

# A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform

Akash Puliyadi Jegannathan<sup>§</sup>, Rounak Saha<sup>§</sup> and Sourav Kanti Addya

Department of Computer Science and Engineering

National Institute of Technology Karnataka, Surathkal, India

Email: {pjakash10,rounak.rs69}@gmail.com, souravkaddya@nitk.edu.in

**Abstract**—Serverless computing is a buzzword that is being used commonly in the world of technology and among developers and businesses. Using the Function-as-a-Service (FaaS) model of serverless, one can easily deploy their applications to the cloud and go live in a matter of days, it facilitates the developers to focus on their core business logic and the backend process such as managing the infrastructure, scaling of the application, updation of software and other dependencies is handled by the Cloud Service Provider. One of the features of serverless computing is ability to scale the containers to zero, which results in a problem called cold start. The challenging part is to reduce the cold start latency without the consumption of extra resources. In this paper, we use SARIMA (Seasonal Auto Regressive Integrated Moving Average), one of the classical time series forecasting models to predict the time at which the incoming request comes, and accordingly increase or decrease the amount of required containers to minimize the resource wastage, thus reducing the function launching time. Finally, we implement PBA (Prediction Based Autoscaler) and compare it with the default HPA (Horizontal Pod Autoscaler), which comes inbuilt with kubernetes. The results showed that PBA performs fairly better than the default HPA, while reducing the wastage of resources.

**Index Terms**—Serverless computing, cold start, cloud computing, function launching, SARIMA.

## I. INTRODUCTION

Serverless Computing is a state-of-the-art Cloud Computing paradigm that engrosses tremendous attention from industry and academia due to its architectonics. Serverless Computing does not mean that there are no servers; it merely means that the servers are there, but the user does not have the hassle to manage those [1]. In Serverless computing, applications or microservices are deployed as functions that start working when any event has triggered it. During the function life-cycle, all the operational overheads are managed by the cloud service provider (CSP). Deploying multi-tier application [2] become easier for CSP using the concept of inter dependent function in serverless computing.

Serverless Computing comes with many advantages. Significant of them are no back-end operational overheads means the CSP does all back-end-related stuff such as server management, auto-scaling, load-balancing and others so that the users focus only on developing the business logic and granular

on-demand pricing in order of milliseconds, which means the execution cost of any function charged on the millisecond scale [3]. Due to this flexibility in service-deployment, there is a prolific shift of users towards serverless computing seen in the recent times, especially for applications such as deep learning [4], blockchain [5], Internet of Things [6], big data analytics [7], and others [8], [9]. Therefore, major cloud service providers such as AWS, Microsoft Azure, Google Cloud, and IBM Cloud came up with their serverless platform named AWS Lambda<sup>1</sup>, Azure Function<sup>2</sup>, GCP Function<sup>3</sup>, and IBM Function<sup>4</sup> to gratify this large faction, respectively.

However, despite having much prosperity, Serverless Computing has several drawbacks that deteriorate its performance at an extensive level; the Cold-start problem is one of the major of them. The definition of the Cold-start problem can be derived from the function life-cycle. In serverless, the function life cycle starts when an event triggers it; after that, containers are created according to the function's requirements, then the environment and network are set up according to the functions image file finally, the source code files of the function is loaded, and the function starts execution. The phase that starts after triggering the function until before execution is called containerization [10] and the time required for it is majorly called cold-start time. The problem of minimizing this cold-start time is called as the cold-start problem. A recent study done by [11] showcases that in severe circumstances, the cold-start time can go up to 59 times higher than the actual time of execution. These circumstances occurred for many reasons; the unavailability of computing resources is one of the major of them [12]. In serverless, auto-scaling of resources is one of the significant features that when the incoming web traffic load is getting low, resources are scaled down according to the need. However, this is the leading cause of the unavailability of computing resources. Whenever resource requirements increase rapidly w.r.t actual scaling of resources, lesser resources need to handle heavy workloads

