this issue, some of which are offered directly by cloud providers, but none of them is actually able to make an efficient usage of resources while mitigating the issue.

3.1 HPA and VPA

HPA (Horizontal Pod Autoscaler) and the VPA (Vertical Pod Autoscaler) are two autoscalers offered by Kubernetes to deal with the varying workload. They are two powerful tools designed to manage resources for containerized applications. HPA automatically adjusts the number of pod replicas in a deployment based on observed CPU utilization or other select metrics, allowing applications to scale horizontally to meet demand. Conversely, VPA optimizes resource allocation by adjusting the CPU and memory requests for individual pods based on their actual usage, enabling vertical scaling. However, a significant limitation of these tools is that they cannot be used together on the same pods; utilizing both may lead to conflicts, as HPA attempts to change pod counts while VPA modifies resource requests, which can create instability in resource management. Furthermore, both tools operate by continuously activating and deactivating containers. They shut down containers when some are not needed or don't have enough resources dedicated to them and spin up new ones as the workload increases and resources constraints need to be modified. This approach entails paying initialization overhead with each operation. Additionally, it is important to note that VPA cannot scale CPU allocation down to zero, meaning that a certain amount of resources remains allocated even when they go unused.

3.2 KOSMOS

KOSMOS [2] is a novel autoscaling solution for Kubernetes, it offers a unique combination of both Vertical and Horizontal autoscaling capabilities, seamlessly integrating with the Kubernetes API to enhance resource management. As explained in the previous paragraph, HPA and VPA components are not designed to work together and this is precisely where KOSMOS steps in, effectively bridging this gap by delivering a comprehensive autoscaling solution that allows for dynamic and precise allocation of resources based on real-time demands. This is an effective solution but one major drawback is its inability to scale resource requests down to zero. The logic of KOSMOS does not completely free resources dedicated to containers when they are not needed. Consequently, each container will always occupy a

[2]https://github.com/deib-polimi/kosmos

minimum level of resources, regardless of the actual workload requirements, leading to lower efficiency of resource utilization.

4 SOLUTION

This report presents a solution designed to dynamically adjust container resources, enabling vertical scaling down to zero when they are not in use. This approach helps to prevent having containers lock resources even when there is no workload to use them. This research project leverages a feature of Kubernetes, InPlacePodVerticalScaling [3], that allows to change the CPU allocated to a container without requiring a restart, ensuring seamless operation from the application's perspective. Specifically, Kubernetes' API allows for the specification of resources requests (CPU and RAM fully dedicated) and resources limits (maximum CPU and RAM potentially utilized) for each container. From a high level perspective, the core functionality of this project involves monitoring traffic to a designated container and utilizing the Kubernetes API to adjust CPU requests and limits to zero when the application is not actively handling requests.
The solution provided here consists of two components: a proxy and a central controller. The proxy is a container, deployed alongside each relevant application container, responsible for monitoring its operation and making necessary updates. The central controller monitors all applications deployed on Kubernetes and manages the associated proxy containers. Currently, the project exclusively supports applications communicating over HTTP. For the purpose of this research project, this is not a significant limitation as utilizing with HTTP requests sufficiently demonstrate the feasibility of the proposed solution while still supporting a diverse array of applications.
The programming language selected for this research project is Go, a compelling choice due to its speed and lightweight nature, which allows for the development of high-performance applications that efficiently manage resources. The language's simplicity and concurrency features make it easy to implement a reverse proxy, essential for handling incoming requests seamlessly. Leveraging Go's high-performance capabilities allows the proxy to forward requests with only a negligible increase in application response time. Its efficiency is further highlighted by the minimal memory footprint of compiled code, enabling the addition of a proxy container for each application container without significantly increasing resource utilization. Additionally, Go's strong integration with Kubernetes—both as its primary language and an ecosystem filled with libraries—facilitates straightforward interaction with Kubernetes APIs.

