Adaptive Resource Scaling Algorithm for Serverless Computing Applications

Mohammed Ali Awla
Department of Computer Engineering
University of Kurdistan
Email: mohammed.awla@uok.ac.ir

Sadoon Azizi
Department of Computer Engineering
University of Kurdistan
Email: s.azizi@uok.ac.ir

Ayshe Rashidi
ICT Organization
Sanandaj Municipality
Email: a.rashidi@gmail.com

https://doi.org/10.31972/iceit2024.050

Abstract

Serverless computing has transformed cloud-based and event-driven applications by introducing the Function-as-a-Service (FaaS) model. This model offers key benefits, including greater abstraction from underlying infrastructure, simplified management, flexible pay-as-you-go pricing, and automatic scaling and resource optimization. However, managing resources effectively in serverless environments remains challenging due to the inherent variability and unpredictability of workload demands. This paper introduces an Adaptive Resource Scaling Algorithm (ARSA) tailored for serverless applications. ARSA leverages the Auto-Regressive Integrated Moving Average (ARIMA) model to forecast workload demands. Using these predictions alongside a strategy focused on maintaining service quality, ARSA dynamically adjusts the number of container instances needed. The goal is to optimize resource usage while minimizing the occurrence of cold starts. We validated ARSA using a real-world dataset from Microsoft Azure Functions. Our evaluation compared ARSA against fixed instance settings (one, two, and three instances) and the standard Kubernetes Horizontal Pod Auto-scaler (HPA). The results demonstrate that ARSA outperforms these baseline methods by significantly reducing number of cold starts, improving CPU utilization, decreasing memory costs, reducing the number of rejected requests, and enhancing response times. These improvements underscore ARSA's potential in efficiently managing dynamic workloads and enhancing the performance of serverless environments.

Keywords: Serverless Computing, Function as a Service, Resource Provisioning, Cold Start Delay, Auto-Scaling, ARIMA, Workload Prediction.

I. Introduction

With the rise of virtualization technologies, cloud computing has become a key part of our daily activities [1]. In 2014, Amazon introduced serverless computing, which changed the way we think about computational models. This approach helps reduce operational costs and simplifies the complexities involved in system development, making businesses more agile and responsive. By handling tasks like infrastructure provisioning, deployment, and management, cloud providers take on the burdens that companies traditionally faced, allowing them to focus more on their main business goals [2], [3], [4]. Serverless computing also benefits developers by letting them concentrate on building functional logic without worrying about managing the infrastructure. This shift not only streamlines the development process but also speeds up the deployment and scalability of applications, fostering innovation and competitive advantages in businesses.

Function as a Service (FaaS) has gained popularity as an emerging trend in cloud computing, largely due to its event-driven, serverless nature. This model is increasingly used across industries, supported by leading cloud providers like AWS Lambda [5], Google Cloud Functions (GCF) [6], IBM Cloud Functions [7], and Microsoft Azure Functions [8]. The popularity of FaaS reflects the growing demand for solutions that are scalable, flexible, and cost-effective, capable of handling dynamic workloads and changing business needs. The wide availability of FaaS options also indicates the maturity of the serverless computing ecosystem, offering developers a variety of tools and platforms to suit their needs. As more organizations incorporate FaaS into their cloud strategies, the field of cloud computing continues to grow and evolve.

Despite its advantages, serverless computing has some challenges, particularly with cold start delays, which occur when there is a lag in setting up the function's execution environment [9]. While serverless functions can "scale to zero" to save resources when idle, this feature can lead to increased latency when functions are restarted, as seen in cold starts [10], [11]. Research shows that cold starts can sometimes be significantly longer than the actual execution time, up to 59 times in extreme cases [12]. Therefore, addressing cold start delays is crucial for maintaining optimal performance and user satisfaction in serverless systems.

Moreover, serverless computing's automatic scaling can lead to resource shortages during sudden demand spikes, worsening the cold start issue. Some serverless platforms try to address this by capturing hidden resource information to optimize scaling. For example, the Horizontal Pod Auto-scaler (HPA) is a popular tool used in Kubernetes for automatically adjusting resources [13]. Other frameworks, such as Kubeless, use resource data and function settings to better manage resource use and avoid performance bottlenecks during workload changes. Effective scaling strategies are critical for optimizing resources and managing the operational challenges in serverless environments.