<sup>1</sup><https://aws.amazon.com/lambda/>

<sup>2</sup><https://azure.microsoft.com/en-in/services/functions/>

<sup>3</sup><https://cloud.google.com/functions>

<sup>4</sup><https://cloud.ibm.com/functions>

<sup>§</sup> Equal contribution

hence containerization process gets prolonged; hence cold-start problem gets severe. Various serverless platforms offer to capture the concealed computing resource information, and some open-source serverless frameworks use this to take advantage of anticipation resource scaling. The most used and optimized resource auto-scaling is Horizontal Pod Autoscaler (HPA) which is provided by Kubernetes [13] primitive serverless framework Kubeless. Kubeless uses underlying resource information and function configuration knowledge for resource scaling decision-making.

According to the earlier observations, the proper accessibility of resources could efficiently minimize the cold-start problem. In this work, we propose a Time Series based learning model using Seasonal Auto-Regressive Integrated Moving Average (SARIMA). This model analyses previous functions invoking data patterns. Based on that, it predicts the future incoming function invocation request and scales up/down resources correspondingly. This model also utilizes resource usage efficiently. We compare the proposed model with HPA provided by Kubernetes, which is the best in class auto-scaler available. This helps in analyzing and examining the performance of both configurations.

The **key contributions** of our work are as follows:

- A learning-based time series forecasting model named *Prediction Based Autoscaler* (PBA) that predicts the number of future function invocations and allocates computing resources accordingly, making it possible to spin up the containers before high function invocation traffic. This also helps to reduce the number of resources that would have gone waste if there were no scaling requirements.
- Comparative analysis of our proposed model against HPA comes inbuilt with Kubernetes for the synthetic function workload pattern.

The paper is composed of various sections. Section II contains related work. The problem of cold start is explained in section III. Section IV describes the strategy that will be used to mitigate the cold start latency. The Experimental setup along with the results and analysis will be discussed in Section V. We conclude the paper in Section VI.

## II. RELATED WORK

Serverless is a state-of-the-art field cloud computing paradigm, which has gotten tremendous attention in the last few years from the industry as well as academia. However, due to its short journey, not much literature is available that tackles the problem like cold-start. Even though organizations rapidly migrate their applications to the cloud, the cold start problem persists. McGrath [14] and Wang [15] measured cold start latency incurred on various serverless platforms and compared them. The requests were sent in an increasing interval, from one minute to thirty minutes. It was shown that if the interval between two function calls were more than 5 minutes, it resulted in an apparent spike in the latency. Also, it

was shown that cold start latency differed from one platform to another. After conducting more rigorous experiments on different platforms, Wang et al. [15] showed that the median warm start latency in Google is more than three times that in AWS, and Azure is almost thirteen times that of AWS, and the median cold start latency of AWS is more than ten times of its warm start, in Azure, it is 11.37 times. It is evident that the time required to handle a request is more for the cold start than the warm start. Some of the authors came up with innovative approaches to tackle the problem.

Some of the authors came up with innovative approaches to tackle the problem. Slacker [16] identifies critical packages while launching a container. By prioritizing these packages and lazily loading others, it can reduce the container startup latency. Gias and Casale [17] found out that the problem of cold start is very similar to the enhancement in the cache hit ratio. By applying the same method as in the cache hit ratio to the Faas architecture, it was seen that most of the resources were not being used. They proposed COCOA, a queuing-based approach to evaluate the effect of cold start on response times. Depending on the state of the function, variable values of memory function were taken into account. They developed a simulator, which showed that COCOA could cut down on overprovisioning by almost seventy per cent in some workloads while following the SLAs.