4.1 Proxy container

Before diving into the specifics of this component, it is essential to mention what pods are in the context of Kubernetes. A pod is essentially a group of containers designed to work tightly together and share network interfaces. This shared network environment allows containers within a pod to communicate seamlessly, as if they were two processes operating on the same machine. When an application is deployed to a Kubernetes cluster with *n* replicas, the cluster schedules and keeps running *n* pods, each composed by the same containers. One common reason to deploy two containers in the same pod is to implement a sidecar pattern: a single-node pattern made up of two containers. The first is the application container, which contains the core logic for the application. In addition to the application container, there is a sidecar container, which role is to augment and improve the application container, often without the application container's knowledge.
The proxy container component is a container that is present in each pod of the applications of interest. Its intended purpose is to monitor traffic and allocate resources for the containers within its own pod. The way in which the proxy

[3]https://kubernetes.io/blog/2023/05/12/in-place-pod-resize-alpha/

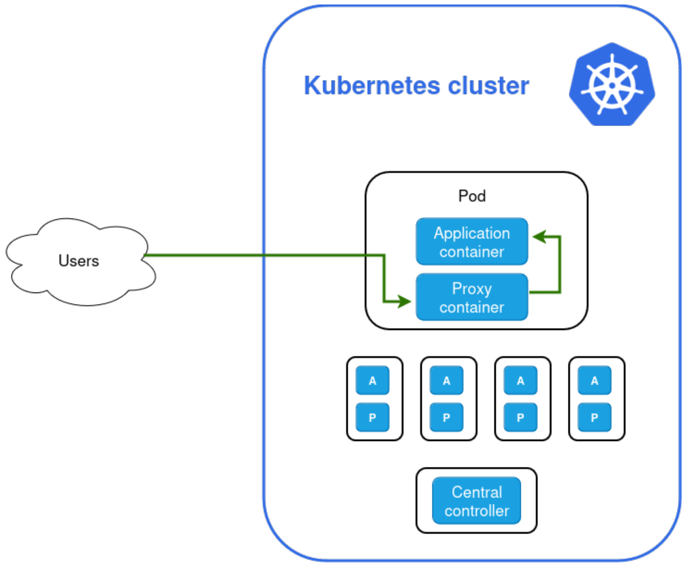

Fig. 1. Proxy Sidecar Architecture.

container is deployed implements a sidecar pattern having the role of the sidecar: indeed, its task is to improve the efficiency of the application container, being transparent to it. A significant advantage of this approach is the proxy container's complete independence from the application itself, eliminating the need to understand the application's internal workings. Consequently, the code developed in this research project can be seamlessly deployed with any HTTP application within a Kubernetes cluster. This component implements two distinct functionalities: it observes the traffic being exchanged by the application and interfaces with the cluster's API to dynamically scale CPU resources as needed. Figure 1 illustrates an example of the proxy designed in this project and deployed as sidecar container.

To effectively monitor the traffic being exchanged in this context, utilizing a reverse proxy proves to be an excellent solution. A reverse proxy acts as an intermediary, standing between clients and the backend application, and forwarding client requests to the backend. One of the primary advantages of employing a reverse proxy is it being able to monitor the requests it has to forward. Moreover, with such a component it is possible to store incoming requests until the target application is ready to handle them, ensuring that no requests are lost if the application is not ready. These capabilities make a reverse proxy an ideal choice in this scenario.

The second feature of the proxy-container component is to scale resources. It achieves this by using the Kubernetes API to set CPU requests and limit of the target container to zero when there are no requests to process. As soon as a new request is received, the container's resources are restored to their original values. One important configuration parameter in this operations is the one that determines the number of seconds after the last request is processed before the application is deactivated. A lower value means that the application container will be shut down more often, trying to use less resources, while a higher value implies that less requests will encounter the issue of cold start; the evaluation section explains with more details the importance of this parameter.

