National College of Ireland

Optimization of Resource Allocation and Prediction Analysis in Serverless Computing using Dynamic Resource Algorithm

MSc Research Project
Cloud Computing

Saifali Sayyed
Student ID: x18178294

School of Computing
National College of Ireland

Supervisor: Prof. Vikas Sahni

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Saifali Sayyed
Student ID:	x18178294
Programme:	Cloud Computing
Year:	2020
Module:	MSc Research Project
Supervisor:	Prof. Vikas Sahni
Submission Due Date:	17/08/2020
Project Title:	Optimization of Resource Allocation and Prediction Analysis in Serverless Computing using Dynamic Resource Algorithm
Word Count:	6624
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	
Date:	16th August 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Optimisation of Resource Allocation and Prediction Analysis in Serverless Computing using Dynamic Resource Algorithm

Saifali Sayyed
x18178294

16th August 2020

Abstract

Resource Allocation in Serverless Computing is an essential factor when there is a necessity for better application performance. But, to allocate the resources dynamically based on the serverless application requirement is a tedious task. In this proposed approach, OpenSource Serverless Platform has been leveraged named OpenFaaS for creating the functions which are deployed on AWS EC2 Instance. In this novel approach, Dynamic Allocation of Resources for serverless functions has been implemented using the Dynamic Resource Algorithm. The algorithm analyzes the past resource consumption and based on the thresholds configured for each resource, it allocates the resources to serverless functions. Moreover, the proposed approach, predicts future resource utilization of serverless functions using the ARIMA time-series analysis model. Once, the resources are forecasted they are sent to the AWS CloudWatch Dashboard. Next, the Dynamic Resource Algorithm is evaluated with the Default Docker Resource Allocation Method. Whereas, ARIMA time-series model is evaluated with the Standard time-series model. Lastly, their results are explained and compared with each other and concluded which worked best.

1 Introduction

Serverless computing has become an emerging technology and also called as Function-As-A-Service(FaaS). It allows the developers to execute the code without provisioning any compute services and only charge for the execution time the code has taken. Because of its flexibility, most of the companies have accepted this paradigm and have started to gain benefits from it. The market-size of serverless computing by 2021 will get increase to 7 Billion as stated by Google Trend. Public Cloud providers have started to provide these services on their platform for their users. The well-known platforms of FaaS are AWS Lambda on AWS, Google Cloud Functions on GCP, and Azure Functions on Microsoft Azure Cloud. Castro et al. (2019).

As serverless computing platforms are provided by the Public Cloud Providers they use a Shared Responsibility Model where the users are responsible for the code storage, code security, and permissions. Whereas, the Cloud Providers are responsible for the

underlying platforms and infrastructure[1]. Consequently, the users or customers do not have the permissions to access or monitor the underlying resources on which their functions are executed. This creates a problem when resources have to allocated as per the requirement of the function.

1.1 Motivation and Background

From the user's perspective, there are some challenges that have occurred while using the public cloud serverless platform such as resource allocation and monitoring the underlying platform. The current technique for allocating the resources to a serverless function is, first, the memory size has to be allocated and proportion to that the CPU power is assigned which is not efficient. This will put the lambda function out of resources and eventually will affect the overall application. Malawski et al. (2017). However, the serverless platform depends on the containerization technique, in which whenever the function is executed or triggered by an event, it creates a container on which the function code is executed. Savage (2018). So for monitoring the resources that are consumed by the functions can be done by monitoring the underlying container itself. This technique can help for allocating the resources to a container based on the function requirement and its application nature. This heuristic approach can be carried out using the OpenSource Serverless Platform such as OpenFaaS in which it allows us to monitor the underlying function platform such as container and also provides us the authority to allocate the resources to the container based on its usage. The proposed approach is implemented for dynamically allocating the resources to container based on its previous usage and predicting the future resource usage of a lambda function using the ARIMA time-series analysis model.

1.2 Project Specification

The proposed approach solves the resource allocation and monitoring issues for the serverless computing platform. Furthermore, the paper comprises the evaluation and implementation of resource optimization and allocation to lambda functions and predicting future resources of a function using the ARIMA time-series analysis model. For dealing with the resource allocation problem in a serverless platform, the below research question is proposed.

1.2.1 Research Question

Current techniques for allocating the resources to lambda functions do not provide efficient results, consequently affects the application performance. Resources such as CPU, Memory, and BlockI/O runs out of resources if not managed properly.

Research Question 1: *"Can Resource Management in Serverless application be optimised using novel Dynamic Resource Allocation Algorithm on different cloud platforms e.g AWS and OpenStack?"*

Research Question 2: *"Can future resources be predicted for a lambda function using the ARIMA time-series analysis model?"*

1.2.2 Research Objectives

The following objectives are addressed by the research question: Objective 1: First, the implementation of retrieving the monitored metrics for the serverless functions are

[1]https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html

performed. Objective 2: Next, Based on the gathered metrics, the analysis are performed for implementing the dynamic resource allocation to the functions underlying container. Objective 3: Once, the allocation is done for Cpu, Memory, and BlockI/O, then the resource predictions are implemented using the time-series ARIMA model. Objective 4: After the predictions, at last, both approaches which are Dynamic Resource Algorithm and Time-Series ARIMA Model is evaluated.

1.2.3 Research Contribution

In the proposed research, multiple studies have been reviewed related to the research question. Further, the implementation has been carried out of the proposed approach. Lastly, the proposed approach is evaluated and results are compared and concluded which works best. The structure of the research paper is as follows: In section 2, various studies has been reviewed related to the proposed approach. Further, the subsections in section 2 are in the logical format of the proposed approach. Next, in section 3, the tools, and techniques that are leveraged in the proposed approach have been discussed. Section 4, specifies the design specification. Whereas in Section 5, the proposed approach implementation is carried out. In Section 6, the proposed approach is evaluated and results are compared and has been specified which works best. Lastly, in Section 7, demonstrates the conclusion and future work of the proposed work.

2 Related Work

Function-as-a-Service has become an essential platform for reducing the overall public cloud cost billings and moreover for increasing the performance of an application. In this section, multiple studies that are based on the research topic will be explained. Furthermore, the critical analysis and the research gaps will also be specified if there are any.

2.1 Benefits of using OpenSource Function-As-A-Service

Serverless lambda functions cause multiple benefits only when they are implemented appropriately. As in this study Mohanty et al. (2018), the performance evaluation of multiple open-source serverless platforms namely Fission, Kubeless, OpenFaaS, and OpenWhisk is presented. The document has also mentioned the drawbacks such as resource allocation, size of the code, duration of the lambda function takes to executes, programming languages, and vendor-lockin on a public cloud platform. The evaluation is based on the response time and success ratio. After the evaluation, it was concluded that OpenFaaS is the most flexible platform and Kubeless provides the best performance in various scenarios. The **important characteristic** which is found is that the Fission can be configured when there is a need for low latencies as it can keep the containers in the warm state. The **research gap** which is found is that the other container orchestrator tools are not explored for performing the evaluation test.