In this paper, we propose an **A**daptive **R**esource **S**caling **A**lgorithm, ARSA, using a Time Series-based model called Auto-Regressive Integrated Moving Average (ARIMA). This model predicts workload demands based on historical function invocation data, allowing for proactive adjustments in resource allocation to better handle workload fluctuations. Our approach focuses on maximizing resource efficiency by predicting future needs and making timely scaling adjustments. To test our algorithm, we conducted comparisons with existing methods, including the Horizontal Pod Auto-scaler (HPA) in Kubernetes and a fixed container setup, using real-world data from Microsoft Azure Functions. The results highlight the strengths of our proposed method in managing resource allocation, reducing cold starts, improving CPU utilization, and minimizing rejected requests.

In summary, the key contributions of this paper are:

- We propose a Time Series-based model using ARIMA for predicting future function invocations and scaling resources accordingly.
- We thoroughly evaluate our approach against Kubernetes' HPA and a fixed container setup using a real-world dataset from Microsoft Azure Functions, focusing on metrics like cold start frequency, average CPU usage, and request rejections.
- Our findings demonstrate the effectiveness of our method in optimizing resource management, reducing cold starts, enhancing CPU utilization, and lowering the number of rejected requests.

The rest of this paper is organized as follows: Section II reviews related work, Section III identifies the research gaps, Section IV explains the cold start problem, Section V outlines our proposed method, Section VI discusses the experimental results, and Section VII concludes the paper.

II. Related Work

Previous research has made significant progress in improving serverless computing, especially in tackling the cold start issue [12], [14]. This section reviews various studies that have explored autoscaling techniques to predict the number of pre-warmed containers, helping to reduce cold start delays in serverless environments.

Somma et al. [15] developed and tested a container-based provisioning system for cloud computing. Their approach addresses two key areas: the deployment of containers and the management of container scaling. They introduced a resource management strategy that uses both admission control and autoscaling, driven by a Q-learning algorithm. This algorithm adapts based on the load on physical processors assigned to containers, without depending on the specific computing environment. Similarly, Schuler et al. [16] explored serverless computing and emphasized the need for workload-based autoscaling. They studied how performance changes with different workloads and used a Q-learning model to adjust concurrency limits, improving throughput. Their work also examined how concurrency settings impact performance and how reinforcement learning can effectively adjust these settings.

In [17], Agarwal et al. proposed using reinforcement learning to minimize cold start frequency in serverless systems by analyzing CPU usage and invocation patterns. Their approach showed that it could optimize the number of function instances ahead of time. Benedetti et al. [18] explored a reinforcement-based strategy for autoscaling in OpenFaaS, focusing on edge computing. They demonstrated that their model could learn to adjust scaling policies based on CPU usage, reducing service latency. This reinforcement learning approach allowed the system to autonomously adapt resource allocation in response to changing demands, enhancing the responsiveness and efficiency of serverless deployments in edge environments.

Zafeiropoulos et al. [19] developed autoscaling methods for serverless applications using reinforcement learning techniques. They implemented several RL agents and environments, including Q-learning, DynaQ+, and Deep Q-learning algorithms, to manage dynamic workloads while ensuring Quality of Service (QoS) and efficient resource use. Vahidinia et al. [20] introduced a two-layer adaptive strategy. The first layer uses a holistic reinforcement learning approach to identify function invocation patterns and determine when to keep containers warm. The second layer, based on long short-term memory (LSTM), predicts future invocation times and the required number of pre-warmed containers.

Similarly, Kumari et al. [21] presented an adaptive model that uses a deep neural network and LSTM to forecast the idle container window and assess the need for pre-warmed containers, addressing cold start delays in serverless computing. Their predictive models help allocate resources proactively, reducing latency associated with cold starts. Phung et al. [22] focused on maximizing performance with minimal resource use, aiming to optimize response times and meet QoS requirements. They proposed a strategy to identify scaling policies per pod, using Bi-LSTM for workload forecasting to adjust the number of pods dynamically, making Knative more responsive to workload changes and reducing delay.

In reference [23], Kumari et al. introduced the ACPM framework for dynamic container provisioning during runtime. ACPM operates in two steps: first, it uses an LSTM model to determine the optimal number of containers that need preheating; then, it employs a Docker module to deliver preheated containers quickly, cutting down on cold starts. Suo et al. [24] examined the cold start problem in serverless frameworks and developed HotC, a runtime management system that uses an adaptive live container control algorithm. This algorithm combines exponential smoothing with the

Markov chain method to predict requests and efficiently manage hot containers. Pan et al. [25] proposed a new framework to improve serverless computing efficiency by addressing the cold start issue with a gradient-based pre-warming strategy, along with an automated resource scheduling algorithm for better workflow execution. Finally, Heidari and Azizi [26] presented a serverless architecture that handles infrastructure heterogeneity by dividing the serverless cluster into homogeneous pools, allowing for customized resource allocation and improved load balancing, which optimizes multi-core hardware use and overall system performance.