4.2 Central controller

To understand why there is the need for a central controller, it is essential to first understand the concept of deployments in Kubernetes. A deployment serves as a blueprint detailing how a certain application should be deployed, configurations such as the containers to use and the number of desired replicas are specified with this manifest. When a developer applies a deployment to run an application in the cluster, the deployment will not include the proxy container. Consequently, it is needed to adjust the deployment and integrate the proxy container appropriately. This is where the central scheduler comes into play. It continuously monitors every deployment within the cluster, filtering for those that are relevant. Upon identifying relevant deployments, then it interacts with Kubernetes' API to retrieve each deployment that needs to be updated, modifies it injecting the proxy container and the applies the edited deployment back to the cluster.
While injecting the proxy container, the central controller specifies the configuration for it. To determine which deployments are relevant, this component looks for a tag within the manifest. This enables developers to selectively modify, injecting the proxy container, only the applications they choose.

5 EVALUATION

This section introduces the experiments that were conducted in order to assess the performance of the proposed solution. The empirical evaluation aims to answer the following research questions:

- **RQ1.** How much overhead does the proposed solution introduce?
- **RQ2.** How does the proposed solution manage cold starts?
- **RQ3.** How does the proposed solution perform in general?
- **RQ4.** Does the proposed solution improve resource usage?

5.1 Experimental Setup

The empirical results obtained in this section were achieved using a Kubernetes cluster deployed in the following way:

- Relevant hardware specs: AMD Ryzen 7 2700X (8 core, 16 threads 3.7 GHz), 16 GB RAM DDR4;
- Operating system: Windows 11, build 10.0.22631;
- Virtualizazion engine: Docker desktop v4.26.0;
- Cluster details: kind 0.20.0, kindes node kubernetes version 1.28.0;
- Golang details: versions 1.21.5 and 1.22.0.

To conduct the measurements, an representative workload was built. This workload performs floating point operations, to mimic the behavior of a back-end application engaged in computational tasks. By utilizing a configurable parameter, it is possible to easily adjust the amount of operations executed: this allows to effectively model target applications with varying levels of workload complexity.

5.2 RQ1. How much overhead does the proposed solution introduce?

The evaluation of the overhead introduced by the newly proposed components reveals that their impact on system performance is almost negligible. We monitored resource consumption using *Kubernetes metrics-server* to obtain detailed measurements of CPU and memory usage. The central controller, responsible for orchestrating the scaling mechanism, exhibits a low memory footprint, utilizing only 7 MB of RAM while remaining largely idle during operation.
In addition to the controller, each sidecar container, deployed alongside the application pods to facilitate vertical scaling, consumes approximately 11 MB of memory and 3 mCPU. These figures suggest that the sidecar containers are

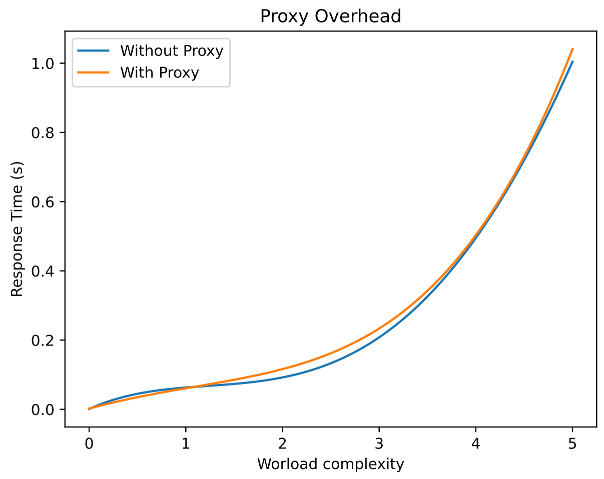

Fig. 2. Proxy Overhead.

lightweight, ensuring that the memory and CPU resources required for their operation are negligible, especially when compared to the overall resource needs of the primary application workloads they accompany.