In this next study Kim et al. (2018), for better performance, the author has implemented a GPU based serverless architecture that is integrated with the Docker NVIDIA container. This study has used an Open Source serverless platform and integrated it with the GPU environment so that the CPU intensive applications can work efficiently. This paper shows the flexibility of serverless platforms. This is what has implemented in the proposed approach. The **study gap** which is found in this study is that the author has

not implemented it on the container orchestrator platform which would have turned in better performance.

As far as public cloud serverless services are concerned, they have also been useful where the underlying infrastructure monitoring is not considered. Giménez-Alventosa et al. (2019) propose an approach where a map-reduce job is deployed on the AWS Lambda functions. In this study, a mechanism is implemented to analyse the resource allocation system of the AWS Lambda. As public cloud services were used in this study there were some complications related to the programming languages as it does not support all the languages. The arguments and claims are very well supported as they have properly mentioned the configuration details in advance with the version details of the platforms.

2.2 Tracking of Metrics and Management of Containers

In this section, monitoring and retrieving the metrics for docker containers is discussed. It becomes an essential factor when the real-time metrics need to be considered for further analyzing and allocation of the resources. Chang et al. (2017) provides us the research study for monitoring and tracking the metrics for the same. This study proposes a comprehensive platform for monitoring the containers and making further decisions based on that. They have used the Heapster platform for accessing and collecting the metrics for the containers. However, in our proposed approach, an automated way is implemented using a bash script for collecting these metrics. The author has also used InfluxDB for storing these metrics.

Here is another study Enes et al. (2018) which is based on collecting the time-series monitored metrics. This study proposes a new approach for gathering the applications monitored data rather than the whole instance metrics. Which is almost the same case in our project, in which gathering the time-series monitored metrics of those containers on which our functions are deployed. The **research gap** found in this study has not explored more towards gathering the monitored metrics from the underlying Cgroups instead of that another tool is used. Furthermore, related to this paper is that in our proposed approach the gap which is present in this reviewed study is implemented by retrieving the monitor data from the underlying host itself.

The next study propose Kubernetes architecture in which the author MackDioufa et al. (2020) has implemented an architecture for achieving the fault-tolerance of an application. The interesting thing to look here is that their solution has customized the Kubernetes controller for getting the monitored metrics via API server from the deployed cluster. This method of getting the metrics from the Kubernetes platform itself is the best practice method. However, it does not provide full metrics other than CPU and memory metrics. In which, in our project that is not the case.

The arguments and claims made in these documents are very well supported by genuine references and can be easily reproduced as they have clearly mentioned the architecture and configuration details as well.

2.3 Impact of Container Orchestration on Applications

In this section, the discussion on various studies based on the container orchestration is presented. Carlos et al. (2018) proposes an approach, in which their aim is to optimize the container orchestrator on which they have deployed a micro-service based application. Additionally, they have also evaluated the performance of their approach with the Greedy

and NSGA algorithm. After the evaluation, it is found that the proposed approach increased the performance of an application and also for container orchestration. The evaluation results are specified using the graph charts. The configuration details are mentioned properly so that the reproduce can be easily done if required. Lastly, there is not much to find the **research gaps** in the study but however, some areas should have been explored more for an instance the greedy approach.

In this study, Acuña (2016) defines that with the help of Minikube it is easy to use Kubernetes where a single node cluster has to deploy such as on a laptop or virtual machine. It is an easy tool that can be used to get started with Kubernetes for deploying any single node cluster application. In the proposed approach,the Minikube tool has been leveraged for creating a single node cluster for deploying the OpenFaaS platform and to execute the functions on top of it. In the next study, Gogouvitis et al. (2020) proclaims that when they integrated their approach with the container orchestrator such as Kubernetes they have found seamless computation in the results which increases the performance of their approach. This overall approach was carried out with the Kubernetes platform. In this next study Pereira Ferreira and Sinnott (2019), propose the evaluation for the public cloud Kubernetes service. It is found that the selection of the Public Cloud Kubernetes platform depends on the applications nature. For instance, if the application is CPU intensive then AWS EKS will be the best choice, if its network-intensive then the Google Kubernetes will be the better choice. Further, the underlying host machine is more responsible when performance is concerned. That is the reason in our novel approach the Open Source Kubernetes platform is leveraged so that the flexibility is maintained for selecting the underlying host and configuration details which will help us to dynamically allocate the resources to the container as per the requirement. The claims and arguments are very well supported using authentic references.

2.4 Optimisation and Dynamic Allocation of Resources to Containers

Dynamic allocation of resources is an imperative technique on every computing service be it as virtual instances or containers. In this subsection, Piotr and Soares (2018) states the approach for allocating the docker containers onto the virtual machines based on the values, priorities, application nature type, and resources required, which is almost the same in our novel approach. However, the imperative factor to analyze here is the reviewed study has not considered the resource metrics such as memory and network. This is the **research** gap has found in this study, however, in our novel approach, this is not the problem. After evaluating the performance with the Docker Scheduling Algorithm itself, they found their approach has increased the overall performance of the system, especially for the CPU heavy workloads.

In this paper Xinjie et al. (2017), the author has implemented the resource allocation to containers and scaling of containers when particular thresholds met or more resources are required. By implementing this, their imperative focus is to decrease the application implementation cost. They have fairly achieved their aim. The arguments and claims are very well supported using authentic references. Furthermore, this study has also compared their algorithm with other models such as greedy and best-fit. Retrieving the real-time metrics and based on that metrics the dynamic allocation decision is made. Enes et al. (2020) propose the same approach for real-time big data applications but with more advanced functionality such as real-time scaling as well. They have implemented an approach where they are retrieving the monitored metrics for the docker container.

When the time for scale-up comes, that means upgrading the resource size instead of creating the new container the allocations are performed for the big-data applications.

2.5 Prediction performed using ARIMA model

As our proposed approach predicts the resources for the serverless functions it becomes an essential task to analyze which model will best suit our environment. Dalmazo et al. (2017) propose an approach where they are predicting the network resources for an application to work seamlessly. The technique which they have used is the ARIMA model for predicting the network resources. Whereas, in our proposed approach, prediction of all the resources is considered which are required for a containerized application to work efficiently.

El Mezouari and Najib (2019) propose the predictions for the soil moisture and irrigation culture to provide the results in a proper way. In this approach, they have implemented a Hadoop based architecture in which the real-time metrics are gathered of the soil conditions and irrigation resources consumed. Furthermore, they are predicting the data using multiple prediction algorithms such as XG-Boost, Random Forest, and ARIMA. After the evaluation, they found that all these models provided precise results. But the ARIMA model provided the best results for the soil conditions.

The underlying host becomes an essential factor when the performance of an application has to be calculated. This next study Yi et al. (2013) propose an approach where they are evaluating the performance for the AWS EC2 and Azure compute service based on the requests received and processes that running. Furthermore, they are also predicting the running instances that will be placed in the future on the underlying cloud platform hosts. The prediction is done by using the ARIMA model which best suits the requirements on their approach.