III. Research Gap

In serverless computing, one of the main challenges is determining the right number of instances, even though progress has been made in reducing cold start latency. Current studies mainly aim at shortening startup times and decreasing the frequency of cold starts, but they often do not effectively decide the optimal number of instances required at any moment. Closing these gaps is vital for improving the autoscaling effectiveness and efficiency in serverless environments.

There are several limitations when using fixed containers and the Horizontal Pod Autoscaler (HPA) for autoscaling. Fixed containers struggle with scalability and often result in overprovisioning because they can't adjust to changing workloads, leading to inefficiencies and higher costs. They also have limited flexibility in deployment and do not adequately tackle the cold start problem, adding to operational challenges. On the other hand, while HPA provides autoscaling features, it can introduce delays in scaling and relies mainly on CPU and memory metrics, which might not fully capture the diverse needs of serverless applications. HPA also struggles with the cold start issue, has complex configurations, and can lead to resource waste since it may not scale down to zero when resources are idle. Additionally, HPA scales at the pod level, which may not be precise enough for the finer scaling required by serverless functions.

To address these issues, an adaptive algorithm is needed that can predict workloads and accurately estimate the required number of instances. This would help balance key factors such as reducing the number of cold starts, improving CPU utilization, and minimizing rejected requests, thereby ensuring efficient use of resources, cost-effectiveness, and optimal performance in serverless computing.

IV. Problem Statement

In serverless computing, the preparation of containers for function execution plays a crucial role in system performance. This process involves either loading function code onto existing containers or using a standard containerization procedure. However, scaling resources down during idle periods and the delay in preparing containers, known as cold start, can create challenges. Although serverless computing's ability to scale down to zero is cost-effective, it also leads to cold start delays.

In practical applications, such as image recognition, initiating functions involves multiple steps like container initialization, resource allocation, function loading, and execution, as illustrated in **Error! Reference source not found.**. Each of these steps can add to the cold start delay, particularly when incoming requests exceed the capacity of available containers. This delay can negatively impact the system's responsiveness and the efficient use of resources. Therefore, reducing cold start times is crucial for improving the performance of serverless computing.

Common approaches to mitigating cold start delays include optimizing how containers are prepared, refining resource allocation methods, and enhancing scaling processes. By reducing the time required to initialize containers and allocate resources, serverless platforms can improve their

responsiveness and better manage varying workloads. This study focuses on reducing cold start times through statistical learning methods, aiming to boost the overall efficiency and responsiveness of serverless computing systems.

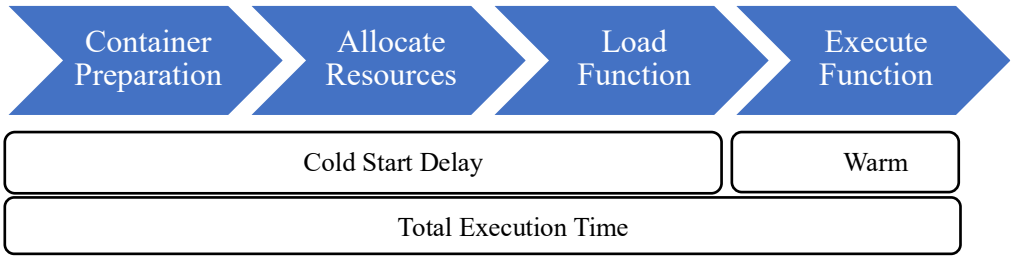

Fig. 1. Serverless function executing steps

V. Proposed Method

In this section, we first provide the background for the ARIMA model, followed by our proposed self-adaptive scaling algorithm for serverless resource provisioning.

A. ARIMA Model

The Auto-Regressive Integrated Moving Average (ARIMA) model is widely used for forecasting time series data in fields like economics, finance, and engineering. This model relies on historical data to predict future trends. It is composed of three main elements:

Auto-Regression (AR): This aspect examines the connections between current observations and those from the past.

Moving Average (MA): This part focuses on the relationship between recent and earlier errors.

Integration (I): This process involves differencing data to achieve stationarity, which means the data's mean and variance remain consistent over time.