To further evaluate the impact of these components, we measured the potential latency introduced by the sidecar's reverse proxy functionality, which is responsible for handling network requests to the application pods. Specifically, we focused on response times to determine whether the introduction of the proxy layer caused any noticeable delay in communication between the pods and external clients. Figure 2 presents these findings, comparing response times across a series of tests conducted with workloads of varying complexity. These workloads were designed to represent a range of computational demands, allowing us to assess the overhead under different operational conditions.

The results indicate that the additional overhead in terms of response time is minimal. Across all test scenarios, the reverse proxy introduced only a minor increase in response time. The ratio of response time with the proxy to response time without the proxy remained consistently low, with the proxy-enabled configuration performing within acceptable limits for real-time applications. Specifically, the average response time with the proxy was only, at most, 1.2 times that of the configuration without the proxy, demonstrating that the introduced components do not significantly degrade system performance.

Answer to RQ1: *The proposed controller requires only 7 MB and is mostly idle. The proposed proxy is lightweight (11 MB and 3mCPU) and adds negligible overhead to the response time.*

5.3 RQ2. How does the proposed solution manage cold starts?

We conducted a targeted experiment designed to simulate real-world conditions to better understand the impact of cold starts on application response times. In this experiment, we sent requests at a constant rate to a sample workload while measuring the response times. The experiment was structured to isolate and highlight the effects of cold starts on overall performance.

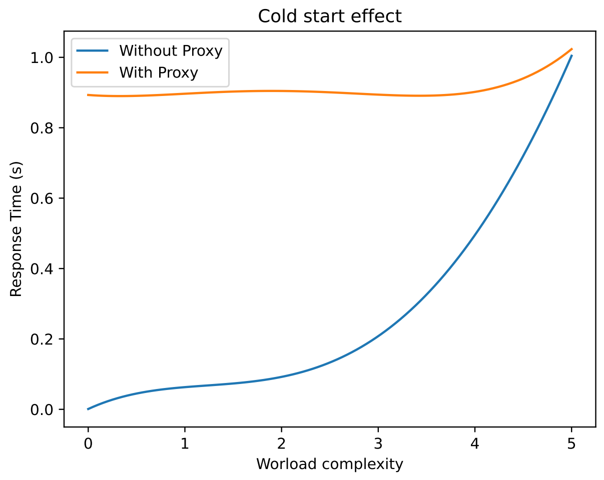

Fig. 3. Proxy Cold Start Performance.

In this experiment the proxy container was configured to follow a greedy shutdown policy. Specifically, after processing and sending the final response to a client, the application container was immediately shut down. Thus, whenever a new request arrived, the application container first had to be initialized before the request could be processed. This setup ensured that each incoming request incurred a delay due to the cold start process, as the application container was not running continuously and had to be restarted upon receiving a new request.

Figure 3 presents the findings from this experiment, illustrating the relationship between the cold start delays and the overall response times under different workload conditions. The results reveal a clear pattern: for workloads that typically have short response times, the cold start overhead significantly impacts the overall response time. In these cases, the time taken to initialize the container dominates, often exceeding the actual processing time of the request itself. This is especially true when the processing requirements of the application are minimal.

However, as the complexity of the targeted workload increases, leading to longer processing times, the influence of the cold start diminishes. In these scenarios, the application processing time begins to outweigh the cold start delay, reducing the relative impact of the initialization overhead on the total response time. In other words, while cold starts still add latency, the overall effect becomes less noticeable as the application's workload demands grow.

It is important to note that the choice of a greedy shutdown policy in the proxy container highlight the cold start problem the most. By terminating the application container immediately after it finishes processing each request, this policy creates a worst-case scenario where a cold start is required for every new incoming request. This experimental configuration was intentionally designed to demonstrate the effects of cold starts in an unmitigated environment, providing a clear understanding of the performance penalty associated with frequent cold starts. In practical applications, strategies such as pre-warming or keeping containers in a ready state could be used to minimize the impact of cold starts, but these were deliberately excluded from this experiment to fully expose the cold start issue.