McGrath et al. [14] proposed a scheme where there would be workers in warm and cold queues wherein old container scans are reutilized, and new ones can be created. The drawback here is that each function is mapped to only one single container. Therefore, it is not an easy task to reduce the latency caused by cold start and improve the performance of the cloud computing platform. Ghosh et al. [18] looked at various serverless architectures and evaluated them using Lambda functions in AWS and compared the performance and the latency incurred in it with a traditional virtual machine (VM) based approach. It was seen that the time taken to access and query the database in serverless is nearly fourteen times that setup based on VM. With the help of some caching techniques, they reduced the response latency, which considerably improved the overall response time. Daw et al. [19] tried to remove the problem of cascading cold starts, in which they introduced a novel approach Xanadu, where the resources are provisioned to the serverless platforms by speculating it beforehand. Even in standalone correlated functions, Xanadu minimizes overhead latencies when used with the implicit chain detection technique. When compared to Knative and OpenWhisk, Xanadu offers considerable performance gains, restricting cascading cold starts to a single event and minimizing the cost overruns that come with resource pre-deployments.

## III. PROBLEM DESCRIPTION

In Serverless Computing, before the actual execution, functions are containerized. This containerization process is done in two ways; first, source codes of the function are loaded

onto the readily available container, and second is if readily available containers are not available regular containerization procedure is followed, which includes container creation, dependency loading, environment setup, network creation, and source code loading.

To prepare or store containers, computing resources are required. In serverless, if there is no incoming request for a specific time, readily available containers get deleted, and resources that handle those containers get deallocated, also known as resource scale down. Suppose a certain number function gets invoked in this situation due to the unavailability of required computing resources. In that case, the containerization process will take much more time, which will cause cold start. In another case, if the number of function invocations is more than the resources can handle, it will also cause a higher time to prepare the containers, causing cold start. Let us assume the set of upcoming functions,  $F$ , and the average execution time of  $i^{th}$  function is  $t_i$  and then the function with the added latency  $T_i$  of cold start will as per 1.

$$T_i^{Cold} = t_i + T_i \quad (1)$$

Through this paper we aim to minimize the cold start time  $T_i^{Cold}$  as much as possible using learning based statistical methods.

#### IV. PREDICTING THE FUTURE LOAD

In order to mitigate the cold start time, the proposed model predicts the probable future incoming function requests; this enables the model to determine how much computing resources are needed to be provisioned. The model will predict the number of upcoming future requests by using the statistical time-series model named Seasonal Autoregressive Integrated Moving Average, also known as *SARIMA* [20]. This statistical model works better when the prediction period is on a day-to-day basis and reactive to instantaneous changes, making this model a better selection over deep learning models, which are also computationally intensive concerning the used technique.

##### A. SARIMA Model

A SARIMA model or a Seasonal ARIMA model is given by seven whole parameters, the first three are  $p$ ,  $d$  and  $q$ .  $p$  corresponds to the order of the AR piece, the  $d$  to the order of the integrated piece or how many times we take a difference to make it stationary and  $q$  to the order of the MA piece. The next three are kind of just analogues of the first three except in a seasonal context.  $m$  is the seasonal factor, which is the number of periods within a year it takes for the seasonality to repeat.

After combining all these parameters, the notation for SARIMA can be represented as:

***SARIMA(data, order=(p,d,q), seasonal\_order=(P,D,Q,m))***

In order to understand the (SARIMA) model [21], let's first break ARIMA into three parts. AR (denoted by  $p$ ) is autoregressive, which means we want to evaluate the value

of our time series today, based on the value of the time series some periods in the past. *I* (denoted by  $q$ ) stands for integrated which means the time series might have some sort of upwards or downwards trend so we use differencing to make it stationary. *MA* (denoted by  $d$ ) or Moving Average means that we are taking errors from a previous time period and apply that information to the next prediction. At last, the *S* stands for seasonality, which means there might be a repeating pattern in a day, week, month, year etc that happens over and over and over again over time. For example, ***SARIMA(3,0,0)(3,0,2)24***

The parameters  $P$ ,  $D$ , and  $Q$  change with respect to seasonality  $m$ . For example, an  $m$  of 24 for hourly data suggests a seasonal cycle on a per day basis. A  $P=3$ , would use the last three seasonally offset observations in the model  $t-(m * 1)$ ,  $t-(m * 2)$ ,  $t-(m * 3)$ . A  $D$  of 0 would not take any seasonal differencing into considerations and a  $Q=2$  would use errors of the second order in the model (e.g. moving average).