Table 1 : Comparison of Reviewed Studies

Reviewed Studies	Aim	Dynamic Allocation	Time-Series Prediction
Piotr and Soares (2018)	Dynamic Placement of Containers	Yes	No
Xinjie et al. (2017)	Resource Allocation and Scaling based on threshold	Yes	No
Enes et al. (2020)	Resource Allocation for Big Data Apps	Yes	No
Dalmazo et al. (2017)	Predicting the Network Resource	Yes	Yes
Yi et al. (2013)	Prediction of Virtual Machines	No	Yes
Proposed Approach	Dynamic Resource Allocation and Predicting Future Resources	Yes	Yes

2.6 Conclusion

Many studies have been reviewed in this section related to our proposed approach. Multiple studies are there in which there are some research gaps which is found. These gaps are filled using our proposed approach using the dynamic resource algorithm and implementing the resource prediction using the ARIMA time-series analysis model.

3 Methodology

In this section, the research methodology of the proposed approach is explained from start to end. It contains the phase from selecting and implementing all the pre-requisites which are needed for the project to work until the final results of the data prediction.

3.1 Tools and techniques used in the proposed approach

Docker: Ahmed and Pierre (2019) has stated Docker as a light-weight Opensource platform for executing and encapsulating an application. This approach also leverages the Docker API calls for retrieving the real-time monitoring metrics with the help of *docker stats* command.
Kubernetes: It is a container management platform. Wu et al. (2019) states that Kubernetes also balances the loads between the containers in a very efficient manner. Apart from that it also restarts and brings back to life if any container fails.
OpenFaaS: Palade et al. (2019) demonstrates OpenFaaS as an opensource serverless platform on which users can build and invoke their functions. This study describes the two fundamental modules of OpenFaaS which are functions and handler. OpenFaaS API Gateway is responsible for scaling and collecting the metrics which are directly sent to **Prometheus** monitoring tool. Sukhija and Bautista (2019) manifests Prometheus as an open-source monitoring tool in which time-series monitored data are stored. **Jupyter Notebook:** Ueno and Imai (2020) proclaims that Jupyter Notebook is a web application that executes the code. It helps users to write code step by step in a notebook file. In the proposed approach, Jupyter Notebook is leveraged for executing the **ARIMA** prediction python code. ARIMA is a model that is responsible for predicting the time-series data. Next, the ARIMA predictions are forwarded to the **AWS CloudWatch**.

3.2 Proposed Approach Overall Structure

Step1: In this step, the collector module comes into account for collecting the resources metrics that are consumed by OpenFaaS function container on first execution.
Step2: Next, the controller module analyzes these metrics, and based on the consumed metrics it performs the calculations and allocates the resources to the function. The thresholds are configured based on resource utilization in percent and MBs. Such as, if CPU utilization is greater or less than 50 percent then based on that it will allocate the required CPU-shares to the OpenFaaS functions. Next, the memory allocation is based on the MBs utilized. Such as, if memory utilization is less or equal to threshold values set, then based on that the memory resource is calculated and allocated to the functions. For blockio, it depends on the block I/O utilized in megabytes.
Step3: Further, is the optimiser module which will again execute the function after the optimised resource values are allocated. This will show us the unused resources now after the optimised resource allocation process has been done.
Step 4: Last module is the prediction module, which will predict the resource utilization of the serverless function using the ARIMA time-series analysis model.
Step 5: These values are sent to the AWS CloudWatch monitoring tool as expected future resources using the AWS CloudWatch Agent.
Figure 1 provides more in-depth details of the proposed approach.

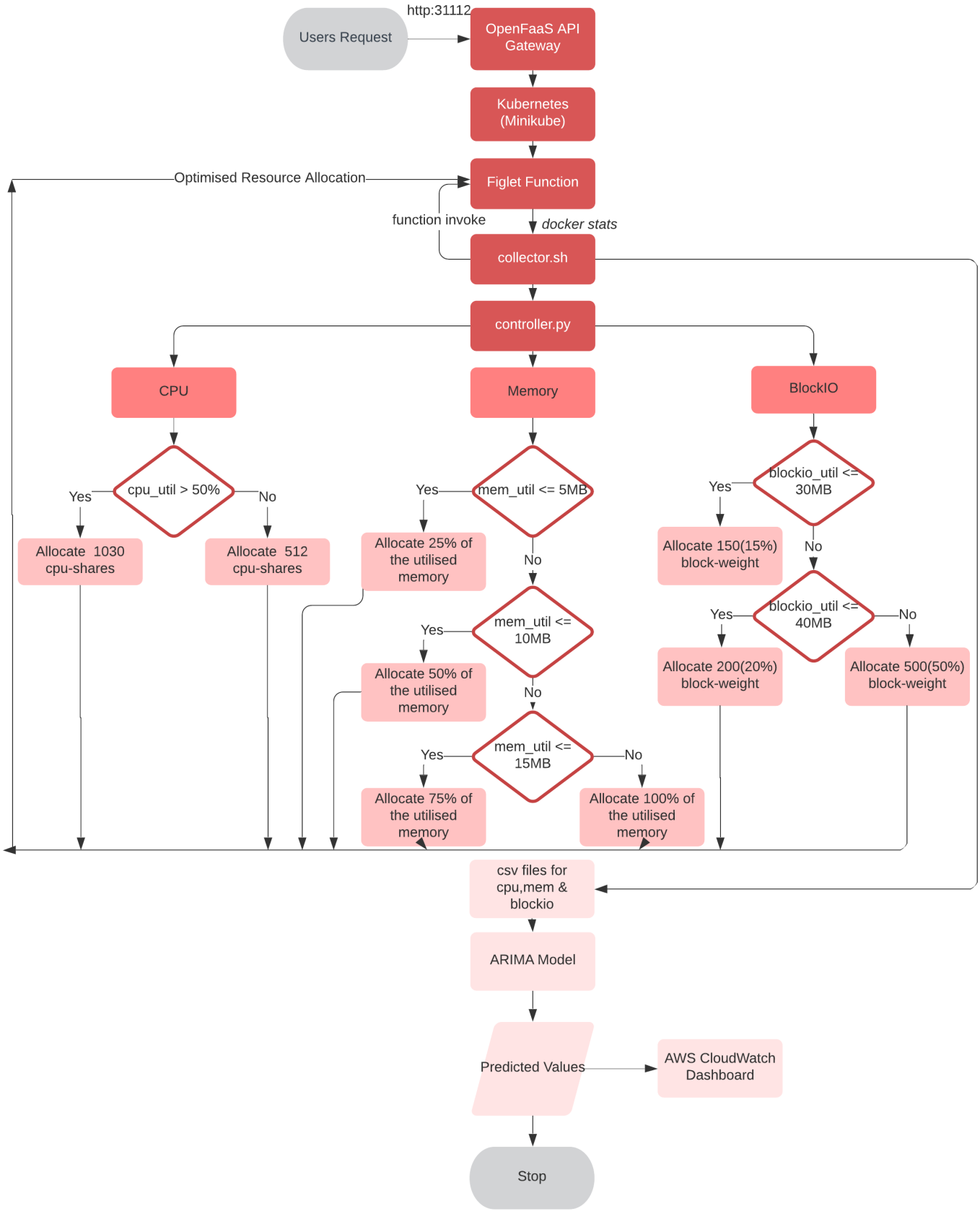


Figure 1: Proposed Approach Flowchart

4 Design Specification

The below figure 2 depicts the proposed approach architecture diagram. The components of this architecture are explained below.

