

QoS Aware Function Scheduling in Serverless Computing



Deyas Deme Merga

A Thesis Submitted to the department of Computer Science
and Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the
Degree of Master's in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

February 2023

Adama, Ethiopia

QoS Aware Function Scheduling in Serverless Computing

Deyas Deme Merga

Advisor: Dr.-Ing Frezewd Lemma

A Thesis Submitted to the department of Computer Science and

Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the Degree

of Master's in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

February 2023

Adama, Ethiopia

Declaration

I hereby declare that this Master Thesis entitled “**QoS Aware Function Scheduling in Serverless Computing**” is my original work. That is, it has not been submitted for the award of any academic degree, diploma or certificate in any other university. All sources of materials that are used for this thesis have been duly acknowledged through citation

Deyas Deme Merga

Name of student

Signature

Date

Recommendation

I, the advisor of this thesis, hereby certify that I have read the revised version of the thesis entitled “**QoS Aware Function Scheduling in Serverless Computing**” prepared under my guidance by Deyas Deme Merga submitted in partial fulfillment of the requirements for the degree of Master’s of Science in Computer Science and Engineering. Therefore, I recommend the submission of revised version of the thesis to the department following the applicable procedures.

Dr.-Ing Frezewd Lemma

Major Advisor/Supervisor

Signature

Date

Approval Page of M.Sc. Thesis

I/we, the advisors of the thesis entitled “**QoS Aware Function Scheduling in Serverless Computing**” and developed by Deyas Deme Merga, hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis.

Dr.-Ing Frezewd Lemma

Major Advisor/Supervisor

Signature

Date

We, the undersigned, members of the Board of Examiners of the thesis by Deyas Deme Merga have read and evaluated the thesis entitled “**QoS Aware Function Scheduling in Serverless Computing**” and examined the candidate during open defense. This is, therefore, to certify that the thesis is accepted for partial fulfillment of the requirement of the degree of Master of Science in Computer Science and Engineering.

Chairperson

Signature

Date

Internal Examiner

Signature

Date

External Examiner

Signature

Date

Final approval and acceptance of the thesis is contingent upon submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

Department Head

Signature

Date

School Dean

Signature

Date

Office of Postgraduate Studies, Dean

Signature

Date

Table of Contents

List of Tables	iv
List of Figures.....	v
List of Equations.....	vii
List of Abbreviation and Acronyms	viii
ACKNOWLEDGEMENT.....	x
<i>Abstract</i>	xi
CHAPTER ONE.....	1
INTRODUCTION	1
1.1. Background of the Study	1
1.2. Motivation of the Study	3
1.3. Statement of the Problem.....	3
1.4. Research Questions.....	4
1.5. Objective of the Study	4
1.5.1. General Objective	4
1.5.2. Specific Objectives	4
1.6. Significance of the Study	5
1.7. Expected outcome of the study	5
1.8. Scope and Limitation of the Study	6
1.8.1. Scope	6
1.8.2. Limitation	6
1.9. Methodology of the Study	6
1.9.1. Implementation and Design tools.....	7
1.10. Thesis Organization.....	8
CHAPTER TWO.....	9
LITERATURE REVIEW AND RELATED WORKS.....	9
2.1. Overview.....	9

2.2.	Introduction.....	9
2.3.	Cloud Computing Characteristics	11
2.4.	Cloud Computing Service Models.....	12
2.5.	Cloud Computing Deployment Models	13
2.6.	Cloud Computing Architecture.....	18
2.7.	Serverless computing	20
2.7.1.	Serverless Architecture.....	25
2.7.2.	Function-as-a-Service (Faas).....	25
2.7.3.	Serverless Vs FaaS	26
2.7.4.	Serverless Computing Platforms and Frameworks	26
2.7.5.	Serverless Functions	31
2.7.6.	Scheduling functions in serverless Computing	32
2.8.	Related Works.....	34
CHAPTER THREE		40
MATERIALS AND METHODS		40
3.1.	Overview.....	40
3.2.	Research Method	40
3.3.	Research Strategy	40
3.4.	Research Approach	41
3.5.	Data Collection Methods and Tools	41
3.6.	Research Process.....	42
3.7.	Simulation Environment	42
3.8.	Proposed Architecture.....	44
3.9.	Proposed Algorithm.....	46
3.9.1.	Assumptions	47
3.9.2.	Execution time prediction.....	47
3.9.3.	Scheduling	49

3.9.4. Load Balancing.....	53
3.10. Research Metrics	56
3.11. Testing and Methods of Evaluation	57
3.12. Development Tools and Techniques	57
CHAPTER FOUR	59
RESULTS AND DISCUSSIONS	59
4.1. Introduction.....	59
4.2. Implementation of Hybrid Algorithm.....	59
4.3. Experimental setup	59
4.4. Working Environments.....	60
4.5. Experiments and results	60
4.5.1. Experiment One	60
4.5.2. Experiment Two	63
CHAPTER FIVE	70
CONCLUSIONS AND RECOMMENDATIONS	70
5.1. Conclusion	70
5.2. Contribution	70
5.3. Future Works	71
Special Acknowledgment	72
References	73
Appendix A List of sample codes	82
A. Creating Function	82
B. Matrix function	82
C. Prime number Function	83
D. ETLAS Scheduler.....	84

List of Tables

Table 2.1: Summary of related work	38
Table 3.1: The summary of the notations used in the mathematical model	51
Table 4.1: Input, execution times, and invocations percentage of benchmark's functions ..	68
Table 4.2: Results of Experiment for time interval (1s)	68
Table 4.3: Results of Experiment for time interval (2s)	68
Table 4.4: Results of Experiment for time interval (3s)	68

List of Figures

Figure 1.1: Serverless Architecture (Adopted from (Hassan et al., 2021).)	2
Figure 1.2: Research process	7
Figure 2.1: Cloud Computing definition schema	10
Figure 2.2: Essential Characteristics of cloud computing	11
Figure 2.3: Cloud Service Models comparision (Chou, 2018)	12
Figure 2.4: Cloud Computing deployment Models (Sabyrkulov, 2021)	14
Figure 2.5: Public cloud deployment model	15
Figure 2.6: Private cloud deployment model	16
Figure 2.7: Hybrid cloud deployment model	17
Figure 2.8: Community cloud deployment model	17
Figure 2.9: Cloud Computing Architecture (Satyabrata, n.d.)	19
Figure 2.10: Comparison: On premise vs. IaaS vs. PaaS vs. FaaS vs. SaaS	23
Figure 2.11: Serverless platform architecture	25
Figure 2.12: Existing Serverless Platforms and Frameworks	27
Figure 2.13: OpenWhisk Architecture	30
Figure 3.1: OpenWhisk programming model (Apache OpenWhisk;, n.d.)	44
Figure 3.2: Proposed Architecture	45
Figure 3.3: A key-value storing structure for storing a function execution time	49
Figure 3.4: Flow Chart of the proposed algorithm	55
Figure 3.5: Our Scheduling Criteria	57
Figure 4.1: Average waiting times and standard deviations for queued functions with maximum queue length at various average time intervals	61
Figure 4.2: Average waiting times and standard deviations for queued functions with maximum queue length in various average execution times	62
Figure 4.3: Average waiting times and standard deviations for queued actions with a maximum queue length in different workloads in each scheme	63
Figure 4.4: Average waiting times and standard deviations for queued actions and invoker throughput at various time intervals and schemes	64
Figure 4.5: Container startup latency in different states with different sizes of memory in three invocation patterns	65
Figure 4.6: Average waiting times and standard deviations for actions in the queue and throughput of the invoker in each scheme	66

Figure 4.7: Rate of containers states in each scheme 67

Figure 4.8: Rate of resource utilization in each scheme..... 67

List of Equations

Equation 3-1	47
Equation 3-2	53
Equation 3-3	53
Equation 3-4	53
Equation 3-5	53
Equation 3-6	54

List of Abbreviation and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Service
BaaS	Backend as a Service
CPU	Central Processing Unit
CSP	Credential Service Provider
DAG	Directed acyclic graph
EC2	Elastic Compute Cloud
FaaS	Function as a Service
GCP	Google Cloud Platform
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated development environment
IT	Information Technology
LSTM	Long short-term memory
ML	Machine Learning
NIST	National Institute of Standards and Technology
NOAH	Non-cooperative Online Allocation Heuristic
PaaS	Platform as a Service
PASch	Package-Aware Scheduling
PAYG	Pay-as-you-go
QoS	Quality of Service

RAM	Random-Access Memory
S3	Simple Storage Service
SaaS	Software as a Service
SNS	Simple Notification Service
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UML	Unified Modeling Language
VM	Virtual Machine
VS	Visual Studio

ACKNOWLEDGEMENT

First and foremost, I want to express my gratitude to God for everything that has happened, for all his blessings and for helping me through all the challenges and the hard times that I could not face on my own, thank you very much dear Lord.

My heartfelt thanks also go to Dr.-Ing. Frezewd Lemma, my advisor, whose genuine advice, helpful comments, observations, and dedication made this study possible. In my study sessions, as a counselor and instructor, he was the one who showed me the way and helped me to overcome my obstacles. I owe him great respect and gratitude for this valuable guide.

Additionally, I would like to express my truthful thanks to my family who have supported me along the way. My special thanks go to Degefa Guluma and Sifen Tulu; I will never forget the help of your support. The Department of Computer Science and Engineering, the postgraduate coordinator, and the entire Astu community who involved directly and indirectly deserve my respect and appreciation as well.

Finally, many thanks to all my friends and all participants who participated in the study and made this research possible.

Abstract

It is challenging when users manage cloud environments, such as availability, load balancing, auto-scaling, monitoring, etc. These difficulties have driven the development of a new cloud computing approach known as serverless cloud computing. The serverless model differs from other computing models by shifting the responsibility for server management entirely to the provider, effectively making the model serverless from a developer's perspective. Many applications have begun to employ serverless computing platforms in recent years, owing to their ease of deployment and cost-effectiveness. However, traditional serverless platform scheduling algorithms fall short of responding to the particular characteristics of such programs, which include burstiness, short and unpredictable execution periods, statelessness, resource utilization, system throughput, cold start rate, and QoS satisfaction. The existing techniques, in particular, fall short of addressing the needs imposed by the combined effect of these characteristics: scheduling millions of function invocations per second while maintaining predictable performance. To address these difficulties, we propose an execution time and load aware scheduler (ETLAS) that schedules function for Serverless computing. It is a hybrid scheduling discipline that determines the order of function executions based on the function's predicted execution time and arrival time, which has a notable impact on worker node latencies and throughputs. Every worker node's queuing delay is precisely calculated to estimate how many containers are required, and if there are not enough containers to handle all the queued requests, reactive container spawning is used to prevent SLO violations caused by queuing delays. We implemented ETLAS in Apache OpenWhisk and show that ETLAS outperforms others by reducing average waiting time by 34% and increasing throughput by 42% compared to the OpenWhisk worker scheduler and multiple queue scheduling schemes.

Keywords: *Serverless computing, Function scheduling, Predictive Scheduling, Quality of Service, Function as a Service.*

CHAPTER ONE

INTRODUCTION

1.1. Background of the Study

Cloud computing is a growing trend in distributed computing and information technology (IT). With the emergence of virtualization in software and hardware infrastructure, cloud computing emerged and cloud providers started using it to provide services to customers(Boza et al., 2018; Großmann et al., 2019). Customers can use the Internet to access these cloud services. Due to the benefits of cloud technologies, such as scalability, availability, and flexibility, software developers have begun to use them in their software solutions(Villamizar et al., 2017).

Software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) are the three primary categories of cloud computing depending on the provision of services. In the SaaS category, cloud providers offer different types of software as a service to users. For example, Google offers many apps as a service (e.g., Gmail, Google docs, Google Sheets, and Google Forms). In this type of cloud, the user is not responsible for the development, implementation, and management of the services. Users here just use them regardless of settings, configurations, etc.

Managing cloud services is a challenging task. The authors of (Jonas et al., 2019) addressed several challenges when users manage cloud environments, such as availability, load balancing, auto-scaling, security, monitoring, and more... For example, cloud users must ensure the availability of services that if a machine failure occurs, it will not affect all services. In addition, he should consider distributing copies of the services geographically to protect them in the event of a disaster. Another challenge is a load balancing; in which, cloud users must ensure that demand across services is balanced to use all resources efficiently.