Answer to RQ2: *In the worst case, cold start scenarios, the proposed solution adds a minimal overhead when handling complex workloads but introduces significant delays when dealing with simple workloads.*

5.4 RQ3. How does the proposed solution perform in general?

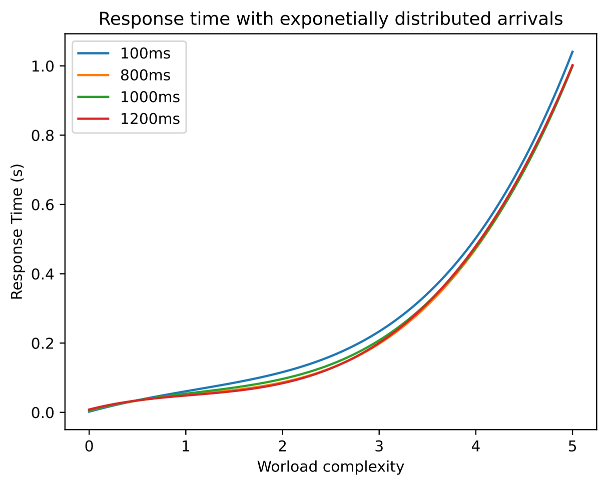

Fig. 4. Response time with varying workload.

We conducted a series of controlled tests to thoroughly evaluate the performance of the proposed solution. These tests were designed to assess how well the system performs under varying conditions, particularly in response to changes in traffic patterns and workload complexity. The proxy container was configured with a policy that dictated the shutdown of the application container exactly $1000ms$ after processing the last request, provided no new requests were received within that window. This one-second buffer was chosen to simulate a typical idle time before scaling down the application to zero.

The experiment involved sending a consistent set of requests, each with varying levels of computational complexity, to the application. These requests were tested in repeated iterations, allowing us to observe performance under different conditions. A key variable in the experiment was the *think time*, which refers to the interval between consecutive requests. This think time was varied in milliseconds to simulate different user behaviors and traffic patterns, reflecting real-world scenarios where user requests may arrive in bursts or with irregular intervals. The think time was sampled from an exponential distribution with $\lambda \in \{100ms, 800ms, 1000ms, 1200ms\}$. Those value were chosen to show how the proposed solution behave when the think time is almost none ($100ms$), when it is close but less than the shutdown time ($800ms$), when it is exactly the same ($1000ms$), and when it is close but greater ($1200ms$).

Figure 4 illustrates the results from these experiments, showcasing how the system performed under different traffic intensities and workload complexities. To accurately assess the effectiveness of this solution, it was important to design the tests in a way that allowed for realistic traffic engagement with the application pod. Specifically, the test scenarios were designed to capture two key conditions: some requests should arrive after the shut-off policy's timer has been triggered, requiring the system to perform a cold start, while other requests should arrive before the application pod has been scaled down to zero, avoiding a cold start altogether. This balance ensures that the tests accurately measure the system's ability to handle both immediate and delayed requests.

To simulate these conditions, the think time between successive requests was modeled using an exponential distribution, a common practice in performance evaluation studies. Exponential distributions are frequently employed to simulate user-generated traffic patterns in real-world systems, where the time between requests often follows a stochastic process. By using this approach, we ensured that the request intervals reflected realistic scenarios, capturing a range of think times from short bursts to longer pauses between requests.

The experimental data indicates that the system's response time remains largely stable even as the mean of the exponential distribution increases. This finding suggests that the proposed solution is, on average, effective in mitigating the cold start issue, especially in scenarios where user traffic varies over time. Even with increasing think times—where cold starts would typically introduce significant delays—the results demonstrate that the solution performs consistently well, keeping response times within acceptable limits. This indicates that the proxy container and its associated policies are effective in maintaining a balance between resource efficiency (scaling down to zero when necessary) and minimizing the latency associated with cold starts.