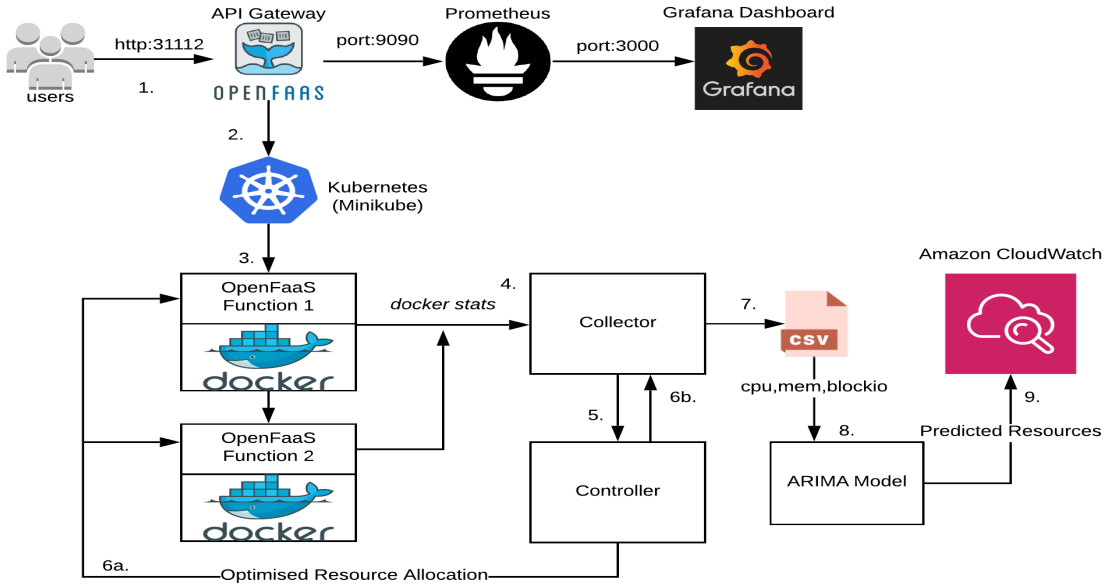


Figure 2: Proposed Approach Architecture

4.1 Data Collection of Monitored Containers

In this section, the process of run-time metrics resources of OpenFaaS functions are collected using the Docker API command named as *docker stats*. Further, these metrics are collected to a file named docker-metrics in which the data is again filtered and rounded up to the integer value. The resources consumed by the functions are stored as a total percent used. Further, these values are passed as an argument to the controller module and based on these values further analyzing is performed which is explained in the next section 4.2

4.2 Analyzing the Collected Data

Once the resource arguments are received by the controller module, they are sent to perform optimization analysis. First, the controller optimizes the CPU-shares of an OpenFaaS function. If the passed arguments that is the CPU percentage are more than 50 percent then the CPU controller will allocate 1030 of CPU-shares to the function which corresponds to the 1 CPU core. However, if the CPU percentage is less than 50 percent than it will allocate 512 CPU-shares that correspond to the half CPU core. Once CPU resource optimization is done, next comes the memory usage. If the memory utilization is less than or equal to thresholds configured in the memory segment of the controller than allocate the memory based on the memory limits specified in the controller. Next, the block I/O section is considered, for optimising the block I/O resource. Based on the blocks consumed by the OpenFaaS functions and the thresholds set for blocks the blockio-weight is allocated to the function container.

4.3 Dynamically Allocating the Optimised Resources

This section illustrates the dynamic allocation of the optimised values which are analyzed in the above section 4.2. The optimised values are stored in a python variable and using the *docker update* Docker API command the optimised resources are allocated to the OpenFaaS functions. Next, once these resources are allocated to the container, then the next phase comprises of the optimized module which is responsible for again invoking the function and analyzing how much resource are now unused and how much is the resource utilization in percent and MB. Next, the prediction phase comes into picture where the collector is responsible again for fetching up the monitored data and generating a CSV files for each resource, which is discussed in next section 4.4

4.4 Resource Prediction using ARIMA

In this section, the prediction for future resource consumption is predicted using the ARIMA model. For executing the ARIMA model, the Jupyter notebook is implemented in which Python 3 language is used. Further, the CSV files which are generated by the collector module is configured in the Jupyter notebook for each resource. The CSV file contains the time-series analysis data which are in comma-separated and resource utilized in percentage. The library which are used for predicting the values are tsa plots, pandas, and ARIMA. After forecasting the predicted resources they are sent to the AWS CloudWatch Dashboard as an expected future resource for serverless functions.

5 Implementation

This section will explain the actual implementation of the proposed approach.

5.1 Creation of OpenFaaS Function and Collecting the Resource Metrics

In this section, the process of creating an OpenFaaS function and collecting the resource metrics for the specific function is explained. The OpenFaaS figlet function is created with the help of the OpenFaaS CLI commands. As soon as the figlet function on the OpenFaaS is created, it creates a docker container with the help of the Kubernetes Minikube tool. Next, the Minikube creates the container under a namespace called as openfaas-fn where all the OpenFaaS functions are created. This figlet function is responsible for converting the normal text into an ASCII character.

Further, for generating some load on the underlying figlet container, this function is invoked 500 times and meanwhile, monitoring and gathering of the resource metrics are also performed. Once these metrics are gathered from the *docker stats* Docker API call, it is stored in a file for performing further analysis. These stored metrics are then fetched from the collector.sh script which is a bash script. The collector.sh script will filter out the unnecessary symbols and characters which are attached to resources values, for example, a percent sign and units, and will round up to the single integer value and will store these values in a resource variable. Once these values are stored in a bash variable then they are passed as an argument to the controller.py file which is a python file. Furthermore, for retrieving the information regarding the OpenFaaS function, such as execution duration, invoked count, and scaling of the replica can be seen on the Grafana Dashboard which can be seen in Configuration Manual.

5.2 Optimising the Figlet Function Resources Using Controller

As soon as the controller.py receives the arguments from the collector script. It uses these resource values for optimizing the resource allocation and will also analyze how much resources were used and unused. Next, the resource optimization and allocation are performed based on the CPU percent utilized, Memory and BlockI/O in MBs. Once these values are allocated after the Dynamic Resource approach it displays us the new resource utilization with the used and unused resource for the figlet function.

The CPU-share is calculated based on the past utilization consumed by the figlet function container. The memory allocation is performed based on the thresholds and mathematical calculation performed using the past memory metrics. Next, the block-weights allocation comes into account where the block I/O resource is allocated based on the block I/O consumed by the figlet function container.