ARIMA is useful for analyzing time series data that shows trends, seasonality, or complex patterns, allowing for both short-term and long-term forecasts based on data properties. It can also handle and predict multiple time series concurrently, making it adaptable and easy to use with various data types. Its capability to understand complex patterns and relationships makes it particularly effective for predicting workloads in serverless environments, as shown in the following equation:

$$X(\mathrm{t}) = \mathrm{C} + \sum_{i=1}^{p} \phi_i X(\mathrm{t}-\mathrm{i}) + \sum_{j=1}^{q} \theta_i \varepsilon(\mathrm{t}-\mathrm{j}) + \varepsilon(t)$$

where:

- C is the constant term,
- $X(t)$ represents the observed value at time t,
- ϕ_i denotes the coefficients for the autoregressive terms,
- θ_i denotes the coefficients for the moving average terms,

- $\varepsilon(t)$ represents the error term at time t.

B. Proposed adaptive resource scaling algorithm (ARSA)

The Adaptive Resource Scaling Algorithm (ARSA) is designed to dynamically manage the number of active container instances in serverless computing environments. By using the ARIMA model to forecast workload demands, ARSA adjusts resources ahead of time, reducing cold start delays and improving performance. It calculates the required number of instances for each time slot based on predicted workload, average execution time, and failure rate.

Unlike traditional scaling methods that rely on metrics like CPU or memory usage, ARSA employs a predictive approach using ARIMA model forecasts and failure rates. This involves creating a custom auto-scaler, implemented as a Python script, that integrates ARIMA predictions. Through simple calculations, ARSA adjusts the number of active containers to match forecasted workloads, preparing the environment and containers ahead of demand surges. By focusing on predicted needs rather than reactive measures, ARSA aims to streamline resource management and improve responsiveness in serverless computing.

Algorithm 1. *Adaptive Resource Scaling Algorithm (ARSA)*

```
Initialize:
    num_instances = 1;
    interval = 10 minutes;
Repeat Every 10 minutes:
    try:
        Get latest workload prediction using ARIMA model (pred)
        Calculate average execution time from start to now (avg_exe_time)
        Calculate failure rate in past 10 minutes (failure_rate)
        num_instance = math.ceil((pred * avg_exe_time) / interval);
        if failure_rate >= Δ: # Δ is the threshold for failure rate
            numb_instance += 1;
    except Exception as e:
        Log error message: "Error occurred: {e}"
        numb_instance = 1; # Reset numb_instance to the default value
    Schedule next iteration after 10 minutes
End.
```

The pseudocode for the proposed algorithm is presented in **Algorithm 1**. Below, we provide a detailed description of the algorithm.

Initialization:

The algorithm starts by setting the initial instance count to 1 and defines a 10-minute interval for checking workload.

Workload Prediction:

Every 10 minutes, the algorithm uses the ARIMA model to forecast future workload demands (denoted as $pred$). It also calculates the average execution time (avg_exe_time) from the beginning of the monitoring period up to the current time. Additionally, it monitors the failure rate over the last 10 minutes to evaluate system performance.

Instance Adjustment:

The algorithm calculates the required number of instances using the formula:

$$num\_instance = math.ceil((pred * avg\_exec\_time) / interval)$$

If the failure rate in the past 10 minutes exceeds or equals Δ, indicating potential performance issues, the algorithm increments the number of instances by 1 to mitigate workload bottlenecks. It is worth mentioning that is this work, we set Δ to 3% based on empirical observations.

Error Handling:

The algorithm employs a try-except block to handle potential errors or exceptions during workload prediction and calculation processes. In case of an error, the algorithm logs the issue and resets the number of instances to its default value 1 to ensure continuous operation.

In summary, the ARSA improves serverless workload management by adjusting resource allocation dynamically. It uses workload predictions, execution times, and system stability data to optimize resource use and ensure responsive performance in serverless environments.

VI. Experimental Results

In this section, we will first introduce the dataset used for our evaluation. Next, we will assess the accuracy of predictions made using the ARIMA model. Following that, we will describe the other algorithms considered in our comparison and the performance metrics employed. Finally, we will present the experimental results obtained.

A. Dataset

To evaluate the effectiveness of our approach, we utilized an open-source dataset provided by Microsoft Azure Functions [27]. This dataset comprises authentic invocation traces collected from real-world scenarios, capturing minute-by-minute invocation counts over a 14-day period. The initial 10 days of data were designated for pattern modeling, serving as the training phase for our evaluation. The remaining four days were dedicated to simulation activities. Each day contains 144 data points, representing the average number of requests per 10 minutes over a 24-hour period. This granular level of detail allows for a comprehensive assessment of our proposed methodology's performance. Fig. 2 illustrates the dataset.