The parameters can be picked by analysing Auto Correlation Function (ACF) and Partial Auto Correlation Function (PACF) plots by examining some of the recent time series values. These plots can also be used to get the values of seasonal components (P, D, Q) by looking at correlation at seasonal lag time steps.

#### V. EXPERIMENTAL RESULTS AND ANALYSIS

In this section we have verify the proposed method through simulated experiment.

##### A. Experimental Setup

1) *System Configuration*: In order to evaluate the model and test the PBA, we create an experiment setup on top of Docker and Kubernetes on a physical node. The system configuration can be found in Table I.

TABLE I  
SYSTEM CONFIGURATION

Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Memory	8018MB
Operating System	Ubuntu 20.04.2 LTS
Kernel	Linux 5.4.0-73-generic (x86_64)
Kubernetes	version=v1.18.4
Docker	version=20.10.3

2) *Dataset*: Instead of using a real world data set, we have used a data set generator, which produces random data for the average number of request in an hour for any number of days. In our experiment we generated a data set for 10 days, each day containing 24 values, which denotes the average number of requests per hour for 24 hours. Figure 1 shows the data set taken into consideration.

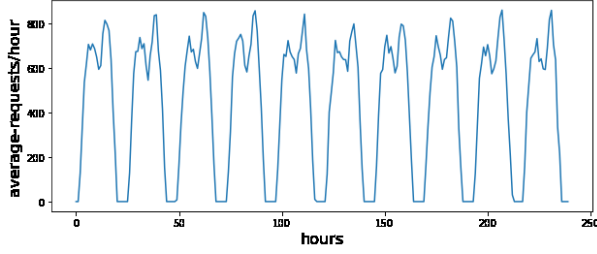


Fig. 1. Dataset

3) *Horizontal Pod Autoscaler (HPA)*: Kubernetes provides a default autoscaler which is called the Horizontal Pod Autoscaler (HPA), which scales up or scale down the pods with respect to the CPU and memory usage. This default autoscaler can be configured to an extent such that it can handle the load reasonably well. By tweaking and testing some of the parameters, HPA can give really good results and can be applied on most of the normal application. But if the application needs to be highly responsive or gets huge amount of requests in a very short period of time, then the HPA might not be able to handle it very well. There is also the case of idle time, where all the functions and containers are deleted due to no incoming requests, and are needed to be deployed again. The default HPA will almost always incur an added latency due to the default behaviour of cloud platforms of scaling the application to zero when not in use. Figure 2 shows the HPA architecture.

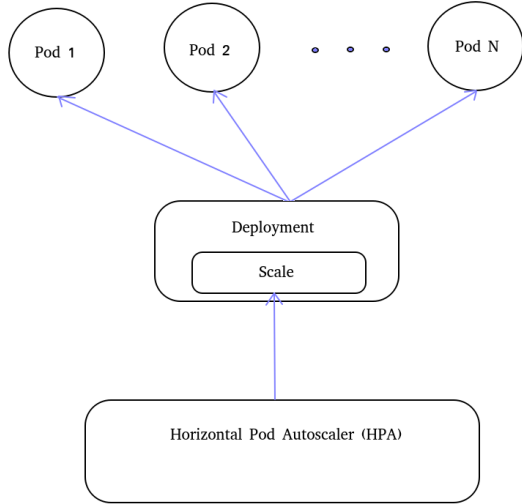


Fig. 2. HPA architecture.