5.3 Resource Prediction and Monitoring it on AWS Cloud-Watch

In this section, the prediction for CPU, memory, and blockI/O process using the ARIMA time-series method has been explained. The prediction for CPU, memory, and blockI/O is thoroughly implemented. The prediction is performed on the figlet function container resource usage. These resource values are fetched from the metrics file which was created by the collector file. The collector.sh script is then responsible for converting the resource consumption values into a time-series data format for configuring the CSV in the Jupyter notebook.

The Jupyter notebook is installed using the pip3 module on the AWS EC2 ubuntu 16.04 instance. Jupyter Notebook uses python 3 language for executing the ARIMA time-series code. Next, the CSV files which are generated from the collector is checked manually for finding any corrupted or unnecessary values which will affect the prediction results.

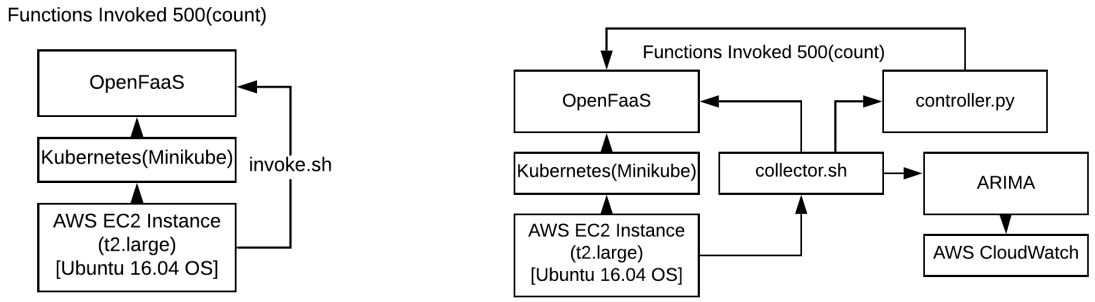
Further, the parameter variables for p,d,q are selected based on the auto-correlation and partial-correlation plots. These plots will show the correlation between past values and also the lag values correlations. Next, the AIC (Akaike Information Criteria) statistical model is leveraged for analyzing the righteousness fit and the clarity of the model. Low AIC represents the better model, hence high represents a complex model. Further, the forecast method is implemented for forecasting the resource values and then they are evaluated with the past values which will show the prediction results and its differences. Once the evaluation is performed between the past and future values, then the most appropriate values are sent to the AWS CloudWatch Dashboard that will represent the future expected resource utilization values.

6 Evaluation

In this section, the evaluation for Dynamic Resource Algorithm(DRA) and ARIMA model has been conducted. Both of these models are validated and lastly, the results of both of them are analyzed.

6.1 Experiment 1: Evaluation for Proposed Approach

In experiment 1, there are two subsections in which section 6.1.1 will evaluate the performance of the Dynamic Resource Allocation Algorithm, and section 6.1.2 will evaluate the time-series ARIMA model. Whereas, in Experiment 2, the function invoke count is increased to 2000 for evaluating the DR Algorithm.



(a) Default Docker Resource Allocation

(b) Using Dynamic Resource Algorithm for resource allocation

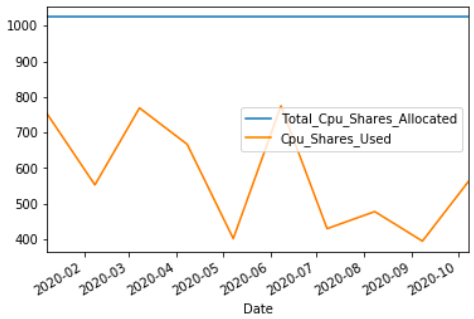
Figure 3: Process of Evaluation

6.1.1 Dynamic Resource Allocation Evaluation

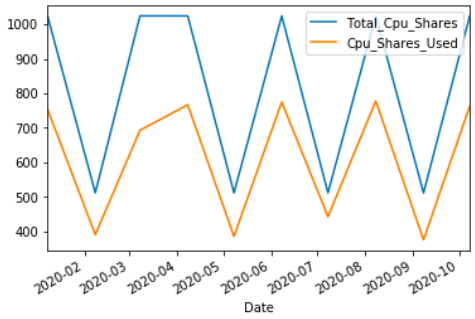
In this section, the evaluation for the Dynamic Resource Algorithm (proposed approach) is performed with default Docker Resource allocation. Two scenarios are implemented and compared in the evaluation. In the First scenario, the Docker Default Resource allocation is used. Whereas, in the second scenario the DR algorithm is used for the OpenFaaS functions container for dynamic resource allocation. Furthermore, the results are shown and compared of both of these scenarios and analyzed which worked best. In the first scenario, the OpenFaaS function was invoked 500 times using a bash script. The default Docker Resource allocation for the figlet function container was the same as the resource size as the host machine. Now, in the second scenario, the resource allocation was performed by the proposed approach which is the DR algorithm. In this, the resource allocation was done dynamically and based on its usage. So that no resource should be underutilized. First, the Cpu resource is evaluated with default docker resource allocation and with the proposed approach.

In figure 4, two graphs are shown, in which CPU-Shares allocated and CPU-Shares consumed on Y-axis, and the X-axis represents the dates. Figure 4a represents default docker allocator resource allocation. As the 4a represents there is a lot of CPU-Shares has been wasted especially on 06 and 09 August. But when figure 4b is considered which represents the CPU-Shares allocation using the Dynamic Resource Algorithm, it shows the tremendous difference in the CPU-Shares allocated. Figure 4b also represents when CPU-Shares consumed was less than 512, the proposed approach allocated 512 CPU-Shares so that the CPU-Shares should not be underutilized. So our proposed approach performed better **CPU optimization** as compared to the Default Docker Resource allocation.

Next for memory, the figure 5 represents memory allocated and consumed in MB on Y-axis whereas the dates on X-axis. In figure 5a, the default memory allocated to the figlet function container was 8000MB which is 8GB. The memory consumed by the figlet container was very less as compared to memory allocated. Figlet container was using 7



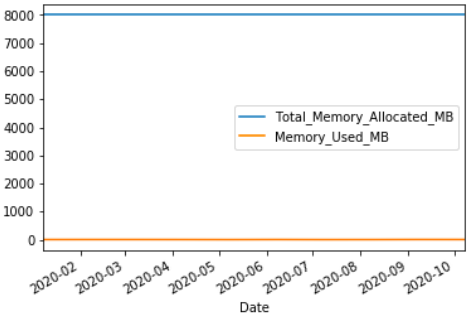
(a) CPU-Shares Allocation by default Docker Resource allocation



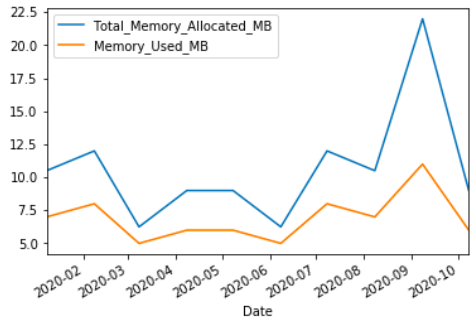
(b) CPU-Shares Allocation using Dynamic Resource Algorithm

Figure 4: Evaluation for CPU Resource

to 15 MB maximum for processing the functions. This proves that most of the memory such as 7993 MB is wasted by the figlet container. But, when the allocation was done by the dynamic resource algorithm which is shown is the figure 5b, the memory allocation was done dynamically as per its usage. For instance, on 02 August, figlet container took 8 MB of memory for processing the request, so the total memory which was allocated to the container was only 12 MB. So the calculation goes like, 8 MB was utilized previously and when buffer has to be add then 50% of the utilized memory, that is 50/100*8=4, this 4 value is added in 8 MB as 4MB as buffer value. So the total comes to 12 MB which is allocated to the memory. So when we compare the difference between the default docker resource allocation and dynamic resource algorithm (proposed approach), our approach provides better **memory allocation** for figlet container which saves a lot of memory for the host machine and other containers as well.