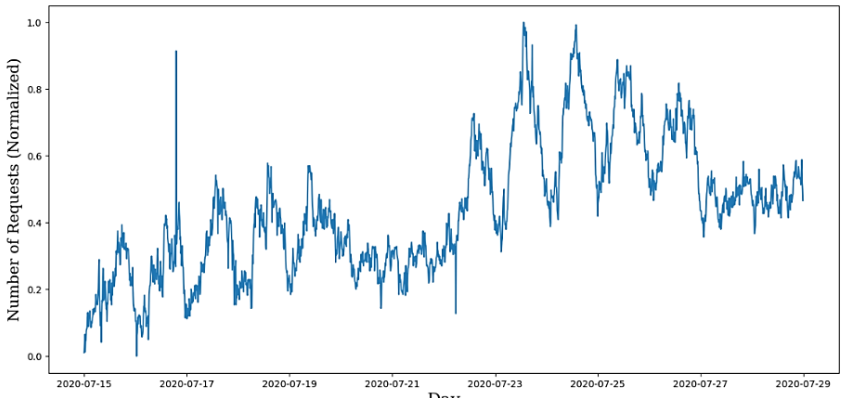

Fig. 2. Real-world dataset provided by Microsoft Azure Functions

B. Actual Data Versus Predicted Data

To evaluate the ARIMA model's predictive accuracy, we used it to forecast future invocation counts based on the training data. Fig. 3 compares the observed values with the predicted values. As shown, the ARIMA model's predictions closely match the actual observed data, indicating high accuracy. The model effectively identifies the underlying patterns and trends within the dataset, confirming its reliability for predicting serverless workload demands. This accuracy is crucial for the success of our proposed adaptive resource scaling algorithm, as it ensures accurate prediction of future workloads and optimal resource allocation.

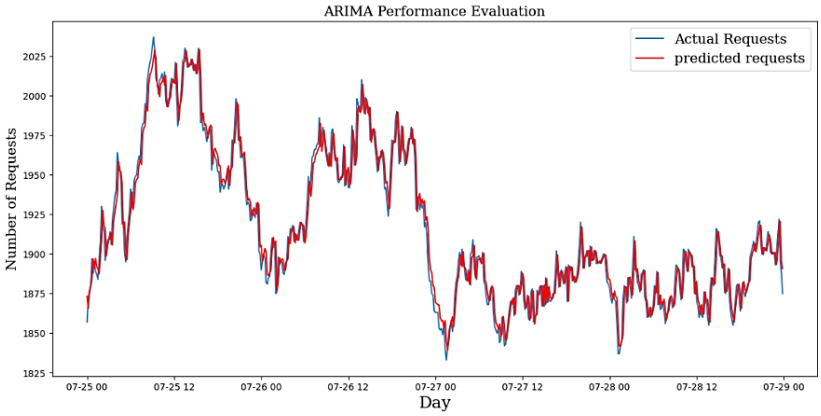

Fig. 3. Actual Data Versus Predicted Data

C. Compared Algorithms

We compared our proposed algorithm, termed "Adaptive Resource Scaling Algorithm (ARSA)," with several fixed instance configurations and the default HPA algorithm in Kubernetes. The comparison involved the following configurations:

- **Fixed Instance Configurations:** We tested configurations with 1, 2, and 3 fixed instances separately to evaluate their performance under varying workloads.
- **Horizontal Pod Auto-scaler (HPA):** We also compared our approach with the HPA algorithm, which requires specific configuration settings. For this study, the HPA was configured with a minimum of 1 instance and a maximum of 10 instances. The target average utilization was set to 18% based on our dataset analysis. This configuration was chosen to optimize the balance between the number of rejected requests and the incidence of cold starts.

D. Metrics

Number of Cold Starts: We measured the number of times new instances were initialized. Each instance initialization is considered a cold start.

Average CPU Utilization: We evaluated the average percentage of CPU capacity used across different scaling strategies. This metric assesses resource allocation efficiency and performance.

Memory Usage Cost: We calculated the cost associated with memory usage based on a rate of $0.005 per megabit per second (MBps). This analysis provides insights into the financial impact of different scaling strategies on memory resource utilization.

Number of Rejected Requests: We examined the number of requests that were rejected due to missed deadlines. This metric reflects the system's ability to handle requests within required time constraints and indicates performance and reliability.

Average Response Time: We evaluated the average time it took the system to process and respond to requests. This metric is crucial for understanding the system's scalability and efficiency under varying loads.