4) *Prediction Based Autoscaler (PBA)*: Instead of using the deciding metric for scaling as CPU or memory usage, we rely only on the prediction made by the SARIMA model and scale

up or scale down in accordance with the predicted values. With the help of Kubernetes API, we propose a custom autoscaler known as the Prediction Based Autoscaler (PBA). PBA is a simple python script fed with the predicted values obtained from the SARIMA model. After some experiments, it was evident that a pod can only handle a certain number of requests for a particular set of configurations that the pod is allowed to have. Also, the amount of time required to create one more replica took around 30s. Therefore, if we knew the number of incoming requests beforehand, then with the help of some simple arithmetic, we could easily set up the required number of containers just in time to serve the incoming requests; this will potentially prevent the cold start since we have set up the environment and containers and done all the pre-processing required to serve the incoming requests.

---

**Algorithm 1** Prediction Based Autoscaler (PBA)

---

```
Initialize: first = True, reqPerPod = int(sys.argv[1]),
initialDelay = int(sys.argv[2]), interval = int(sys.argv[3])
for prediction in pred: do
    if condition then
        | ct = math.ceil(prediction/reqPerPod)    ft =
        | math.ceil(prediction/reqPerPod)
    else
        | ft = math.ceil(prediction/reqPerPod)
    end
    try
        scale up the number of pods with max(ft, ct)
        ct = math.ceil(prediction/reqPerPod)
    except
        print("error")
        time.sleep(interval - (initialDelay if first else 0))
        first = False
```

**end**

---

5) *Application Deployment and Monitoring*: For the purpose of testing our strategy, we created a simple *nodejs* application with 3 endpoints, one static end point and two other endpoints, out of which one writes to the database and the other reads from the database. We have used *redis* as our database so that there is minimal interference due to the network latency, and we can see the actual response time of the requests that are being served. We create *.yaml* files for both *nodejs* and *redis-db* and deploy it on the *Kubernetes cluster* with the help of *kubectrl*, which converts the configuration written in the *.yaml* files to JASON format and makes the API request to the Kubernetes. All of the deployments pertaining to the application were in the default namespace.

In the monitoring namespace, all the components required for the purpose of monitoring the metrics generated from the application and the cluster itself are deployed. The two main monitoring deployments were Prometheus<sup>5</sup> and

<sup>5</sup><https://prometheus.io/>

Grafana<sup>6</sup>. *Prometheus* maintains a time series database that stores all the metrics data like current CPU usage, number of incoming requests, response latency etc. *Grafana* fetches the values from the time series database produced by *Prometheus* and makes it easy to visualize it through very cool looking graphs, gauges, charts etc in a web browser. All of the manifest can be found in the [22].

6) *k6* - load generator: *k6*<sup>7</sup> is an open source load testing tool used to measure the performance of an application under varied amount of loads. The two common use cases of *k6* is load testing and performance monitoring. There are various kinds of load testing that can be done upon a website or microservice like spike, stress and soak test. We could monitor the performance of our production environment when put under various test. The *k6* load generator was also deployed on the default namespace in the *Kubernetes* cluster along with the node application and *redis-db*.

## B. Experimental Results

1) *Actual vs. Predicted Values*: In order to predict the future values, we used the SARIMA model. Figure 3 shows the comparison between the actual and the predicted values. It can be seen that the prediction is quite accurate and close to the actual values.

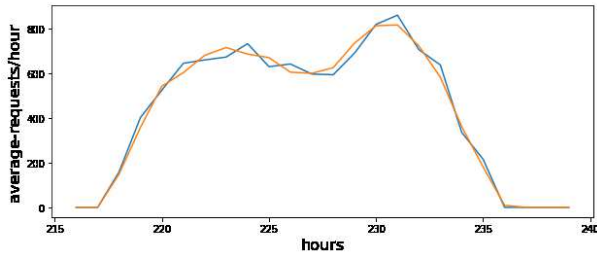


Fig. 3. Predicted *vs.* Actual data.

2) *Average Response Time per Second*: In order to evaluate our strategy of using a classic statistical model like SARIMA for predicting the incoming requests along with PBA, we compare it with the default autoscaler of *Kubernetes*, HPA. Firstly, we show the average response time taken to execute a request. The number of requests sent to the node are proportional to the number of requests an Indian payments service company gets during the 24 hours of the day. The results can be seen in Figure 4, depicting a *Grafana* dashboard that compares the *requests/sec* along with the average *response time/sec* of both PBA and HPA.