These difficulties have driven the development of a new cloud computing approach known as serverless cloud computing(Jonas et al., 2019). As demonstrated in Fig. 1.1, serverless cloud computing provides backend as a service (BaaS) and function as a service (FaaS). Storage, messaging, user management, and other services are included in the BaaS. The FaaS, on the other hand, allows developers to deploy and run their programs on several computing platforms. The FaaS relies on the services provided by the BaaS such as a

database, messaging, user authentication, etc. The FaaS approach, commonly known as "event-driven functions," is the most widely used serverless model (Geng et al., 2018; Kulkarni et al., 2019).

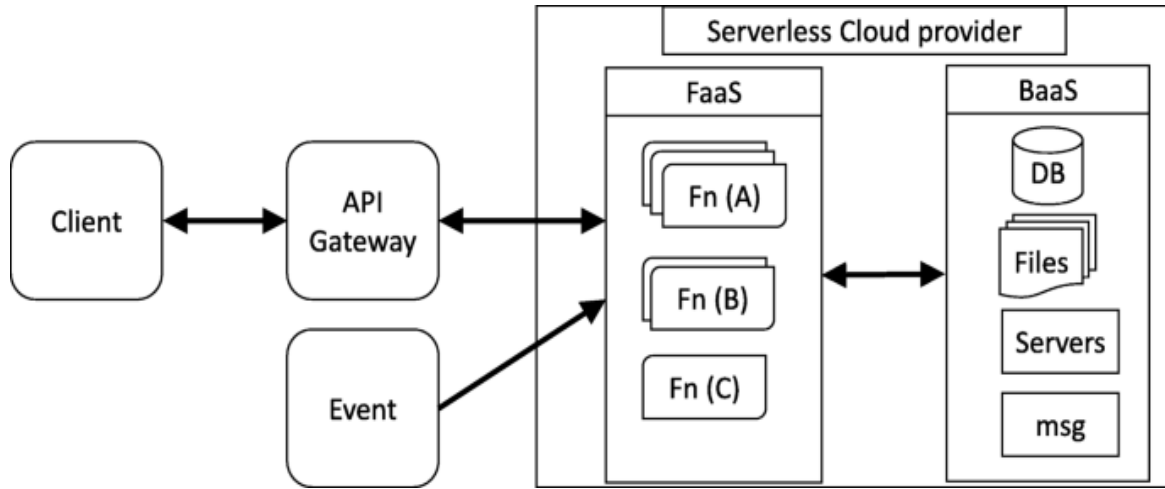


Figure 1.1: Serverless Architecture (Adopted from (Hassan et al., 2021).)

Amazon Lambda was the first to propose the serverless cloud paradigm in 2014, and cloud companies like Google and Microsoft adopted it in 2016. Serverless cloud computing adds a new abstraction layer to existing cloud computing paradigms, abstracting server-side management away from developers (Baldini et al., 2017). Developers can concentrate on the application logic rather than server-side management and configurations with the serverless paradigm. Developers, for example, upload their programs as functions to the serverless cloud (see Fig. 1.1). The cloud provider is then responsible for monitoring, scaling, and delivering various resources to ensure that these functions execute smoothly (Adzic & Chatley, 2017; Jambunathan & Yoganathan, 2018).

Serverless computing is a new paradigm that enables companies to design and deploy applications efficiently without having to worry about the underlying infrastructure (Lloyd et al., n.d.). AWS Lambda, Azure Functions, and Google Cloud Functions are some of the serverless computing technologies that have been proposed to the market. Such platforms allow developers to concentrate solely on business logic, leaving the infrastructure monitoring, provisioning, scaling, and management to cloud service providers (Baldini et al., 2017). Developers just need to deploy the source code of short-running functions and create triggers for executing them when utilizing FaaS. On-demand, the FaaS provider runs and bills functions as isolated instances, scaling their execution. We will refer to FaaS as "Serverless Functions" for the rest of this paper.

1.2. Motivation of the Study

The serverless model is distinguished from other computing models by the entire transfer of server management responsibilities to the vendor, effectively rendering the model 'serverless' from the developer's perspective. The practice of properly managing server resources for the execution of applications is a critical component of server management. In a serverless environment, resource management refers to the general aspect of efficiently managing the resource requirements of an application workload and the available system resources with minimal user intervention. Because of the autonomic nature of the expected resource management process in these contexts, each stage of the process requires special attention for improved application and system performance (Mampage et al., 2021).

When employing a serverless deployment architecture, developers prefer to do as little effort as possible. As a result, rather than needing to specify a resource configuration and other application characteristics when deploying an application, a serverless platform will be ideal if it could infer application characteristics using workload modeling and prediction techniques. An effective strategy for understanding an application's properties allows for the development of better resource allocation and scaling approaches, which aid in meeting user QoS needs.

1.3. Statement of the Problem

Many applications have begun to employ serverless computing platforms in recent years, owing to their ease of deployment and cost-effectiveness. However, traditional serverless platform scheduling algorithms fall short of responding to the particular characteristics of such programs, which include burstiness, short and unpredictable execution periods, statelessness, resource utilization, system throughput, cold start rate, and QoS satisfaction. The existing techniques, in particular, fall short of addressing the needs imposed by the combined effect of these characteristics: scheduling millions of function invocations per second while maintaining predictable performance.

Serverless computing platforms, such as AWS, are known for their ability to manage a wide range of workload requests with ease. However, in order to maintain high resource efficiency, platform capacity (containers and hosts) must scale out and scale elastically in response to workload requirements, including bursts of function requests. This, unfortunately, can lead to a lot of cold starts. Furthermore, traditional load balancing systems

such as Round Robin or Least Connections are meant to distribute the load among available servers, and hence naturally tend to use all available hosts, thus leading to underutilization. Because of the wide range of serverless function execution and the lack of predicting future load, scheduling in serverless cloud computing is more difficult than in typical cloud services. In addition, each worker node is responsible for managing the resources by allocating the appropriate resources to the function sandbox. The rising amount of service requests in the system, as well as the overhead of moving data around, indicates that the platform is becoming overloaded. The need of making optimum use of these platforms is critical for both consumers and cloud providers.

1.4. Research Questions

This study aimed to answer the following two important research questions

RQ1. How to reduce response times of functions in serverless to meet user's QoS requirements?

RQ2. How to minimize the provider's capital expenses while elastic scaling of the serverless platform capacity?

1.5. Objective of the Study

1.5.1. General Objective

The general objective of the proposed study is to develop a QoS-aware function scheduling algorithm for the serverless computing environment.

1.5.2. Specific Objectives

In order to achieve the general objective, the following specific objectives were addressed:

- ✧ Studying the problems of function scheduling in serverless computing environments.
- ✧ Conducting a literature review on function scheduling in serverless computing
- ✧ Designing an efficient function scheduling algorithm that determines the workload of providers and QoS to the end users for serverless computing environments.
- ✧ Simulating the proposed algorithm in the simulation setting
- ✧ Comparing peer algorithms with the result of the proposed approach in terms of selected measurement metrics.

1.6. Significance of the Study

Scheduling workloads to appropriate host nodes based on resource requirements while efficiently utilizing existing resources is a significant task that both developers and cloud providers or system owners are concerned with. When the resource demand exceeds the available resource capacity, scheduling also entails deciding the order in which applications are executed. While the developer may demand certain QoS assurances, it is critical for the provider to properly manage resources, which is one of the key goals of resource management. In this study, we propose QoS Aware Function Scheduling approaches that benefit both providers and customers by reducing function response times through lowering average waiting times and enhancing worker node throughputs and resource utilization. As a result, the proposed research primarily and considerably benefits the following stakeholders.

End users: End users can quickly access numerous cloud resources over the Internet with pay-as-you-go pricing for functions. These user expectations are met with a good scheduling practice that improves response time and decreases the average waiting time to execute.

Cloud providers: Cloud providers can simply manage a cloud resource and become more cost-effective as a result of appropriate function scheduling and balancing the system's overall burden.

Future researchers and academicians: Researchers and academicians can use the proposed technique to optimize it further. It is also utilized for research and as an academic reference at the university level.

1.7. Expected outcome of the study

The proposed study schedules functions for worker nodes on serverless platforms in order to improve function response times by lowering average waiting times and enhancing worker node throughputs and resource utilization. As a result, quality of service (QoS) should be guaranteed.

1.8. Scope and Limitation of the Study

1.8.1. Scope

Management of resources is a challenging task in a distributed environment. The proposed approach is focused on scheduling function requests in serverless computing environment. The main objective of the study is to develop a second level scheduling method for worker nodes on serverless platforms to improve worker node throughput and resource usage while lowering average waiting times and reducing response times as well as cold starts for function requests. Hence, the user's requirements for service quality should be satisfied.

1.8.2. Limitation

The primary goal of this work is to schedule the incoming request or function in a serverless computing environment and assure the quality of service. However, other quality of services rather than response time, throughput, cold start, and resource utilization is not considered in addition to container scheduling and service migration due to budget, deadline, and limitations of resources. The experiment of the proposed approach is performed only on Apache OpenWhisk serverless simulation framework rather than the real environment.

1.9. Methodology of the Study

In conducting this research, we thoroughly examined the existing knowledge in this field. Book reviews, journal articles, research papers, topic reports, and other online documents were reviewed to gain insight into previously explored methodologies and approaches to function scheduling in serverless cloud computing environments. Function scheduling in serverless cloud computing environments should result in quality of service (QoS) improvements in terms of response time, throughput, cold start, and resource utilization. In general, the techniques and methods described below were used to achieve the above objectives and to answer the research question.

To attain the objectives of the study and to answer the research questions mentioned in section 1.4 of this document, the following methods and techniques will be followed. *Figure 1.2* depicts the research process of this study. We have followed the following steps as a research methodology.

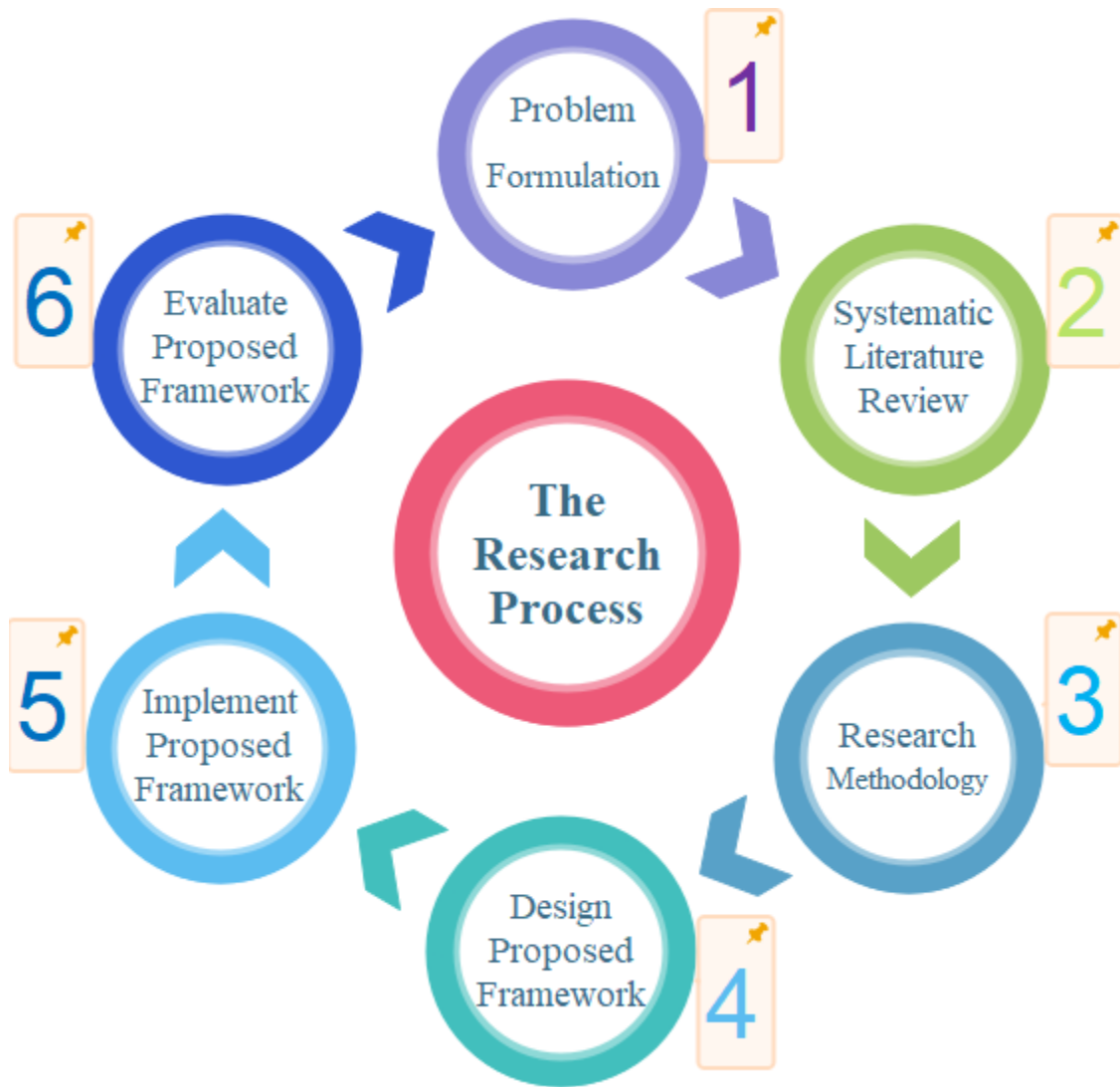


Figure 1.2: Research process

1.9.1. Implementation and Design tools

Visual Studio Code as a development environment was chosen as the best tool to implement the proposed solution. Visual Studio Code has many advantages over any other IDE; like cross-platform support, lightweight, robust architecture, intelli-sense and freeware. VS Code includes repository extensions and support. Therefore, we can connect to any other repository to retrieve or store the instances and also download the extension to use additional programming languages. Apache OpenWhisk Framework packages are integrated with Kubernetes and Helm for simulation, Edraw Max and Drawio are used as design tools. Edraw is a design tool that software developers can use to create diagrams such as flowcharts, workflows, and UML diagrams. Also, Drawio is a kind of design tool that allows