(a) Memory Allocation by default Docker Resource allocation



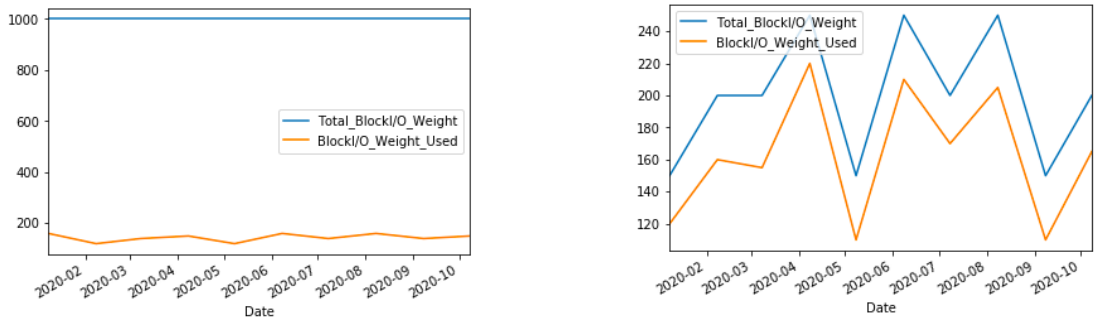
(b) Memory Allocation using Dynamic Resource Algorithm

Figure 5: Evaluation for Memory Resource

Furthermore, for the BlockI/O bandwidth, figure 6 represents the BlockI/O evaluation. The Y-axis represents the BlockI/O relative weight and X-axis the dates. In figure 6a, 1000 Block-Weight is the default allocation done by the docker daemon [2]. The block-weight which was used by the figlet container was around 120 to 160 which is 30 to 60 in megabytes. Most of the block-weight was got wasted. But when the dynamic resource allocation was performed on the Block-Weight, which is represented in the figure 6b it showed immense results. The calculations which was done for allocating the block-weight

[2]https://docs.docker.com/engine/reference/commandline/update/

were if the BlockI/O is more than 30 MB then attached the 15% of the block-weight of the host to the container. There were more thresholds configured for BlockI/O which can be seen in figure 1. For example, on 01 August, 30 MB BlockI/O was consumed which is 120 in block-weight. Such as 30/256*1024=120, so 30 block-weight is added as the buffer value and 15% of the block-weight of the host machine is allocated to the figlet container by the Dynamic Resource Algorithm. Here, 256 KiB is the block size of the AWS EBS and 1024 as the volume throughput[3]. As the figure 6b represents, it performed very well block-weight allocation as compared to the default docker allocation.



(a) Block-Weight Allocation by default Docker Resource allocation

(b) Block-Weight Allocation using Dynamic Resource Algorithm

Figure 6: Evaluation for BlockI/O

6.1.2 Time-Series ARIMA model evaluation

- **Standard Time-Series Prediction**

In this section, two scenarios are implemented, in the first scenario the time-series prediction is performed using the standard time-series model, whereas, in the second model the prediction is performed using the ARIMA time-series model. Lastly, their results are compared with each other and analyzed which worked best.

In the first scenario, Cpu, Memory, and Block I/O resources are included. This model is executed on the Jupyter Notebook by leveraging Python 3 libraries. For standard data prediction, the past ten days of resource utilization are chosen. The dataset is in the data frame format. Next, the statistical values are calculated for each resource type using the describe() function. Dataset and Statistical Analysis can be seen in figure 7.

The mean value depicts the daily average resource consumption value for each resource type and standard deviation shows the variability between the values. These values can be further used for more in-depth analysis.

Next, the standard time-series prediction is performed by shifting the past historical data to t+1. Where t depicts the past data and then it will pass the current day data to future value with +1. These predictions with the past data is as the best mirroring of the future data which can be seen in figure 8.

As shown in figure 8, the past value is predicted as the future value which is not the best prediction results. Further, the mean squared error is calculated for the current dataset. In which NumPy and mean squared error libraries are used. After calculating the mean squared error for the CPU the values were 10.6458, next for memory the error was

[3]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html

Date	Cpu_Used	Memory_Used	BlockIO_Used
2020-01-08	58	55	30
2020-02-08	51	58	36
2020-03-08	59	74	41
2020-04-08	67	72	23
2020-05-08	72	78	34
2020-06-08	51	58	43
2020-07-08	65	63	24
2020-08-08	57	66	34
2020-09-08	63	73	32
2020-10-08	72	64	23

(a) Standard Time-Series Dataset

	Cpu_Used	Memory_Used	BlockIO_Used
count	10.000000	10.000000	10.000000
mean	61.500000	66.100000	32.000000
std	7.633988	7.852105	7.118052
min	51.000000	55.000000	23.000000
25%	57.250000	59.250000	25.500000
50%	61.000000	65.000000	33.000000
75%	66.500000	72.750000	35.500000
max	72.000000	78.000000	43.000000

(b) Statistical Analysis

Figure 7: Standard Time-Series Analysis

	Previous_Cpu_Per	Previous_Memory_Per	Previous_BlockIO_MB	Forecast_Cpu_Per	Forecast_Memory_Per	Forecast_BlockIO_MB
1	51	58	36	58.0	55.0	30.0
2	59	74	41	51.0	58.0	36.0
3	67	72	23	59.0	74.0	41.0
4	72	78	34	67.0	72.0	23.0
5	51	58	43	72.0	78.0	34.0

Figure 8: Standard Time-Series Prediction

9.8262, and for BlockI/O it was 11.2200. These values are calculated based on average calculated for each resource type and its squared is calculated for displaying the mean squared error. However, in the ARIMA time-series model, this is not the case.

- **ARIMA Time-Series Prediction**

In the second scenario, the prediction for each resource is implemented separately. In this, Auto-correlation plots provides the correlation between the datasets and also show if there are some lag in the values. Whereas, in Partial Correlation, it depicts the correlations between the lag values which are the remaining value after the auto-correlation is plotted.

Further, for each resource, the statistical analysis is calculated. The mean value depicts the daily average resource consumption for CPU, Mem, and Block I/O which is shown in figure 9. The standard deviation values depict the variability in the values of the resource for CPU, Mem, and BlockI/O used.