E. Results

This section presents and analyzes the simulation results. Fig. 4 displays the number of cold starts encountered by each algorithm. Fixed instance setups, which do not adjust to workload variations, lead to a constant cold start rate, which can cause inefficiencies in service quality and resource use. On the other hand, the ARSA algorithm, which uses predictive techniques, reduces cold starts to five, which is two fewer than the HPA, the default auto-scaler in Kubernetes. This reduction is attributed to ARSA's proactive adjustment of resources based on predicted workload demands, improving performance and responsiveness in serverless computing environments.

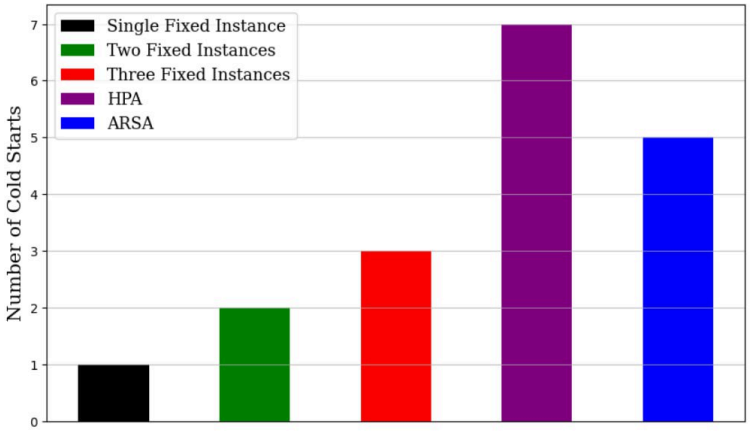

Fig. 4. Comparison algorithms in terms of Number of cold starts

Fig. 5 illustrates the results in terms of the average CPU utilization. The results indicate that the ARSA algorithm achieves a CPU utilization of 21%, closely aligning with the Single Fixed Instance configuration. This demonstrates that ARSA maintains high CPU efficiency while adapting to workload demands. In contrast, the fixed instance configurations show decreasing CPU utilization rates as the number of instances increases, resulting in less efficient resource usage. Our proposed ARSA algorithm improves CPU efficiency by approximately 28% compared to the HPA, which

achieves 16% utilization. This improvement underscores ARSA's effectiveness in optimizing resource allocation and enhancing performance through its predictive capabilities.

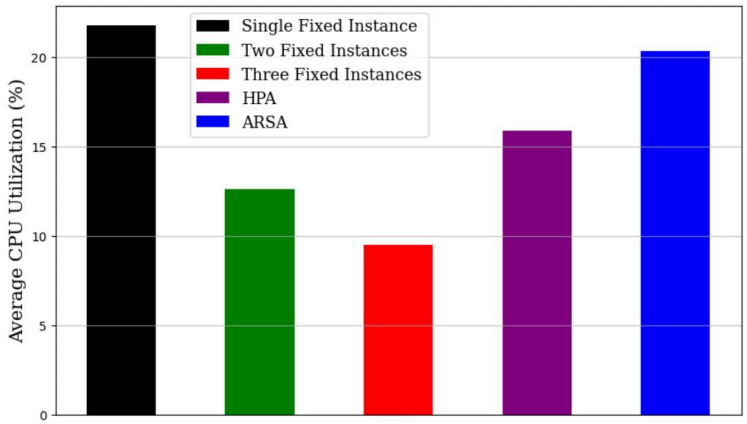

Fig. 5. Comparison algorithms in terms of average CPU utilization

The simulation results for the memory usage costs are shown in Fig. 6. The single fixed instance configuration has the lowest cost due to its minimal memory usage. ARSA achieves the second-lowest cost, demonstrating its cost-effectiveness and efficient memory resource utilization. HPA ranks third, while the configurations with two and three fixed instances have the highest costs. This is due to excessive memory consumption and inefficient resource utilization, as increasing the number of instances leads to higher memory costs. Our proposed ARSA algorithm is 6.8% more cost-effective than HPA, highlighting its ability to optimize memory usage and reduce costs.