the software designer to reliably create and publish many types of diagrams to represent an idea that is freely available online for drawing.

Hardware Requirements

The hardware requirements we used to conduct this research are listed below.

- ✧ Processor: Intel(R) Core (TM) i7-7500U CPU @ 2.70GHz 2.90GHz.
- ✧ RAM: Installed memory (RAM) 8.00GB.
- ✧ System Type: 64-bit Operating System, x64 based processor.
- ✧ Hard Disk: 1TB

1.10. Thesis Organization

The first chapter discussed the introduction of the study, the motivation and problem statement, the research questions answered in the proposed solutions, the scope and limitation, the aim of the study, the methodology and the beneficiaries of the study results. Chapter two deals with the literature review. It includes a systematic literature review, an overview of cloud computing, cloud computing characteristics, services, and deployment models, architecture, serverless computing, platforms and frameworks, scheduling capabilities, and related work. Chapter three covers the methodology of a proposed solution, which includes a method for the proposed work, a data collection method, development methods and tools, techniques, a simulation environment, and research metrics, designing the proposed solution, which includes the proposed solution and its architecture, and also the algorithm in detail as well as the experimental setup of the study. Chapter four presents the results and analysis as well as the performance evaluation and discussion of the proposed solution. The final section consists of a conclusion and a recommendation and identifies future work for further research direction on this research topic.

CHAPTER TWO

LITERATURE REVIEW AND RELATED WORKS

2.1. Overview

This chapter provides a basic foundation for the study's investigation by reviewing the literature on Cloud Computing. The theoretical background of cloud computing is the focus of this chapter. It begins with a definition of Cloud Computing, followed by a discussion of its fundamental properties, as well as enabling technologies that aid in comprehending the Cloud Computing phenomena. Then explores major Cloud computing service models and deployment strategies, as well as the advantages and drawbacks of each. In addition, the benefits that drive Cloud Computing adoption, and the concerns that may affect the adoption are outlined. The Serverless framework along with its benefits and drawbacks is also discussed. Furthermore, the cloud's Serverless Computing paradigm and Scheduling functions are discussed. Finally, a summary of related works on Serverless computing function scheduling is presented.

2.2. Introduction

With the growing demand for automation, artificial intelligence, augmented reality, smart homes and cities, real-time analytics, games, and other industrial and consumer applications, the number of the audience connected to the Internet continues to grow. The global cloud computing industry is expected to grow at a compound annual growth rate (CAGR) of 15.7% from 2022 to 2030, worth \$368.97 billion by 2021(*Cloud Computing Market Size Report, 2022-2030*, n.d.). Emerging technologies such as artificial intelligence (AI) and machine learning facilitate the growth of the cloud by allowing companies to harness the capabilities of artificial intelligence. The COVID19 pandemic has become a significant challenge for the global economy. Cloud computing can help businesses overcome business continuity challenges caused by lockdowns, working from home, social distancing, and other market impact requirements. Remote working has become the latest trend, and it is predicted to remain in the long future, as organizations, managers, and employees continue to prefer to work remotely as a result of the pandemic.

An important aspect of cloud computing is its utility-oriented approach. This approach allows you to access technology services such as databases and online storage, rent virtual

hardware, and use necessary development platforms from cloud providers. Instead of buying, building, and maintaining a physical data center and/or investing in complicated IT infrastructure, you simply enter your credit card information and access the service via your web browser, paying only for the actual use of the service with no or minimum up-front expenditures

Cloud computing uses a pay-as-you-go (PAYG) paradigm to provide businesses with virtualized computer resources on demand. Technology is accessed through a cloud vendor or provider rather than owning and maintaining real computer assets such as servers and data centers. Cloud computing solutions are provided by cloud vendors such as AWS, Azure, and GCP. These cloud vendors are used by businesses to outsource their IT services. Virtual machines and serverless infrastructure, as well as data storage and databases, are all examples of cloud-based resources(What Is Cloud Computing? Complex Guide for 2022, n.d.).

Cloud computing is a paradigm that provides on-demand, convenient, ubiquitous network access to a pool of shared configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be provided and released quickly with minimal administrative effort or service provider involvement. Five key qualities, three service models, and four deployment models make up this cloud model (Mell & Grance, n.d.)

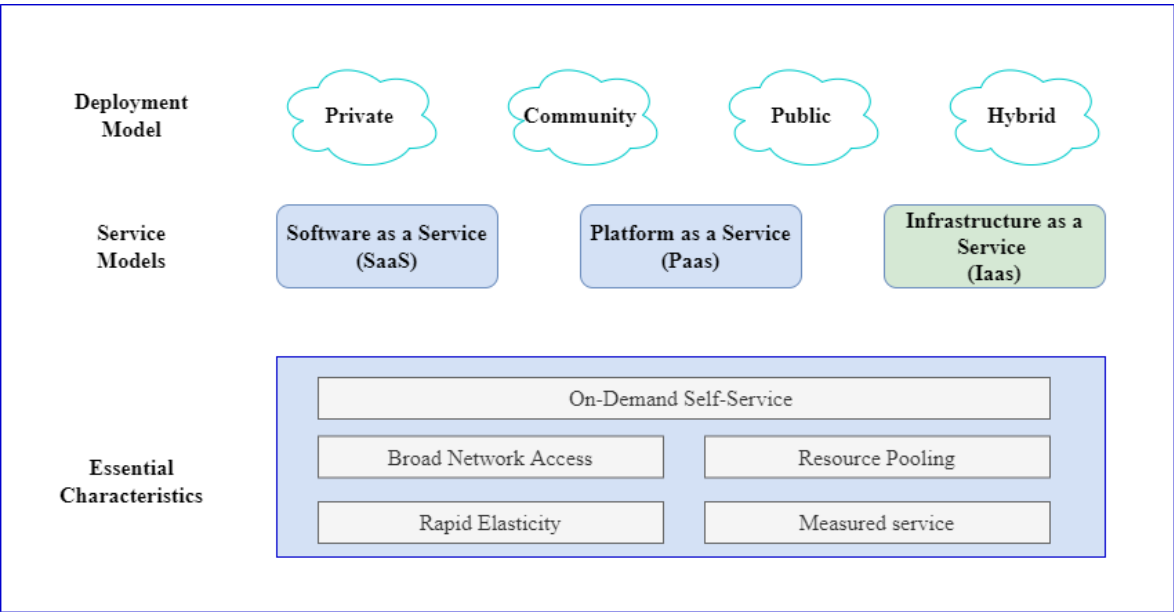


Figure 2.1:Cloud Computing definition schema

2.3. Cloud Computing Characteristics

According to National Institute of Standards and Technology (NIST) definition there are five essential characteristics of cloud computing: measured service, rapid elasticity, resource pooling, on demand self-service and broad network access(Mell & Grance, n.d.).



Figure 2.2: Essential Characteristics of cloud computing

- ✧ **On-Demand Self-Service:** The capability of the user to solely control the service model to get and release services as required without any human supervision, greatly improves the cost efficiency of the services by reducing the number of workforces desired.
- ✧ **Broad Network Access:** refers to the availability of the resources to a wide range of computing devices such as computers, cell phones, and tablets regardless of their platform through standardized network interfaces.
- ✧ **Resource Pooling:** refers to the characteristic that the CSP can use a multi-tenant model to pool physical and virtual resources to meet the need of multiple cloud users simultaneously by using a method called virtualization. The geographical locations of the resources are hidden from the user, generally.
- ✧ **Rapid Elasticity:** refers to the automated process that enables the services to be expanded and contracted to serve the required needs of the customer by instantly provided if the demand rises and released if the demand drops.

- ✧ **Measured service:** The services used are measured by metering capabilities and parameters such as storage, processing, and bandwidth to determine the actual usage (Calabrese & Cannataro, 2015). The measured usage can be used for charging or for controlling the allocation of the resources (Mell & Grance, n.d.)

2.4. Cloud Computing Service Models

Cloud services are services that will be provisioned to customers over the internet allowing easy access to applications, storage, and processing resources without deep knowledge of internal infrastructure. Services given by the cloud are more helpful in reducing cost and scaling resources with more flexibility to provide it only on demands. NIST cloud computing definition defines three possible categories of services as PaaS, SaaS, and IaaS (Moravcik et al., 2018).

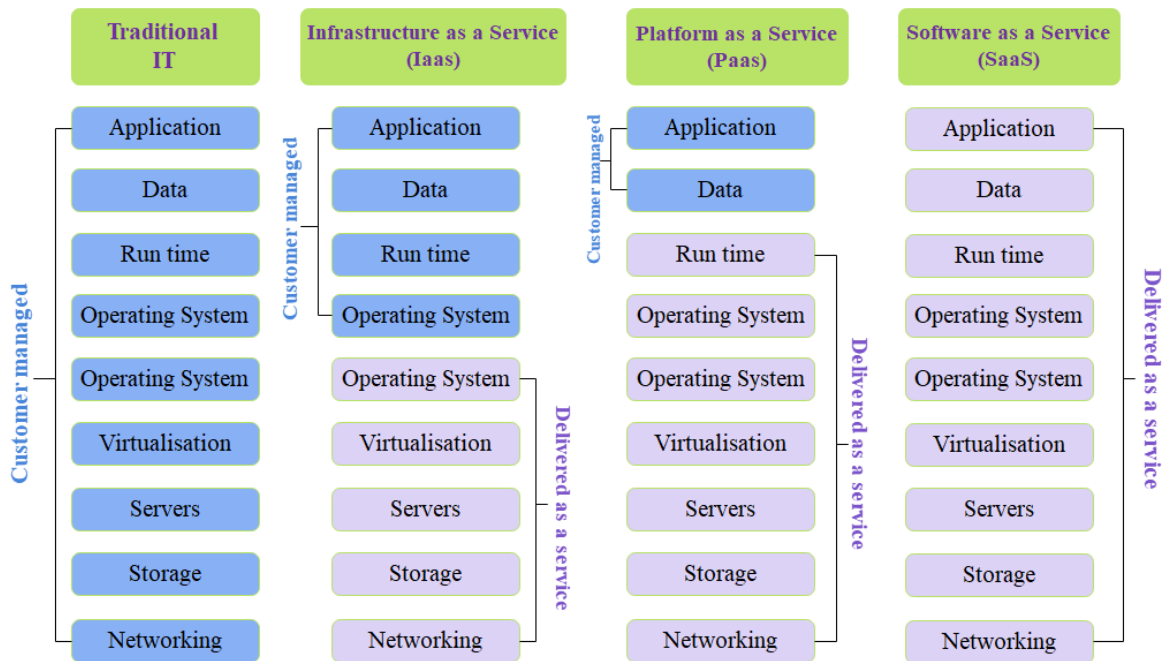


Figure 2.3: Cloud Service Models comparison (Chou, 2018)

Software as a Service (SaaS)

In this model, a complete application is offered to the customer, as a service on demand. A single instance of the service runs on the cloud & multiple end users are serviced. On the customers' side, there is no need for upfront investment in servers or software licenses, while for the provider, the costs are lowered, since only a single application needs to be hosted & maintained. Today SaaS is offered by companies such as Google, Salesforce, Microsoft, Zoho, etc (Sen, 2013).