Further, plotting was performed of the past time-series data on the auto-correlation then p=0 value was predicting as the absolute value which is 1. But when the next value which is p=1,2, there was some correlation between for each resource type, which can be seen in figure 10.

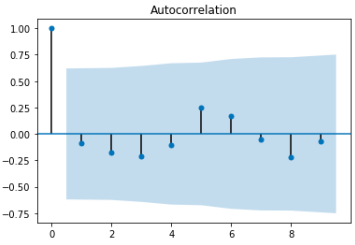
So to identify the q parameter, ACF (auto-correlation) chart has to be observed. Next for identifying the p parameter PACF chart (partial-correlation) has to be observed. So the model parameter values that are taken for CPU was (2,1,0) which was tested with various combination and received the best fit AIC values as 85.828. Next for the memory, the parameter value based on the ACF and PACF charts was taken as (1,1,0) with the

	Cpu_Used
count	10.000000
mean	61.100000
std	7.109462
min	51.000000
25%	57.250000
50%	61.000000
75%	66.500000
max	72.000000

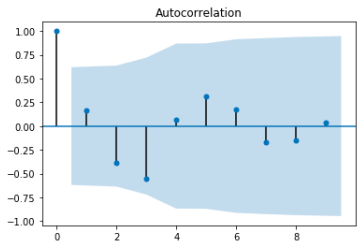
	BlockIO_Used
count	18.000000
mean	31.666667
std	6.507914
min	23.000000
25%	28.250000
50%	30.500000
75%	35.500000
max	43.000000

	Memory_Used
count	10.000000
mean	66.000000
std	7.888106
min	55.000000
25%	59.250000
50%	64.500000
75%	72.750000
max	78.000000

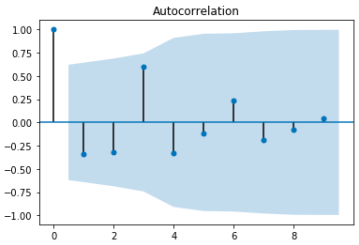
Figure 9: Resources Mean and Standard Deviation



(a) CPU Auto-Correlation



(b) Memory Auto-Correlation



(c) BlockI/O Auto-Correlation

Figure 10: Auto-Correlation

best fit AIC value as 55.449. Whereas, for the blockI/O (2,1,0) was considered with the best fit AIC value as 61.86.

Next, the values are forecasted for the CPU, memory, and blockI/O from the forecast method. Figure 11 depicts the forecasted values performed by the ARIMA model. In figure 11a, the three-step further predictions are considered in which the first value forecasted is 77.48% for CPU utilization and if the 01/August past value is observed, then there is only 5% of CPU resource difference. Next, memory utilization in percent is forecasted for the same date 01/August which is shown in figure 11b, which is 61.34 % and if it is compared with the past 01/August, then there is only 6% of the difference. The Same goes with the BlockI/O in figure 11c where 1MB of difference is forecasted for the first value. Next, for second value 12MB of difference is predicted by the ARIMA model which is a significant prediction.

```
cpu_util_forecast
array([77.48635685, 71.15073246, 62.5355443 ])
cpu_util
```

Date	Cpu_Used
2020-01-08	72
2020-02-08	73
2020-03-08	69

(a) CPU ARIMA Forecast

```
mem_util_forecast
array([61.34964001, 64.59340061, 63.29611248])
mem_util
```

Date	Memory_Used
2020-01-08	55
2020-02-08	58
2020-03-08	59

(b) Memory ARIMA Forecast

```
blockio_util_forecast
array([31.39291708, 28.38509928, 21.66170941])
blockio_util
```

Date	BlockIO_Used
2020-01-08	30
2020-02-08	36
2020-03-08	41

(c) BlockI/O ARIMA Forecast

Figure 11: ARIMA Resource Prediction

As soon as the data forecasting is done, the first index value from the array is sent to the AWS CloudWatch Dashboard as future expected resource consumption utilization

for each resource, which can be seen in figure 12. When standard time-series analysis and the ARIMA model are compared, the ARIMA model provides more realistic prediction values as compared to the standard time-series analysis.

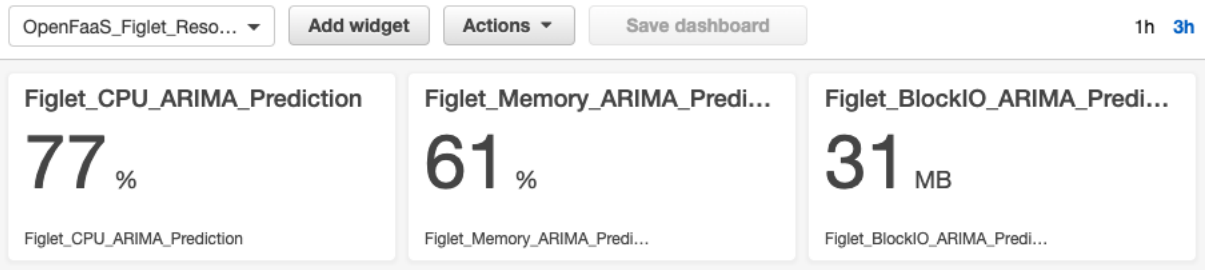
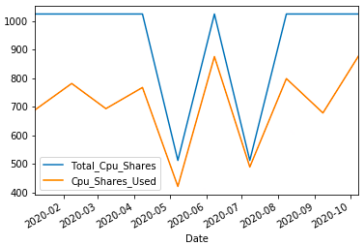


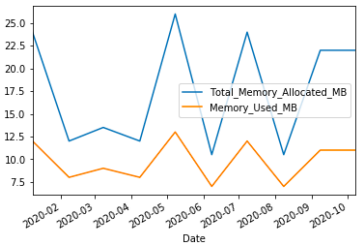
Figure 12: AWS CloudWatch Dashboard Predicted Future Resource Utilization

6.2 Experiment 2: Evaluation for proposed approach

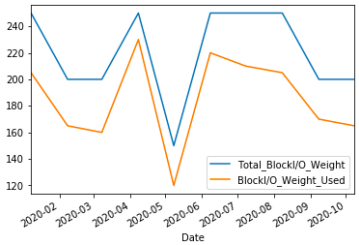
In Experiment 2, the figlet function invoke count was increased to 2000 in which resources utilization got high, especially for CPU, BlockI/O bandwidth. Below are the charts representing the Resource Allocation and Consumption when Dynamic Allocation of resources is performed after invoking the functions 2000 times. The important thing to note here is when the function was invoked for 500 times in the previous experiment. The dynamic allocation was done appropriately and the execution time for the function for processing the requests was also good. But, when the invoke count was exceeded to 2000 times, the execution time for the function of processing that requests were taking too much of time as compared to when it was invoked 500 times. This means that the dynamic allocation of resources and execution time for processing the requests for the figlet function performed well when it was invoked 500 times. But when the invoke count was increased to 2000 it showed some issues for processing the requests and was taking too much time for processing the 2000 requests.