3) *Pod Count Comparison*: Upon conducting the experiment numerous times, it was found out that PBA performed better than HPA in terms of the number of pods being spawned. Figure. 5 and Figure. 6 shows the number of pods that are

<sup>6</sup><https://grafana.com/>

<sup>7</sup><https://k6.io/>

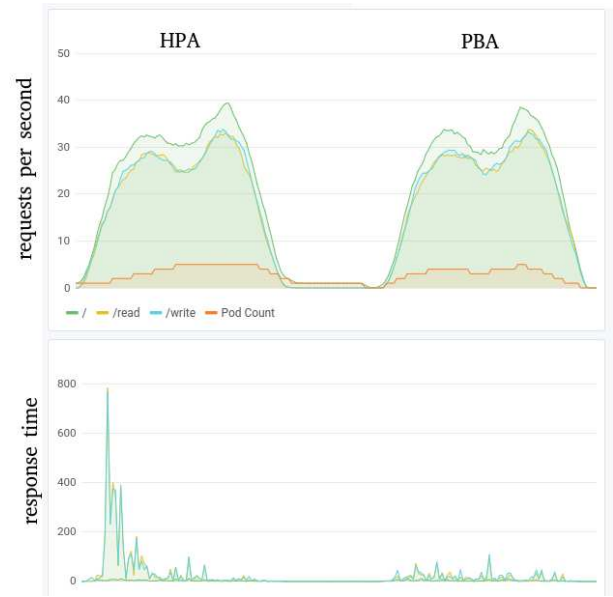


Fig. 4. Comparison of the response time between HPA and PBA

being consumed by HPA and PBA respectively. Since the amount of resources used is proportional to the pod count, we can infer that PBA 18% less resources than HPA.



Fig. 5. HPA pod count



Fig. 6. PBA pod count

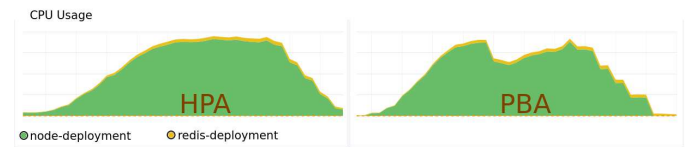


Fig. 7. Comparison of the response time between HPA and PBA

4) *Overall CPU and Memory consumption*: Figure. 7 and Figure. 8 compare the CPU and memory usage respectively for both *nodejs* and *redis-db* deployment. HPA scales up or scales down according to the percentage of CPU used, whereas PBA has the ability to scale up or scale down the number of pods

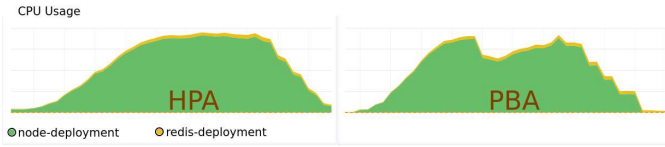


Fig. 8. Comparison of the response time between HPA and PBA

on the basis of the prediction made earlier with the help of SARIMA model. The same logic is applicable for the amount of memory used. The consumption of CPU and memory is found to be much lower in PBA than in HPA.

## VI. CONCLUSION AND FUTURE WORK

Serverless computing is being adopted by a lot of applications, but there is still a major chunk who are not able to use the benefits of serverless computing due to the problem of cold start. It is very well proven that a cold start takes significant amount of time more, than invoking a function whose container is already deployed. The challenging part is to reduce the cold start latency without the consumption of extra resources. In this paper, we intended to reduce the cold start latency as much as possible, but not at the expense of additional resources. We propose to predict the future incoming requests with the help of a classical statistical model called SARIMA. After getting the predictions from the model, we feed it into the PBA, which scales up or scales down the number of pods in accordance with the predictions made by the model. This facilitates the provisioning of roughly exact number of resources that are required by the application at any given point in time. Therefore, there is no over-provisioning or under-provisioning of resources, and the application runs in a smooth manner. In order to validate our strategy we compare our PBA with HPA, which is the default pod autoscaler in Kubernetes. Upon comparing the two, HPA suffered from cold start whereas PBA was able to subdue the effect of cold start in a graceful manner. PBA also consumed 18% less resources than HPA.