Platform as a Service (Paas)

A layer of software, or development environment, is encapsulated and supplied as a service, allowing other higher levels of service to be created on top of it. The customer has complete control over the development of his own applications that run on the provider's infrastructure. PaaS providers offer a predefined combination of OS and application servers, such as LAMP platform (Linux, Apache, MySQL and PHP), restricted J2EE, Ruby and etc. to fulfill the manageability and scalability requirements of applications. Google's App Engine, Force.com, etc are some of the popular PaaS examples (Sen, 2013).

Infrastructure as a Service (IaaS)

Basic storage and computing capabilities are provided as standardized services across the network with IaaS. Servers, storage systems, networking equipment, data center space, and other resources are pooled and made accessible to meet workload demands. Typically, the customer would deploy his own software on the infrastructure. Amazon, GoGrid, 3 Tera are some common examples (Sen, 2013).

2.5. Cloud Computing Deployment Models

A cloud deployment model is defined according to where the infrastructure for the deployment resides and who has control over that infrastructure (Theory & Rountree, 2013). Each cloud deployment model satisfies different organizational needs, so it is important that you choose a model that will satisfy the needs of your (Theory & Rountree, 2013). Based on the NIST definition, (Mell & Grance, n.d.) there are four deployment types, which are outlined as follows:

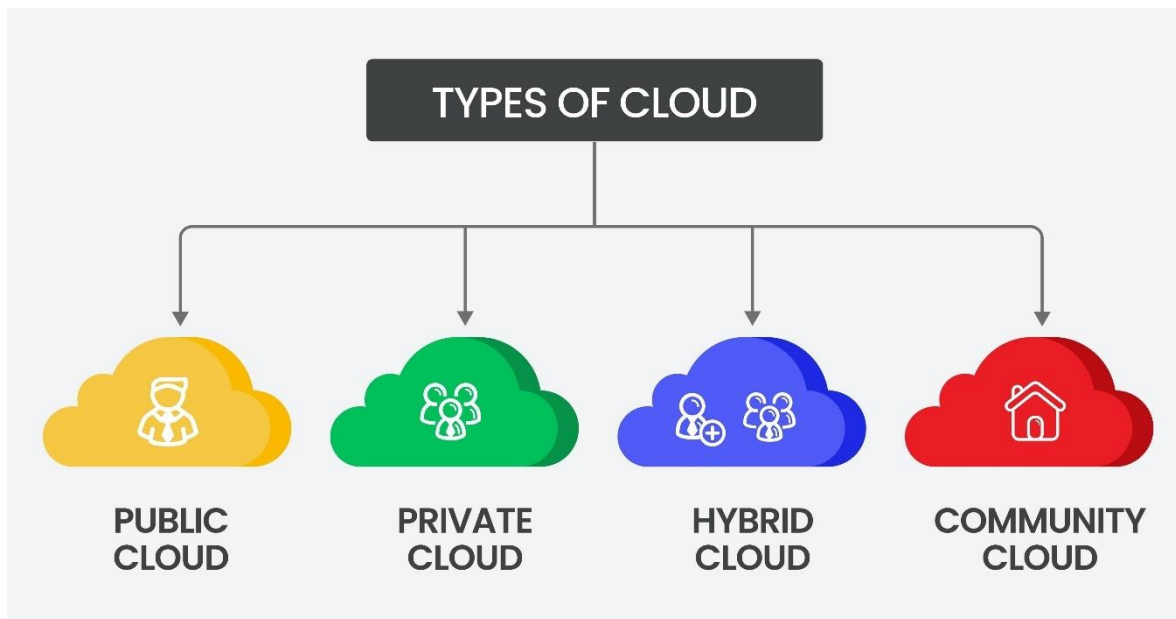


Figure 2.4: Cloud Computing deployment Models (Sabyrkulov, 2021)

Public Cloud

Third parties deliver public clouds over the Internet, with users sharing the services and infrastructure provided by public clouds and accessing them using web browsers(*What Is Cloud Computing? Complex Guide for 2022*, n.d.). In 2022, the global public cloud computing market is worth around \$490 billion and is expected to reach nearly \$600 billion by 2023 (Gartner:, n.d.). The fundamental difference between a public cloud and a private cloud is that it is a multi-tenant environment, which means that companies or cloud tenants use shared public cloud computing resources, such as devices, networks, memory, and other hardware. Public cloud services are hosted and maintained by third-party vendors or providers. Storage, servers, and the network connection between them are among the services that make up the hardware layer. Additional services include system management, maintenance, and updates. In addition, the public cloud provides various services to support application development.

As a result, businesses who use public clouds don't have to invest in expensive storage. They also don't have to worry about server space, maintenance, or the expenditures that go along with it. A public cloud is analogous to a black box in terms of hardware setup from the user's perspective. Connecting via a simple user interface (UI) to consume power and space is a simple way to create new resources. Public clouds operate on a pay-as-you-go (PAYG) model, which means you only pay for the resources you use when you need them(*What Is Cloud Computing? Complex Guide for 2022*, n.d.).

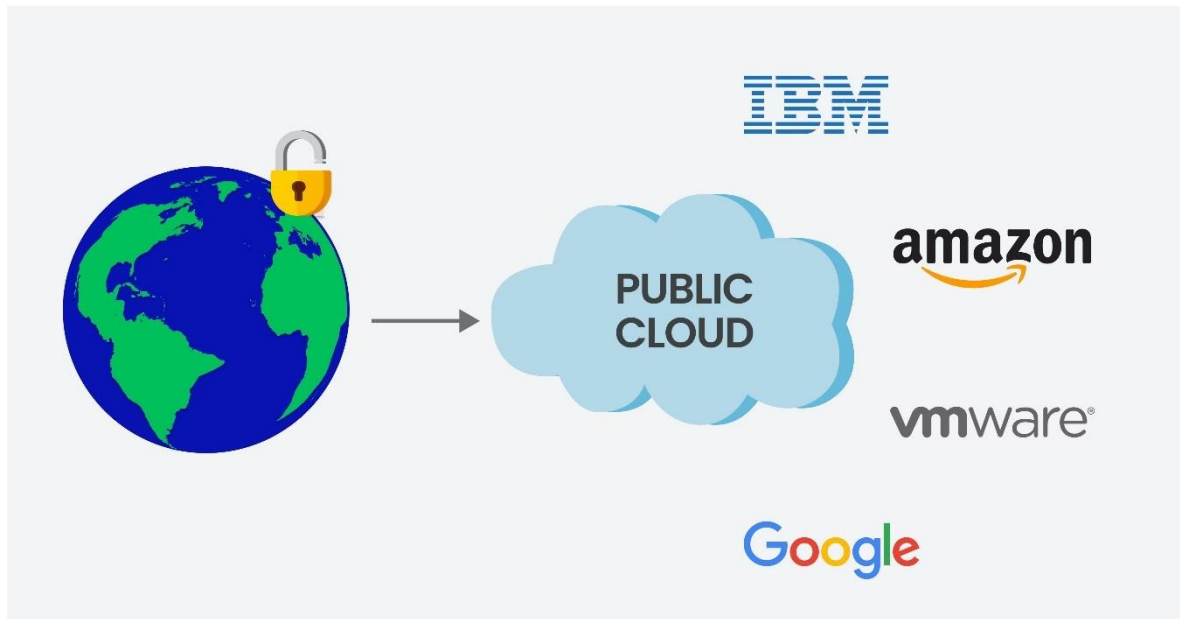


Figure 2.5: Public cloud deployment model

Private Cloud

These work in a single-tenant environment, allowing organizations to have restricted access. A private cloud is dedicated to a single business that uses it exclusively. As a result, resources aren't shared across organizations, which is the major difference between a private cloud and a public cloud. A private cloud can't be accessed via the internet. That's why, when it comes to regulatory compliance and security, they're frequently the cloud of choice for financial and government businesses. Big businesses that require a lot of computer power may find it more cost-effective to build their own data centers rather than renting farms of computing power. Approximately 41% success rate of a company's workloads are run in a private cloud. Private clouds can be hosted and managed in a variety of ways, and they can perform a variety of services depending on the needs of the company.

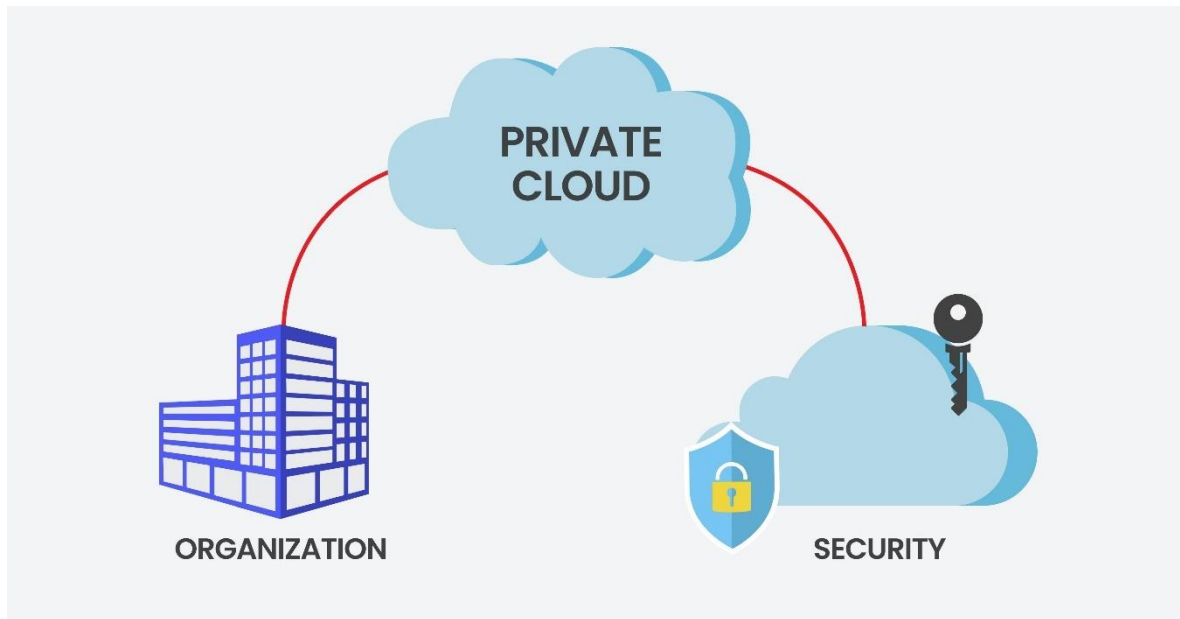


Figure 2.6: Private cloud deployment model

❖ ***Internal private cloud***

Hosted at the premises of a business, there's an internal IT department managing and operating internal private clouds. That means the company needs to purchase servers; it's also down to them to keep them running and maintain the software.

❖ ***Hosted private cloud***

With hosted private clouds, the servers are not located on the company's physical premises. On the contrary, they are hired but completely dedicated to the business. Therefore, it is not necessary to build an on-site data center.

❖ ***Third-party management***

This is a hosted environment in which the provider manages every aspect of the cloud for the organization, including the deployment of additional services like identity management and storage. This is a good alternative for companies who don't have the resources to run private cloud systems on their own.

Hybrid Cloud

In a hybrid cloud, public and private clouds are combined in the same company. This suggests there is at least one private cloud. Hybrid clouds combine the advantages of both types of clouds. A company that uses a hybrid cloud connects public and private clouds and

exchanges data between them. Non-critical operations are performed in the public cloud, whereas crucial and sensitive data and activities are performed in the private cloud.