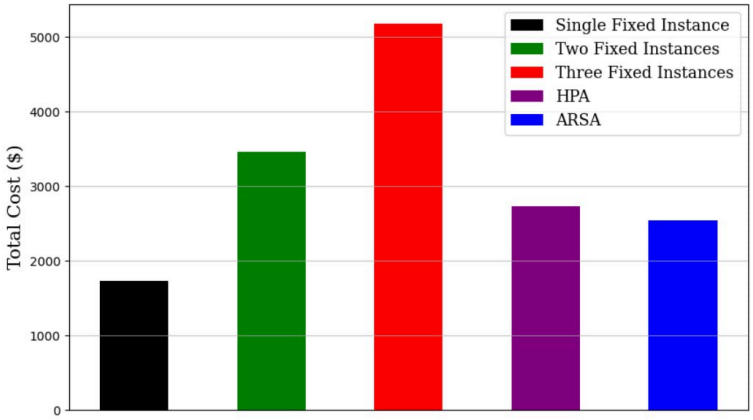

Fig. 6. Comparison algorithms in terms of total memory usage cost

Table 1 compares the number of rejected requests under different scaling strategies. The results clearly demonstrate the superior performance of the ARSA algorithm. Out of 1,103,680 total requests, ARSA only experiences 237 rejections due to missed deadlines. This is significantly lower than the rejection rates observed with HPA and the configurations with one or two fixed instances. The one and two fixed instance configurations have higher rejection rates due to their inability to adapt to changing workloads, resulting in missed deadlines. While the three fixed instance configuration performs slightly better in terms of rejections, it is not workload-aware and lacks the efficiency of adaptive scaling.

Table 1. Number of rejected requests

	Single Fixed Instance	**Two Fixed Instances**	**Three Fixed Instances**	**HPA**	**ARSA**

No. of Rejected Requests	180,222 (16.32%)	80,153 (7.26%)	5 (0.00%)	11,627 (1.05%)	237 (0.02%)

Fig. 7 shows the average response time for requests. The results show that the ARSA algorithm achieves response times similar to the three fixed instance configurations. In contrast, the response times for one and two fixed instances are significantly slower, leading to lower performance due to their inability to adapt to changing workloads. Furthermore, ARSA outperforms the HPA algorithm by 3%, highlighting its effectiveness in improving system responsiveness. This improvement is attributed to ARSA's ability to predict workload demands more accurately, enabling more efficient resource allocation.

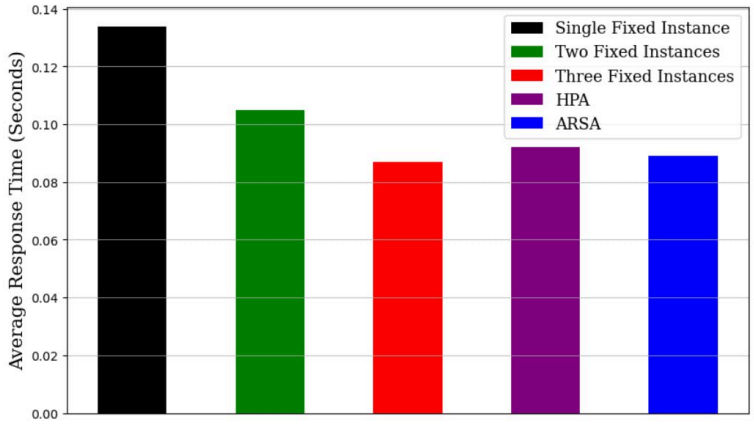

Fig. 7. Comparison algorithms in terms of average request response time

VII. Conclusion

Resource scaling in serverless environments is a major problem to tackle, mostly due to how workloads can be unpredictable. In this work, we addressed this challenge by proposing a self-adaptive approach for scaling serverless resources. Our proposed algorithm makes use of the ARIMA model for workload demand forecasting. Accordingly, we developed a service quality-based approach to address the problem of how many container instances should be in active state given the provisions of the forecaster model. As a case study, we used an industrial open-source dataset provided by Microsoft Azure Functions. Our proposed algorithm ARSA was compared with a single, two and three static instances and the default Horizontal Pod Auto-scaler (HPA) in Kubernetes. Cold start occurrences, average CPU usage, cost of memory expenditure, number of denied requests, and average response time were all included as evaluation metrics. The evaluation demonstrates that ARSA has a significant improvement in workload demand prediction, and therefore the number of instances needed, when compared to the baseline algorithms. This improvement in performance metrics helps to achieve lesser cold starts, higher CPU utilization, lower memory usage cost, reduced number of rejected requests, and better average response times particularly presents ARSA has prospects of being an enhancement to resource utilization and response optimization. This statement is based on the positive achievements offered by the ARSA approach.

VIII. Acknowledgment

We would like to disclose the assistance of any AI tool used for the purpose of initial drafting and language refining of this manuscript, ChatGPT, as an exception. Editing and paraphrasing were

done using this tool. The authors prepared further edits to the documents in order to maintain their academic integrity as well as to meet the requirements of the conference.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009, doi: https://doi.org/10.1016/j.future.2008.12.001.