(a) CPU Resource when invoking function 2000 times



(b) Memory Resource when invoking function 2000 times



(c) BlockI/O Resource when invoking function 2000 times

Figure 13: Dynamic Resource Allocation when invoking function 2000 times

6.3 Discussion

With the help of the above evaluations performed, it can be concluded that when the functions are invoked in less count the Dynamic Resource Algorithm works very well. However, when the functions are invoked in major count such as 2000, the dynamic resource algorithm allocates the resources appropriately but to process that 2000 requests

again the figlet function container takes much time as compared to the functions which are invoked for less count. Whereas, the experiment performed for the resource prediction using the standard and ARIMA time-series model specifies that the ARIMA model provides more realistic resource predictions as compared to the Standard Time-Series Analysis. Furthermore, the research question which is specified in Section 1.2.1, has been answered and the objectives specified in Section 1.2.2 have also been implemented appropriately.

7 Conclusion and Future Work

The aim of the proposed approach was to optimize the resources and dynamically allocate to the serverless functions using the Dynamic Resource Algorithm. Moreover, the predictions are performed for the future resources to be expected for the serverless functions using the ARIMA time-series analysis model. Resources such as Cpu, Memory, and BlockI/O that are responsible for application to work efficiently has been considered. As the OpenSource OpenFaaS Serverless platform leverages Docker container for processing the functions requests, the resource allocations are performed on the underlying Docker container of the function using the Dynamic Resource Algorithm. The proposed model is evaluated with the Default Docker Resource Allocation Algorithm and the results showed that our proposed approach works very well in terms of Dynamic Resource Allocation of the containers when the function was invoked 500 times. However, when the functions are invoked in major count such as 2000 count, the execution time for processing that requests after allocating the resources using the proposed approach took more time to process those requests. Additionally, the ARIMA time-series analysis for predicting future resource consumption for the serverless functions provided more realistic predictions as compared to the standard time-series analysis. Akaike Information Criteria (AIC) was leveraged for analyzing the best fit of the parameter selected for each resource in the ARIMA model. Lastly, for each experiment, the results are illustrated and explained with the help of graphical charts. The **future work** will be implementing the Dynamic Resource Algorithm for the real-time serverless applications which requires more processing and compute power, for instance, Big Data Applications. The NetworkI/O resource can also be implemented as future work in Dynamic Resource Algorithm. Whereas, for the ARIMA model the Seasonal ARIMA can be integrated with the Big Data Serverless Applications which will provide more realistic predictions and best AIC fit.

References

Acuña, P. (2016). Kubernetes., *Deploying Rails with Docker, Kubernetes ECS* p. 27.

Ahmed, A. and Pierre, G. (2019). Docker image sharing in distributed fog infrastructures., *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* pp. 135 – 142. CORE Ranking:B.

Carlos, G., Isaac, L. and Carlos, J. (2018). Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications, *Journal of Supercomputing* **74**(7): 2956 – 2983. JCR Impact Factor:2.157(2019).

Castro, P., Ishakian, V., Muthusamy, V. and Slominski, A. (2019). The rise of serverless computing, *Commun. ACM* **62**(12): 44–54. JCR Impact Factor:5.410(2019).

Chang, C., Yang, S., Yeh, E., Lin, P. and Jeng, J. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6. CORE Ranking:B.

Dalmazo, B., Vilela, J. and Curado, M. (2017). Performance analysis of network traffic predictors in the cloud., *Journal of Network Systems Management* . JCR Impact Factor:1.676(2019).

El Mezouari, A. and Najib, M. (2019). A hadoop based framework for soil parameters prediction., *2019 15th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)* pp. 681 – 687.

Enes, J., Expósito, R. R. and Touriño, J. (2018). Bdwatchdog: Real-time monitoring and profiling of big data applications and frameworks, *Future Generation Computer Systems* **87**: 420 – 437. JCR Impact Factor:5.768(2019).

Enes, J., Expósito, R. R. and Touriño, J. (2020). Real-time resource scaling platform for big data workloads on serverless environments, *Future Generation Computer Systems* . JCR Impact Factor:5.768(2019).

Giménez-Alventosa, V., Moltó, G. and Caballer, M. (2019). A framework and a performance assessment for serverless mapreduce on aws lambda, *Future Generation Computer Systems* **97**: 259 – 274. JCR Impact Factor:5.768(2019).

Gogouvitis, S. V., Mueller, H., Premnadh, S., Seitz, A. and Bruegge, B. (2020). Seamless computing in industrial systems using container orchestration., *Future Generation Computer Systems* **109**: 678 – 688. JCR Impact Factor:5.768(2019).

Kim, J., Jun, T. J., Kang, D. and Kim, D. (2018). Gpu enabled serverless computing framework, *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 533–540. CORE Ranking:C.

MackDioufa, G., Elbiazea, H. and Jaafarb, W. (2020). On byzantine fault tolerance in multi-master kubernetes clusters, *Future Generation Computer Systems* **109**: 407 – 419. JCR Impact Factor:5.768(2019).

Malawski, M., Gajek, A., Zima, A., Balis, B. and Figiela, K. (2017). Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions, *Future Generation Computer Systems* . JCR Impact Factor:5.768(2019).

Mohanty, S. K., Premasankar, G. and di Francesco, M. (2018). An evaluation of open source serverless computing frameworks, *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* pp. 115 – 120. CORE Ranking:C.

Palade, A., Kazmi, A. and Clarke, S. (2019). An evaluation of open source serverless computing frameworks support at the edge., *2019 IEEE World Congress on Services (SERVICES), World Congress on Services (SERVICES), 2019 IEEE* **2642-939X**: 206 – 211. CORE Ranking:B.

Pereira Ferreira, A. and Sinnott, R. (2019). A performance evaluation of containers running on managed kubernetes services., *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* pp. 199 – 208. CORE Ranking:B.

Piotr, D. and Soares, I. (2018). Value-based allocation of docker containers, *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* pp. 358 – 362. CORE Ranking:C.

Savage, N. (2018). Going serverless, *Commun. ACM* **61**(2). JCR Impact Factor:5.410(2019).

Sukhija, N. and Bautista, E. (2019). Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus., *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing* pp. 257 – 262. CORE Ranking:B.

Ueno, M. and Imai, Y. (2020). Ssh kernel: A jupyter extension specifically for remote infrastructure administration, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5. CORE Ranking:B.

Wu, Q., Yu, J., Lu, L., Qian, S. and Xue, G. (2019). Dynamically adjusting scale of a kubernetes cluster under qos guarantee, *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)* pp. 193 – 200. CORE Ranking:B.

Xinjie, G., Xili, W., Baek-Young, C., Sejun, S. and Jiafeng, Z. (2017). Application oriented dynamic resource allocation for data centers using docker containers, *IEEE Communications Letters* (3): 504. JCR Impact Factor:3.457(2019).

Yi, H., Chan, J. and Leckie, C. (2013). Analysing virtual machine usage in cloud computing., *2013 IEEE Ninth World Congress on Services, Services (SERVICES)* pp. 370 – 377. CORE Ranking:B.