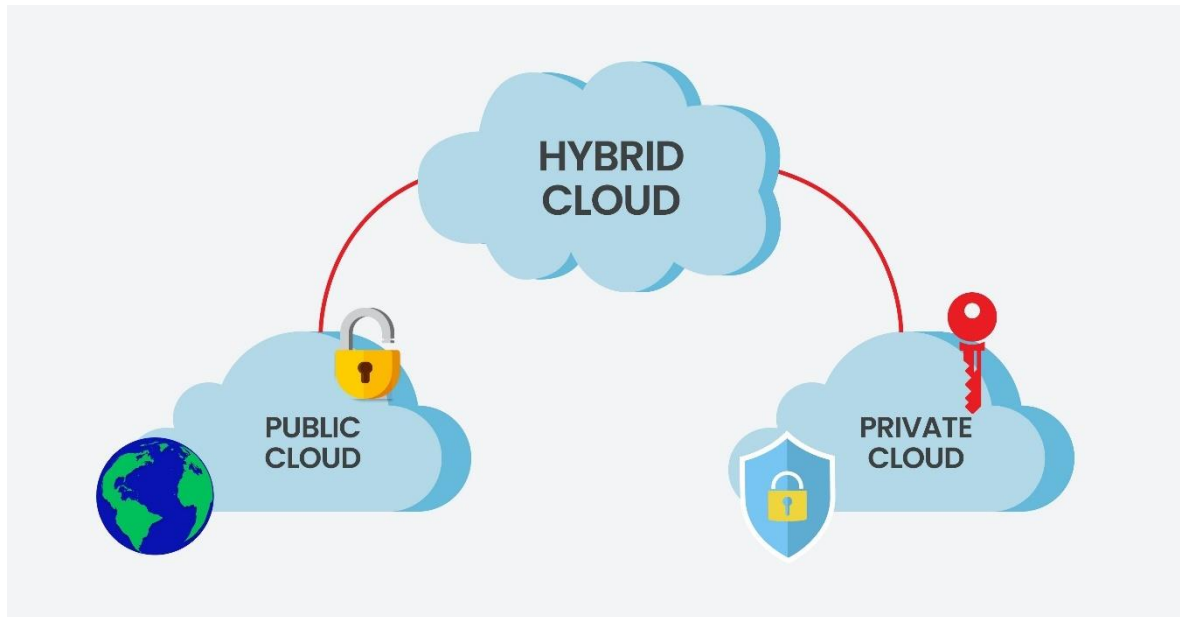


Figure 2.7: Hybrid cloud deployment model

Community Cloud

A restricted number of individuals or enterprises can benefit from this deployment approach, which provides a cloud computing solution. All members, or a third-party managed service provider, are responsible for governing, managing, and securing a community cloud.



Figure 2.8: Community cloud deployment model

❖ **Distributed Cloud**

What you have here is the distribution of public cloud services to different physical locations. However, operation, governance, and updates are the domain of the public cloud provider.

❖ **Multi-cloud**

Multi-clouds are similar to hybrid clouds. The distinction is that businesses use more than one public cloud provider alongside a private cloud. Multi-clouds boast all the advantages of a public cloud, with services deployed by two or more vendors. They reduce reliance on a single provider, increasing flexibility and mitigating against disaster. Large organizations with complex setups usually lean towards multi-clouds.

❖ **Poly Cloud**

This model makes use of various public clouds to take advantage of different suppliers' particular offerings. A poly cloud differs from a multi-cloud in it is not designed to boost flexibility or minimize disasters; rather, it allows a company to accomplish more than it could with a single provider.

❖ **Big Data Cloud**

Transferring huge data to the cloud and maintaining security used to be a challenge, but with so much data now coming from the cloud, big data adoption is now possible. The cloud, for example, can be used for corporate analytics and geographical analysis.

❖ **HPC Cloud**

This refers to the usage of cloud computing services and infrastructure for high-performance computing (HPC) applications that require a lot of memory and processing capacity. Traditionally, such applications were run on computer clusters.

2.6. Cloud Computing Architecture

The components and subcomponents required for cloud computing are known as cloud computing architectures. Front-end platforms (fat client, thin client, mobile), back-end platform (server, storage), cloud-based network, and delivery are the most common components (Internet, Intranet, Internet). These elements come together to form a cloud computing architecture. The frontend and the backend are the two sections of the Cloud

Computing Architecture and they exchange information through a network or the Internet. The cloud computing architecture is depicted in the diagram below:(Satyabrata, n.d.)

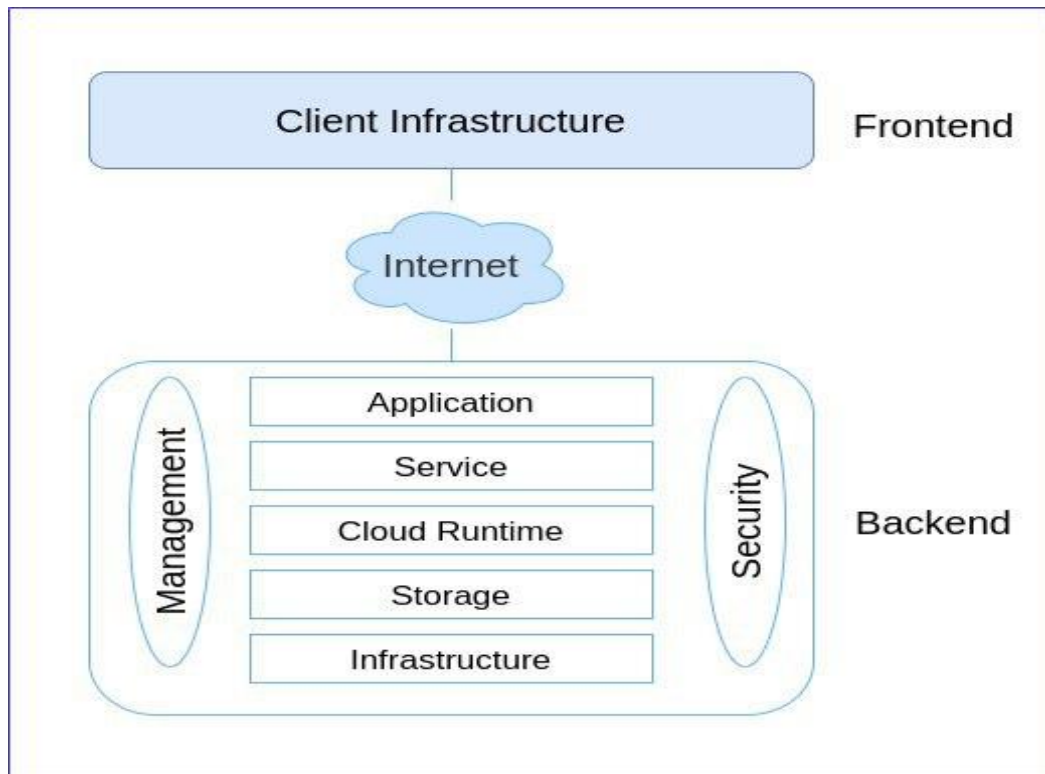


Figure 2.9: Cloud Computing Architecture (Satyabrata, n.d.)

Cloud computing architecture is a combination of SOA (Service Oriented Architecture) and EDA (Event Architecture). Client, application, service, runtime, storage, infrastructure, management, and security infrastructure are all components of the cloud computing architecture.

❖ Frontend

The client-side of the cloud computing system is referred to as the front end of the cloud architecture. It encompasses all of the user interfaces and applications for accessing cloud computing resources and services. For example, use a web browser to access the cloud platform.

Client Infrastructure: The frontend components are referred to as client infrastructure. It includes all of the applications and user interfaces needed to use the cloud platform.

❖ Backend

Backend refers to the cloud itself used by the service provider. It contains resources, manages resources, and provides security mechanisms. With that, it includes massive storage capacity, virtual applications, virtual machines, traffic control mechanisms, deployment models, and more.

Application: The backend application refers to the software or platform accessed by the client. This means that it provides services in the backend according to customer requirements.

Service: Backend service refers to the three main types of cloud-based services such as SaaS, PaaS, and IaaS. Also, manage the type of service the user is accessing.

Cloud Runtime: The runtime cloud in the backend refers to providing the platform/runtime environment and runtime for the virtual machine.

Storage: Backend storage refers to the storing and management of data in a flexible and scalable manner.

Infrastructure: Cloud infrastructure in the backend refers to the hardware and software components of the cloud, it includes servers, storage, network devices, virtualization software, etc.

Management: Backend management refers to the management of backend components such as applications, services, runtime clouds, storage, infrastructure, and other security mechanisms.

Security: Backend security refers to the implementation of various security mechanisms in the backend to secure resources, systems, files, and cloud infrastructure for end-users.

Internet: The internet connection serves as a medium or a bridge between the frontend and the backend, allowing for interaction and communication between the two.

2.7. Serverless computing

With the availability of virtual machines (VM) on-demand, cloud computing in general, and Infrastructure-as-a-Service (IaaS) in particular, have become widely recognized and used computing paradigms(Castro et al., 2020). According to Gartner, enterprise IT groups that can transition to cloud application and infrastructure software, business process services, and system infrastructure will spend 51 percent of their IT budgets on the cloud by 2025(Cooney,

2022). A key factor in the growing adoption of the cloud by enterprise IT is its pay-as-you-go model in which the customer only pays for the resources rented from the cloud provider and has the flexibility to have to get as many resources as needed with no initial cost (elasticity). Unfortunately, scaling was left to developers and system designers, who often used overprovisioning approaches to deal with unexpected spikes in service requests. Studies of cloud resource use in datacenters reveal a significant disparity between the resources that cloud clients allocate and pay for (lease VMs) and actual resource utilization(Castro et al., 2020).

Serverless computing is emerging as an attractive new model for deploying cloud applications, largely due to the recent shift in enterprise application architecture to containers and microservices(Castro et al., 2020; Chaudhary et al., 2017). Serverless enables pay-as-you-go without the need for additional effort to start and stop the server and brings cloud computing closer to its original vision of being considered as utilities. Serverless computing allows developers to achieve cost savings and scalability without requiring a high level of cloud computing expertise, which is time-consuming to acquire(Castro et al., 2020).

Serverless computing is growing in popularity due to its simplicity and cost savings, as evidenced by the increasing percentage of Google Trends for the search term "serverless". By 2026, according to a new report from Reports and Data, the market is expected to reach \$31.53 billion. Most of the major cloud providers, including Amazon, IBM, Microsoft, Google, and others, have released serverless computing with several initiatives. complementary open source led by businesses and academic institutions(Castro et al., 2020).

The serverless paradigm shift presents both an opportunity and a risk from the perspective of an IaaS customer. On the one hand, it offers developers a simplified programming approach for building cloud applications that abstract away the majority, if not all, operational issues. They no longer have to be concerned with issues such as availability, scalability, fault tolerance, over/under-provisioning of VM resources, server management, and other infrastructure concerns. They can concentrate on the commercial aspects of their applications instead. By charging for execution time rather than resource allocation, the paradigm reduces the cost of deploying cloud code. Deploying such applications in a serverless platform, on the other hand, is difficult and necessitates handing over design decisions such as quality-of-service (QoS) monitoring, scaling, and fault tolerance to the platform provider. There's a chance that an application's requirements will outgrow the platform's capabilities(Chaudhary et al., 2017).

Serverless computing offers cloud providers an additional opportunity to control the entire development stack, reduce operational costs through efficient optimization and management of cloud resources, provide a platform that encourages the use of additional services in their ecosystem, and reduce the effort required to author and manage cloud-scale applications(Chaudhary et al., 2017).

A new generation of platform-as-a-service products from major cloud providers is referred to as "serverless". Amazon Web Services (AWS) Lambda introduced these new services, which were first disclosed in late 2014(Press release, 2013), and widely adopted by mid to late 2016. Google Cloud Functions(Google Cloud Functions:, n.d.), Azure Functions(Azure Functions:, n.d.), and IBM OpenWhisk(IBM Cloud Functions:, n.d.) are just a few of the major cloud service providers that currently offer similar capabilities(Adzic & Chatley, 2017).

Traditionally, application developers would purchase or lease dedicated servers to run their systems, which were often hosted in data centers. The initial financial outlay to purchase new machines, as well as the ongoing operational costs, were both high. Increases in capacity took a long time to implement and dealing with peak computational loads in systems with fluctuating demand necessitated careful planning and the provisioning of many computers that were unused during periods of average load(Adzic & Chatley, 2017).