[2] N. Kratzke, "A brief history of cloud application architectures," 2018. doi: 10.3390/app8081368.

[3] I. Baldini *et al.*, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds., Singapore: Springer Singapore, 2017, pp. 1–20. doi: 10.1007/978-981-10-5026-8_1.

[4] M. Sewak and S. Singh, "Winning in the Era of Serverless Computing and Function as a Service," in *2018 3rd International Conference for Convergence in Technology (I2CT)*, 2018, pp. 1–5. doi: 10.1109/I2CT.2018.8529465.

[5] "Amazon. AWS Lambda." Accessed: Aug. 25, 2024. [Online]. Available: https://aws.amazon.com/lambda/

[6] "Google. Google Cloud Functions." Accessed: Aug. 25, 2024. [Online]. Available: https://cloud.google.com/functions

[7] "IBM. IBM Cloud Functions." Accessed: Aug. 25, 2024. [Online]. Available: https://cloud.ibm.com/functions/

[8] "Microsoft. Azure Functions." Accessed: Aug. 25, 2024. [Online]. Available: https://azure.microsoft.com/en-in/products/functions/

[9] M. S. Aslanpour *et al.*, "Serverless Edge Computing: Vision and Challenges," in *ACM International Conference Proceeding Series*, 2021. doi: 10.1145/3437378.3444367.

[10] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *J Grid Comput*, vol. 12, no. 4, 2014, doi: 10.1007/s10723-014-9314-7.

[11] S. Verma and A. Bala, "Auto-scaling techniques for IoT-based cloud applications: a review," *Cluster Comput*, vol. 24, no. 3, 2021, doi: 10.1007/s10586-021-03265-9.

[12] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies," in *2020 International Conference on Omni-Layer Intelligent Systems, COINS 2020*, 2020. doi: 10.1109/COINS49042.2020.9191377.

[13] "Kubernetes." Accessed: Aug. 25, 2024. [Online]. Available: https://kubernetes.io/docs/home/

[14] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions," *ArXiv*, vol. abs/2310.08437, 2023.

[15] G. Somma, C. Ayimba, P. Casari, S. Pietro Romano, and V. Mancuso, "When less is more: Core-restricted container provisioning for serverless computing," in *IEEE INFOCOM 2020 -*

IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS 2020, 2020. doi: 10.1109/INFOCOMWKSHPS50562.2020.9162876.

[16] L. Schuler, S. Jamil, and N. Kuhl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, 2021. doi: 10.1109/CCGrid51090.2021.00098.

[17] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, 2021. doi: 10.1109/CCGrid51090.2021.00097.

[18] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement Learning Applicability for Resource-Based Auto-scaling in Serverless Edge Applications," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events, PerCom Workshops 2022*, 2022. doi: 10.1109/PerComWorkshops53856.2022.9767437.

[19] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," *Simul Model Pract Theory*, vol. 116, 2022, doi: 10.1016/j.simpat.2021.102461.

[20] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach," *IEEE Internet Things J*, vol. 10, no. 5, 2023, doi: 10.1109/JIOT.2022.3165127.

[21] A. Kumari, B. Sahoo, and R. K. Behera, "Mitigating Cold-Start Delay using Warm-Start Containers in Serverless Platform," in *INDICON 2022 - 2022 IEEE 19th India Council International Conference*, 2022. doi: 10.1109/INDICON56171.2022.10040220.

[22] H. D. Phung and Y. Kim, "A Prediction based Autoscaling in Serverless Computing," in *International Conference on ICT Convergence*, 2022. doi: 10.1109/ICTC55196.2022.9952609.

[23] A. Kumari and B. Sahoo, "ACPM: adaptive container provisioning model to mitigate serverless cold-start," *Cluster Comput*, vol. 27, no. 2, 2024, doi: 10.1007/s10586-023-04016-8.

[24] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2021. doi: 10.1109/Cluster48925.2021.00018.

[25] S. Pan, H. Zhao, Z. Cai, D. Li, R. Ma, and H. Guan, "Sustainable Serverless Computing With Cold-Start Optimization and Automatic Workflow Resource Scheduling," *IEEE Transactions on Sustainable Computing*, vol. 9, no. 3, 2024, doi: 10.1109/TSUSC.2023.3311197.

[26] S. Heidari and S. Azizi, "Heterogeneity-aware Load Balancing in Serverless Computing Environments," in *7th International Conference on Internet of Things and Applications, IoT 2023*, 2023. doi: 10.1109/IoT60973.2023.10365354.

[27] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020*, 2020.