There still remain many aspects of cold start in serverless computing that need to be better understood and refined. Future predictions is one of the ways to tackle this problem, but a significant amount of research is required to better understand the core of the problem and if there are any innovative solutions to it. Though cold starts have been in the lime light lately in the world of serverless, they are little known outside the spheres of cloud computing. Nevertheless, research in this field is going on strong, and hopefully, In the coming years, the problem of cold start would perish.

## REFERENCES

- [1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.
- [2] S. K. Addya, A. Satpathy, B. C. Ghosh, S. Chakraborty, S. K. Ghosh, and S. K. Das, "CoMCloud: Virtual machine coalition for multi-tier applications over multi-cloud environments," *IEEE Transactions on Cloud Computing*, to appear, 2022, DOI:10.1109/TCC.2021.3122445.
- [3] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2289–2301, 2020.
- [4] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 257–262.
- [5] S. Benedict, "Serverless blockchain-enabled architecture for iot societal applications," *IEEE Transactions on Computational Social Systems*, vol. 7, no. 5, pp. 1146–1158, 2020.
- [6] I. Wang, E. Liri, and K. Ramakrishnan, "Supporting iot applications with serverless edge clouds," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–4.
- [7] M. M. Rahman and M. H. Hasan, "Serverless architecture for big data analytics," in *2019 Global Conference for Advancement in Technology (GCAT)*. IEEE, 2019, pp. 1–5.
- [8] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2019, pp. 1–10.
- [9] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.
- [10] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2020.
- [11] P. Vahidinia, B. Farahani, and F. S. Aliche, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2020, pp. 1–7.
- [12] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 181–188.
- [13] Kubernetes. (2022) <https://kubernetes.io/docs/home/>.
- [14] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.
- [15] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 133–146.
- [16] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195.
- [17] A. U. Gias and G. Casale, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.
- [18] B. C. Ghosh, S. K. Addya, N. B. Somy, S. B. Nath, S. Chakraborty, and S. K. Ghosh, "Caching techniques to improve latency in serverless architectures," in *2020 International Conference on COMMunication Systems NETWORKS (COMSNETS)*, 2020, pp. 666–669.
- [19] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 356–370.
- [20] Z. A. Farhath, B. Arputhamary, and L. Arockiam, "A survey on arima forecasting using time series model," *Int. J. Comput. Sci. Mobile Comput.*, vol. 5, pp. 104–109, 2016.
- [21] Seasonal Autoregressive Integrated Moving Average. (2021) Sarima. Available: <https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/>.
- [22] Github repository. (2021) monitoring. Available: <https://github.com/prometheus-operator/kube-prometheus/tree/main/manifests>.

This figure "col\_start.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "dataset.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>



This figure "fig1.jpg" is available in "jpg" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "fig1.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "five\_pod.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "hpa.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "hpa\_pod\_count.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "hpavspba\_cpu\_usage\_node\_\_\_\_redis.png" is available in "png" format from

<http://arxiv.org/ps/2206.15176v1>

This figure "hpavspba\_memory\_usage\_node\_\_\_\_redis.png" is available in "png" form

<http://arxiv.org/ps/2206.15176v1>

This figure "one\_pod.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>



This figure "pba\_algo.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "pba\_pod\_count.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "predictions.png" is available in "png" format from:

<http://arxiv.org/ps/2206.15176v1>

This figure "response\_time\_comparison\_mpfs.png" is available in "png" format from

<http://arxiv.org/ps/2206.15176v1>