Developers migrated from physical machines to virtual machines as cloud computing became more popular. The capability to scale an application's deployment footprint up and down in reaction to variations in demand, paying for machine usage at a per-hour resolution, was enabled by dramatic reductions in lead time (as with AWS EC2). This, combined with auto-scaling policies, resulted in a large reduction in operating expenditures, but development and operations employees still had to manage their virtual machines actively. Platform-as-a-Service (PaaS) providers like Heroku and Google App Engine provided an abstraction layer on top of cloud services to lessen the operational burden but at the cost of flexibility and control(Adzic & Chatley, 2017).

The term "serverless" refers to a new generation of platform-as-a-service products in which the infrastructure provider is in charge of receiving and reacting to client requests, capacity planning, scheduling, and operational monitoring. Only the logic for handling client requests should concern developers. This is a big shift from the previous generation of application hosting platform-as-a-service providers. We deploy 'functions' that act as event handlers

instead of continually operating servers, and we only pay for CPU time when these functions are active(Adzic & Chatley, 2017).

A server process, usually listening to a TCP socket, waits for clients to connect and deliver requests in traditional client/server architectures. The ubiquitous web server or a message queue monitor is a classic example of this. This server process is responsible for job dispatching; however, it is also known as a gatekeeper. The application developers are responsible for the logic of processing an event in serverless deployments, but the platform is in charge of receiving and reacting to client requests, task dispatching, and task scheduling. The server process that listens to a TCP socket is no longer under the control of application developers, hence the name serverless(Adzic & Chatley, 2017).

Serverless computing, in general, is a means of providing backend services on a per-use basis. Users can build and deploy code without worrying about the underlying infrastructure configuration with a serverless provider. Any company or individual that uses a serverless vendor's backend services is charged according to their computation band resource usage. Serverless services are usually auto-scaling, so they don't have to reserve and pay for a certain amount of bandwidth or number of servers. Physical servers are still used, despite the term "serverless," and developers do not need to be aware of them(Kumar, 2019).

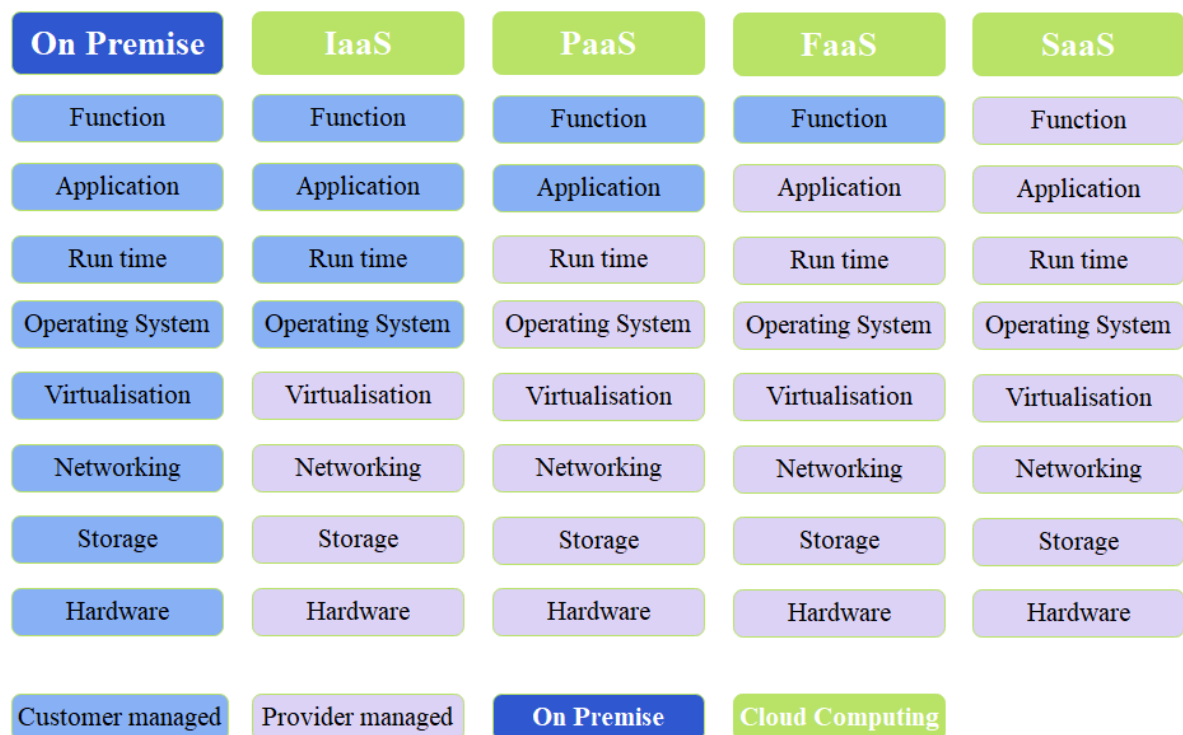


Figure 2.10: Comparison: On premise vs. IaaS vs. PaaS vs. FaaS vs. SaaS

As depicted on fig. 2.10 On Premises approach (1st column), you own the server mostly in redundancy to ensure reliability. Infrastructure-as-a-Service/IaaS (2nd column) IaaS is the next stage and we already use a virtual machine, a server located in the data center. Platform-as-a-Service/PaaS (3rd column) is following. There is no more access to the machine, but we manage container in which the applications will run. Function-as-a-Service/FaaS (4th column) shows an opportunity to control only the code. E.g., AWS Lambda is based on FaaS part of the AWS platform. So, the benefits are easy deployment and connection to triggers like API, S3, SNS, DynamoDB, good scaling and minimum of call costs. Finally, there is Software-as-a-Service (SaaS) (5th column). SaaS is Software available which is offered by third parties over the internet. All underlying infrastructure, middleware, app data are managed by the Public Cloud Providers, like Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Alibaba Cloud, Oracle Cloud, IBM Cloud. Microsoft Office 365 tools or google Apps are good examples to mention a few SaaS offers.

Serverless Advantages(Kumar, 2019)

- ✧ **Lower costs:** The 'pay-as-you-go' strategy is usually quite economical.
- ✧ **Scalability:** The serverless vendor handles all on-demand scaling.
- ✧ **Simplicity:** developers only have to worry about building backend code and not about server administration.
- ✧ **Reduced time-to-market (TTM):** Serverless considerably reduces deployment time.

Serverless disadvantages(Chrissy Kidd, 2021)

- ✧ **Loss of fine-grained control:** because the cloud provider manages all infrastructure, customers don't have access to backend infrastructure for specific configurations.
- ✧ **Loss of features:** End-users will be unable to access some serverless applications due to a lack of specialized setups. This is especially true when working with serverless database applications because several functionalities that are available in traditional database deployments are not always available in serverless versions.
- ✧ **Depending on the use case, it could be costly:** However, if you're dealing with large data sets or requests, having dedicated servers to handle the demand might be more cost-effective.

2.7.1. Serverless Architecture

The main capability of a serverless platform is that of an event-handling system, as illustrated in Fig. 2.3. The service must manage a set of user-defined functions, receive an event sent over HTTP or receive from an event source, define the function(s) to send the event to, find an instance existing function, or create a new instance, dispatch events to the function instance, wait for a response, collect execution logs, provide feedback to the user, and stop the function when it is no longer needed(Baldini et al., 2017).

The problem is to integrate such features while keeping costs, scalability, and fault tolerance in mind. The platform must be able to initiate and process a function fast and efficiently. The platform must also queue events, schedule function execution, and manage to suspend and deallocate resources for idle function instances based on the condition of the queues and the rate at which events arrive. In addition, in a cloud context, the platform must carefully consider how to scale and manage problems (Baldini et al., 2017).

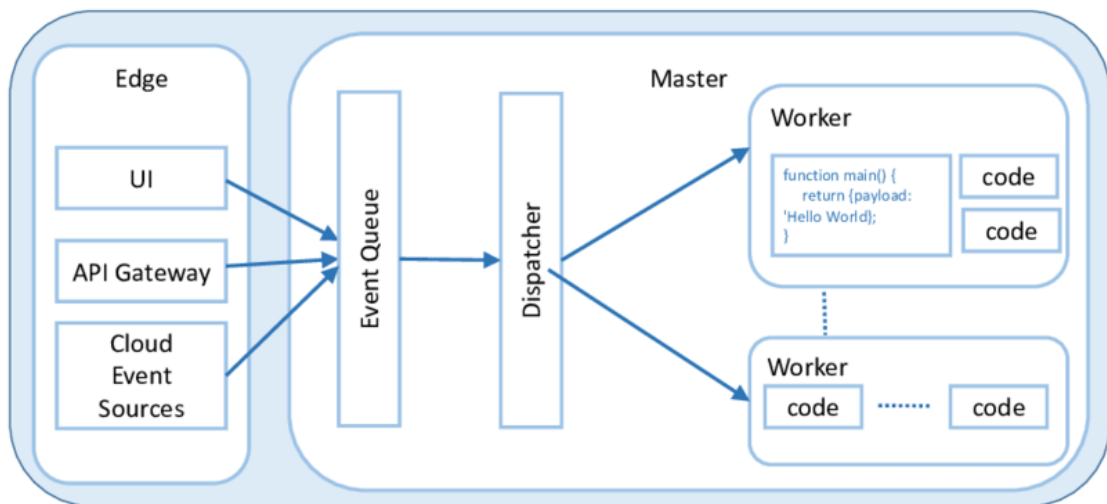


Figure 2.11: Serverless platform architecture

2.7.2. Function-as-a-Service (FaaS)

FaaS (Function-as-a-Service) is a form of cloud computing service that lets you run code in response to events without having to deal with the complicated infrastructure that comes with developing and launching microservices applications (Education, 2019). It is a serverless way to run modular code on the edge and allows developers to write and update a piece of code instantly, which can then be executed in response to an event like a user clicking on a web application element. This makes scaling code simple and is a cost-effective method of implementing microservices (Cloudflare:, n.d.).

FaaS Advantages (BROWN, 2020)

- ✧ FaaS is efficient in that it does not require the developer to write a whole program to complete a simple operation.
- ✧ It is significantly less expensive than serverless systems since small segments of code are only run when they are triggered. You just pay for the services you utilize.
- ✧ When there is an increase in market demand, it is easier to design and copy a single function rather than attempting to replicate a full application.
- ✧ Scaling individual functions is easier than scaling the entire application.

FaaS Disadvantages (BROWN, 2020)

- ✧ FaaS should be small and perform independent tasks.
- ✧ Difficulty managing a large number of functions.

2.7.3. Serverless Vs FaaS

Serverless and Functions-as-a-Service (FaaS) are often confused, but the truth is that FaaS is actually a subset of serverless. Serverless focuses on all service categories, whether compute, storage, databases, messaging, API gateways, and more, where server configuration, management, and billing are hidden from users. FaaS, on the other hand, focuses on the event-driven computing model, where the application code, or the container, executes only in response to events or requests and is perhaps the most central technology in serverless architectures (Alvarado, 2019; Education, 2019; GHOSH, 2019).

2.7.4. Serverless Computing Platforms and Frameworks

At the moment, all of the major cloud service providers such as Amazon Web Services (AWS) Lambda, Google Cloud Functions, Azure Functions, and IBM Cloud Functions have released commercial serverless systems. While they offer extra complimentary services for serverless application executions as well, they also have particular requirements for the function code composition, which over time leads to vendor lock-in. Open-source serverless frameworks have developed throughout time to get around these restrictions. These frameworks could be deployed on edge/fog devices as well as on private cloud resources, giving serverless computing capabilities with greater flexibility on-premise. Figure 13 shows some open-source frameworks as well as the current commercial serverless platforms (Mampage et al., 2022).