Answer to RQ3: *The data indicates that the response time remains stable even as the mean of this distribution increases, suggesting that this solution is effective, on average, in addressing the cold start issue.*

5.5 RQ4. Does the proposed solution improve resource usage?

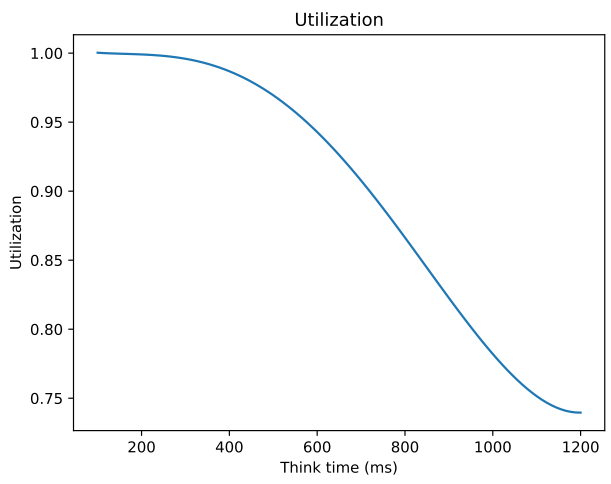

Fig. 5. Proxy CPU consumption.

This experiment uses the same setup to the previous one, focusing on the impact of varying request arrival rates on system resource utilization, particularly CPU consumption. During the evaluation phase, we observed a clear trend: as the mean of the distribution of the arrival rate of requests increases, indicating a longer interval between

successive requests, the average CPU utilization of the system decreases. This trend aligns with the expectations, as longer intervals between requests provide more opportunities for the proxy container to trigger the shutdown of the application container, allowing it to scale down to zero more frequently.
The proxy container's policy, which shuts down the application container after a brief idle period (one second in this case), plays a central role in this behavior. When the time between incoming requests is relatively short, the application container remains active to handle the continuous stream of traffic, resulting in higher CPU utilization. However, as the time interval between requests lengthens, the proxy container can more frequently scale the application container to zero, reducing the overall CPU consumption. This reduction in CPU usage is a direct consequence of the system's ability to temporarily shut down the application container when it is not actively processing requests, thereby conserving computational resources.
Figure 5 illustrates this phenomenon, showing the inverse relationship between the mean arrival rate of requests and the average CPU utilization. As the mean time between requests increases, the CPU usage steadily declines. This means that the system's scaling mechanisms effectively balance resource consumption based on traffic patterns.
This outcome highlights the efficiency of the proposed vertical scaling solution, particularly in handling sporadic or low-frequency traffic patterns. When requests arrive at a slower pace, the system adapts by reducing resource usage without compromising its ability to scale up quickly when new requests arrive. In scenarios where requests are spaced out over longer intervals, the system can leverage the idle periods to scale down, thereby optimizing CPU consumption. Conversely, in high-traffic scenarios with shorter intervals between requests, the system maintains higher resource utilization to meet the demand for continuous processing.

Answer to RQ4: *The proposed solution is capable in reducing the CPU utilization in an effective way, especially when the time between consecutive requests is high.*

6 CONCLUSION

This report presents the VerticalScaleToZero research project, a solution designed to optimize resource management in Kubernetes clusters by allowing the vertical scaling of container resources down to zero. By leveraging this innovative approach, applications can effectively reduce their allocated resources—specifically CPU—during periods of inactivity, all without the need to restart containers. This capability promotes more efficient resource utilization within clusters.

Moreover, this project addresses the prevalent cold-start problem encountered in Kubernetes environments, where scaling resources can often lead to delays and inefficiencies. The proposed solution is efficient, introducing little to none overhead.

6.1 Future work

To gain deeper insights into the performance of the solution presented in this research report, a future task could be designing additional workloads and employing various statistical distributions to generate incoming requests. Additionally, the current proxy container is limited to supporting a single application container; future enhancements could focus on enabling support for multiple application containers within the same pod. Furthermore, an important upgrade for the central controller could involve alterning not only to the deployment process but also to the Kubernetes service configuration, ensuring a seamless integration.