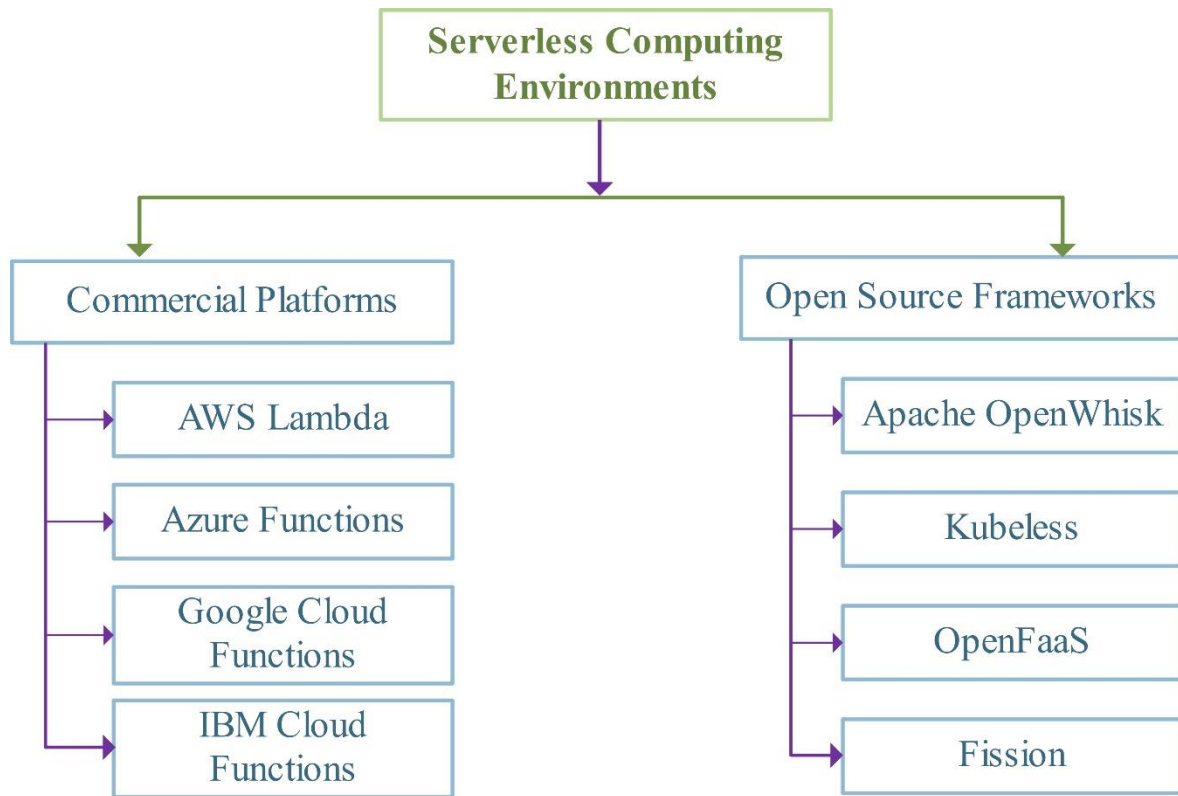


Figure 2.12: Existing Serverless Platforms and Frameworks

❖ **AWS Lambda:**

AWS Lambda is an event-driven, serverless computing platform provided by Amazon Web Services. It is a compute service that runs code in response to events and automatically manages the compute resources required by that code. AWS Lambda can be used to build data processing triggers for services such as Amazon S3 and Amazon DynamoDB; respond to web and mobile backend requests; process streaming data stored in Amazon Kinesis; or create and update DynamoDB tables.

❖ **Azure Functions:**

Azure Functions is a serverless compute service hosted by Microsoft Azure. It allows developers to run code on-demand without having to spin up virtual machines or other infrastructure. Azure Functions provides a platform for running small pieces of code (called “functions”) in response to events triggered by Azure services or third-party webhooks. This enables creating a variety of event-driven applications such as mobile backends, IoT platforms, and event-driven applications that process files from storage services like Azure Blob Storage or Dropbox.

❖ **Google Cloud Functions:**

Google Cloud Functions is a serverless execution environment designed to let users build and connection cloud services easily. It takes care of all the computational resources required making it simpler than ever before to develop applications with features such as image recognition, video encoding, real-time streams analytics, machine learning models and more with just few lines of code. Developers will be able to quickly write short snippets of JavaScript or Python code in order deploy them onto the cloud functions platform with easy versioning, authentication, monitoring and logging almost instantly after deployment

❖ **IBM Cloud Functions**

IBM Cloud Functions is an event-driven computing platform provided as a part of IBM Cloud Platform suite. It allows users write functions which have access to various IBM Cloud Services like Watson (AI/ML), Kubernetes (Containers), Compose (Database) and more. User can also choose from different programming languages to write their function including Golang, Java, Node.js & python. The biggest advantage that IBM provides on their server less Computing platform are security related tools which provide end -to end control & visibility over the application via traces & logging information about each API invocation

❖ **Apache OpenWhisk**

Apache OpenWhisk is an open source distributed serverless platform initially developed by IBM and donated to the Apache Software Foundation. It executes functions in response to events of any size and manages infrastructure, servers, and scaling with Docker containers so developers can focus on building great, powerful applications. The OpenWhisk platform supports a programming model where developers write functional logic (called actions) in any supported programming language, which are dynamically scheduled and executed in response to external sources (inputs) or events related to HTTP requests (via triggers). The project includes a REST API-based command line interface (CLI) and other tools that support packaging, list services, and many common container deployment options(Apache OpenWhisk:, n.d.).

Figure 2.13 depicts the Apache OpenWhisk architecture that consists of two main components, Controller and Invoker that are constructed on top of Nginx, Kafka, Docker, and CouchDB. All these components together form a serverless, event-based programming service. OpenWhisk provides a RESTful interface for submitting the functions and querying

the execution results. On the top we find Nginx, which acts as a gateway and load balancer to distribute the incoming requests. Nginx routes each request to one of the controllers in the current deployment, which is responsible for authenticating and retrieving the called functions from the database and forwarding them to the callers.

The controllers are the components that decide which of the available compute nodes, called workers, will schedule a specific function to run. Controllers and workers do not interact directly, but use Apache Kafka and CouchDB to manage the routing and queuing of execution requests and authorizations, the storage of functions, and their outputs/responses. Workers run functions using Docker containers. Invokers, which are distributed among multiple machines and host the containers of serverless functions, are responsible for executing the invocations by provisioning resources within Docker containers and dedicating one to them.

If there are insufficient resources for a container, functions can be enqueued in an in-memory queue for each invoker. Functions are dequeued for execution using the First Come First Serve (FCFS) protocol after the release of the necessary resources. The same instructions found in the source code of the invoker component are copied by every invoker (written in Scala). As a result, all invokers work in a similar way. On the platform, users can register and post their functions. Only the amount of memory needed to execute the function can be specified by the user. The memory sizes supported by the current version of OpenWhisk are 128, 256, and 512MB.

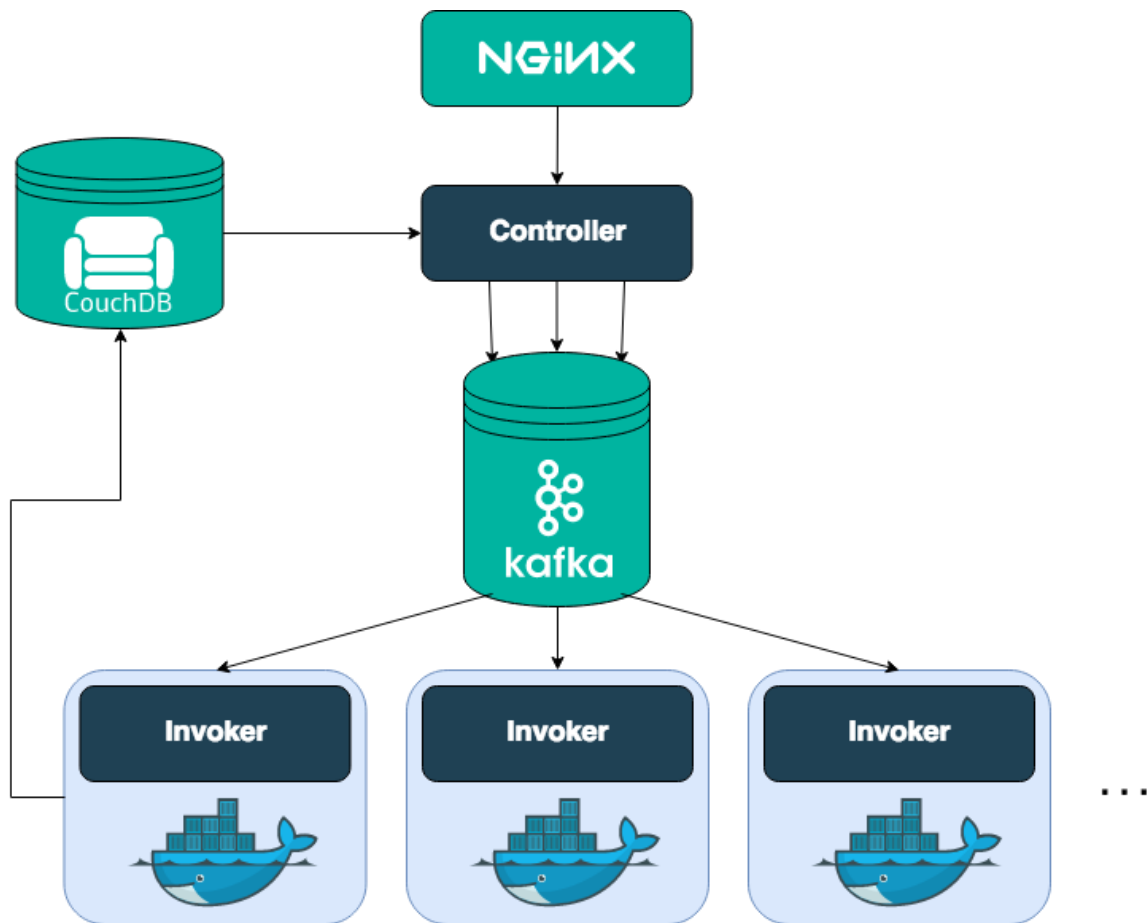


Figure 2.13: OpenWhisk Architecture

❖ **Kubeless:**

Kubeless is an open source Serverless Computing Framework running on top of Kubernetes, making use of Kubernetes native constructs such as namespaces, labels to offer developers a Serverless experience with auto-scaling facility based on demand, support custom resource definitions (for managed deployments), automatic logs management system, etc., as well as Serverless plugins like CI/CD integrations support and monitoring services integration. The main advantages of Kubeless are that it is lightweight, portable across different environments such as local development, CI/CD deployment process and cloud orchestration systems, provides flexibility for writing custom applications with built-in languages and libraries support for Python, JavaScript (Node.js), Ruby & C# language with standardized REST API endpoints to receive events.

❖ **OpenFaaS:**

OpenFaaS is an open-source Functions-as-a-Service (FaaS) project designed to provide the simplest way to run applications using microservices architectures deployed in containers. OpenFaaS abstracts away any DevOps complexity and allows developers to focus solely on their application code. This makes it ideal for developing small applications independently without worrying about infrastructure issues like availability or scaling issues. Furthermore, OpenFaaS employs auto-scaling mechanisms courtesy of container orchestrators like Docker Swarm or Kubernetes ensuring that resources are automatically provisioned whenever needed, resulting in enhanced operational efficiency at minimal cost, one of its key advantages among numerous others.

❖ **Fission:**

Fission is an open-source framework for deploying Function as a Service (FaaS). Fission encourages the development of microservices by providing an interface that makes it easy to expose individual functions as HTTP APIs over distributed systems such as Kubernetes. Fission has two components: A platform for hosting and managing functions in containers (called Fission Core) as well as a service mesh tool adding observability tools like metrics gathering and tracing options over such microservices deployed in Containers (called Fission Mesos). The primary benefits offered by this framework apart from observability are automatic scaling up/down based on CPU demand & usage patterns (provided through mesos scheduling) and accelerated service deployment times due to existing assets cached beforehand through container images, thus positively impacting performance & stability without heavy workload demands comparatively speaking.

2.7.5. Serverless Functions

Serverless functions are single-purpose, programmable functions that are hosted on managed infrastructure in the traditional sense. Cloud computing businesses host and maintain these functions, which are accessed via the Internet. The engineering teams within these companies ensure that serverless functions are always available with near-perfect availability, redundant instances around the world, and scale to any volume of a network demand(Bavosa, 2019).

Functions tend to be small, event-driven units of service that are fired when a trigger arrives and are often short-lived (milliseconds to tens of minutes). These functions take advantage

of native Linux containers to easily create and destroy instances, and use container orchestrators such as Docker Swarm or Kubernetes to scale according to request. To satisfy SLAs, providers tend to over-provision resources to these containers and use custom dispatchers to schedule them(Kumar, 2019).

2.7.6. Scheduling functions in serverless Computing

Invocation requests are sent to the provider by client applications or other functions. These requests usually have a predetermined deadline. This is especially important for real-time, latency-sensitive, or security-critical applications. Vendors must schedule where (i.e., compute nodes) and when to run functions in a way that meets deadlines while considering other system-related criteria such as energy consumption or resources usage. There are different strategies that schedulers can adopt. In the following, we summarize these strategies. Note that real-world schedulers may adopt one or more of these strategies(Shafiei et al., 2022).

Energy-aware scheduling

The main goal of this scheduling method is to reduce energy consumption by hibernating idle containers or runtime environments (or cold-state mode). The switch from cold-state to active mode causes a delay in the execution of called functions that may be greater than the client-specified delay. As a result, the executions of these methods are scheduled in such a way that the number of such transitions is minimized. (Suresh et al., 2020), introduced ENSURE, a serverless application scheduler. To avoid cold starts, it reserves a few extra containers in a warm state that can handle workload variations seamlessly. They aimed to limit the amount of extra capacity while ensuring an upper bound for request latency using a theoretical model. (Gunasekaran et al., 2020a), takes a similar strategy, spawning containers ahead of time to avoid cold starts. An energy-aware scheduler can also lower total energy usage by delaying non-latency-sensitive operations (such as background or maintenance tasks).

Resource-aware scheduling

Serverless applications, like their resource consumption patterns, are diverse. A scientific computing function, for example, is typically CPU-intensive, whereas an analytical application is usually memory-intensive. When several CPU-intensive functions are co-located in a physical node, resource contention occurs, causing delays in the execution of

those functions. Other forms of computational resources, such as memory, disk, and network, are similarly affected. Resource-aware schedulers assign functions to compute nodes in order to meet the functions' resource requirements on a timely basis. The serverless scheduler (Suresh & Gandhi, 2019), belongs to this category. It divides functions into categories based on how much CPU they use. It then dynamically regulates their CPU shares at runtime to reduce CPU contention between co-located functions. (Kim et al., 2020) introduced a dynamic CPU capping method for serverless computing, as well as a group-aware scheduling algorithm, to efficiently reduce CPU contention between functions inside compute nodes. (Reza Hoseinyfarahabady et al., 2018) proposed a prediction tool to estimate the rate of event streams within providers in order to reduce CPU contention and improve QoS. (Hoseiny Farahabady et al., 2017) also introduces a QoS-aware resource allocation strategy for serverless systems that dynamically scales by forecasting the future rate of incoming events. (Gunasekaran et al., 2020a) on the other hand, uses offline profiling to calculate the expected execution time of functions and adaptively balances the load. It gathers samples and predicts the cost and execution time of a serverless function across unknown configuration parameters using statistical learning techniques. The framework selects the appropriate configuration parameters for running a single or a chain of functions while meeting customer objectives based on estimated cost and execution time.

Workflow-aware scheduling

To complete their tasks, serverless applications typically require the execution of several functions. Serverless workflows are made up of stateless small discrete functions that are chained together and orchestrated. Schedulers can predict, schedule, and provision in advance by knowing the chain of function invocations. For example, prepare a warm container for the execution by knowing the chain of function invocations. When compared to OpenWhisk, (Daw et al., 2020) employs such an assumption to minimize overheads and delays by up to ten times. (Tariq et al., n.d.) and (Singhvi et al., 2019) both use this method in different ways. (Singhvi et al., 2019) employs DAG-structure to estimate the size of worker pools and hence achieve low request execution scheduling overheads. Sequoia improves Quality-of-Service (QoS) by prioritizing, scheduling, and queuing function chains, or functions inside workflow chains.

Dataflow-aware scheduling

Although in an ideal serverless environment, functions are stateless and do not rely on external data sources, this is rarely the case in practice. Many machine learning applications, for example, rely largely on external sources. As a result, the scheduler should account for data availability or data serving delays. Freshen was introduced by (Hunhoff et al., 2020), which allows developers or providers to proactively fetch data while executing serverless operations, as well as other runtime reuse features to minimize overheads. (Sreekanti et al., 2020) prioritizes data proximity in its scheduling policy.

Package-aware scheduling

Upon receiving an invocation request, the compute node in some current serverless technologies, such as OpenLambda, should fetch and install application libraries and required packages defined by function. This clearly takes time and causes the execution to be delayed. This issue was addressed by (Aumala et al., 2019), who introduced package-aware scheduling. Other serverless systems are typically designed so that clients can install the packages themselves during the registration process.

Hybrid scheduling

Hybrid virtual-machine/serverless scheduling has lately gained popularity. The goal is to have the best of both worlds: a central scheduler determines whether to route requests to a private virtual machine or a serverless provider. (Gunasekaran et al., 2019) is a hybrid scheduling system that uses VM-based auto-scaling to flatten request peaks. (Das et al., 2020) is a hybrid scheduler that aims to reduce the cost of using public cloud infrastructures while adhering to user-defined deadlines.

2.8. Related Works

With the rapid emergence of serverless technology, much research has been done in this area and we highlight relevant work that has looked at serverless models:

A serverless platform designed for latency-sensitive DAG-structured serverless applications is presented in (Singhvi et al., 2019). The key idea in the paper is to utilize a deadline-aware job scheduling for serverless environments. A cluster-level centralized and core-granular scheduler for serverless functions is introduced in (Kaffes et al., 2019a). (Aumala et al., 2019) proposes separating common dependencies from function code packages (such as

libraries), caching them directly on worker nodes, and scheduling invocations to take package availability into consideration. Because large libraries do not have to be downloaded many times, this strategy decreases startup time; however, the PASch study's performance is not evaluated under a variety of dynamic workloads.

An adaptive resource management framework has been proposed by (Gunasekaran et al., 2020b) to effectively manage function chains on serverless platforms. Usage-aware by efficiently bin-packing jobs in fewer containers with function-aware container scaling and intelligent request batching, while being SLO-compliant by proactively spawning containers to avoid cold starts, minimizing overall response latency, is the key idea in this work. (Gunasekaran et al., 2020c) takes into consideration the impacts of cold starts and considers the execution of function chains (i.e., invocation patterns where one function invokes another). They use LSTM to implement its prediction model for incoming invocations. However, the centralized database is used to make all decisions related to container scaling, scheduling, and load prediction, which can constitute a bottleneck in terms of scalability and consistency for a large-scale system.

(Stein, 2018) examined different techniques to function scheduling and suggested a Non-cooperative Online Allocation Heuristic (NOAH) for scheduling functions on serverless platforms to reduce cold starts. The number of instances required to maintain the waiting time below the threshold is computed heuristically. According to the estimation and the threshold, each worker node decides whether to enqueue a request or launch a new instance.

(Kaffes et al., 2019b) have argued for and proposed a novel cluster-level centralized and core-granular scheduler for serverless operations. There are several schedulers, each with a group of idle workers. An incoming request is assigned to an idle worker by each scheduler. If a scheduler doesn't have any idle workers, it can take them from other schedulers.

(Suresh & Gandhi, 2019) proposed FnSched, a serverless function scheduler that imposes acceptable latencies on requests to reduce provider resource costs. FnShed uses classification to forecast resource consumption and lifetime trends of invoked functions, and then schedules them depending on their class. By reducing CPU contention on the worker, FnSched aims to maximize the number of functions that can be executed simultaneously on a single worker node. They have also presented ENSURE (Suresh et al., 2020), which is based on FnSched but adds autonomous resource management. ENSURE focuses on resource efficiency both inside and outside of containers. It uses FnSched to schedule

functions in such a way that they employ the smallest number of invokers possible in response to load, and FnScale to proactively scale out the number of invokers to eliminate setup delays. The primary goal of these studies has been to improve resource use while lowering provider costs.

Atoll is a scheduling framework developed by (Singhvi et al., 2021) to provide low latency function execution in a multi-tenant serverless context while fulfilling deadlines. To do this, Atoll divides each cluster into smaller worker pools, each of which is scheduled by a semi-global scheduler, and each application is viewed as a DAG of functions. When requests for distinct DAGs arrive at different rates, they are sent to different clusters. To reduce the risk of missing deadlines, each cluster schedules incoming DAGs. A local scheduler in each cluster is in charge of scheduling arrived functions based on the minimum remaining time criteria.

(Jia & Witchel, 2021) reduces overheads by trading off isolation as it proposes running multiple invocations of the same function in the same container and optimizes inter-function IO. (Fuerst & Sharma, 2021) comes up with a workload-aware keep-alive policy that takes into account additional function characteristics such as function size and initialization costs. They implement caching-based keep-alive and resource provisioning policies in the FaasCache system, which is based on OpenWhisk. (Carreira et al., 2021) reduces overheads due to executing unoptimized code in serverless functions through sharing runtime compiler JIT optimizations across active function sandboxes in the cluster.

(Przybylski et al., 2021) proposed a number of scheduling strategies, including SEPT and theoretical variants of Fair Choice. These strategies are evaluated based on the Azure function trace (Shahrad et al., 2020); it is demonstrated, among others, that reliable estimates of execution times can be obtained from just the 10 most recent invocations. (Yu et al., 2021) optimizes the assignment of the invoker to a function call through a neural-network approach. Further optimizations are possible for complex FaaS invocations (in which a function calls another function forming a chain of invocations). (de Palma et al., 2022) presented TAPP, a declarative language that gives DevOps more precise scheduling control for serverless functions. TAPP scripts, which are topology-aware, can limit the execution of functions within zones and enhance serverless applications' performance by taking advantage of data or code locality qualities and tests that emphasize data locality receive significant latency reductions.

(Fu et al., 2022) proposed SFS (Smart Function Scheduler), which works entirely in user space and carefully orchestrates existing Linux FIFO and CFS schedulers to approximate the Shortest Remaining Time First (SRTF). SFS uses two-tier scheduling that seamlessly combines a new FILTER policy with Linux CFS to balance the increased duration of long functions against a significant performance improvement for short functions.

Table 2.1: Summary of related work

S/N	Title (Author)	Quality of Service Achievement	Limitation or Gap	Methodology
1	Beyond load balancing: Package-aware scheduling for serverless platforms (Aumala et al., 2019)	Caching required package, reducing latencies, avoiding imbalances	PASch study's performance is not evaluated under a variety of dynamic workloads.	PASch -applies package affinity when scheduling to reuse execution with preloaded packages
2	ENSURE-Efficient Scheduling and Autonomous Resource Management in Serverless Environments (Suresh et al., 2020)	Increase resource utilization and decrease provider expenditure	Reducing latencies and starvation avoidance is not observed	ENSURE - greedily "packing" requests on one host before moving on to another
3	ETAS-predictive scheduling of functions on worker nodes of Apache OpenWhisk platform (Banaei & Sharifi, 2021)	Increased throughputs, lowered response time overhead, and increased reuse of warm sandboxes	Lack of QoS considerations, like provider cost, and resource utilization	ETAS - Apply predicted function execution time based on previous execution records
4	Atoll - A Scalable Low-Latency Serverless Platform (Singhvi et al., 2021)	Reducing latencies and missed deadline rate	Throughput and resource consumption concerns are not addressed.	Atoll - Conduct proactive sandbox allocation, deadline-aware scheduling, sandbox aware routing, and automatic scaling

5	FaaSCache: Keeping serverless computing alive with greedy-dual caching (Fuerst & Sharma, 2021)	Reducing the cold-start overhead, and application-latency	Increases memory pressure and reduces resource utilization	FaaSCache - Caching-based keep-alive and resource provisioning policies
6	Data-driven scheduling in serverless computing to reduce response time (Przybylski et al., 2021)	Aimed to increase performance as shown by metrics like response time	Increase in the consumption of memory	SEPT and SERPT strategies with theoretical variants of Fair Choice
7	Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider (Shahrad et al., 2020)	Reduce the number of cold starts	Resource utilization and system throughput is not optimized	Uses a varied keep-alive value based on the call frequency and pattern for each user's workload
8	FaaSRank: Learning to Schedule Functions in Serverless Platforms (Yu et al., 2021)	Reduced average function completion time	Imbalances and starvation limitation	FaaSRank - reinforcement learning (RL) and neural networks supported by Score-Rank-Select architecture
9	Fifer: Tackling Resource Underutilization in the Serverless Era (Gunasekaran et al., 2020b)	Improving resource utilization	Avoiding cold starts, minimizing overall response latency	Fifer -bin-packing jobs in fewer containers with function-aware container scaling and intelligent request batching
10	SFS: Smart OS Scheduling for Serverless Functions (Fu et al., 2022)	Reduce the execution duration and improved turnaround time for short functions	Execution duration of long functions increased and starvation	SFS - approximates SRTF with dynamic and adaptive time slicing in the first-level global queue to combine the best of FIFO and RR.

CHAPTER THREE

MATERIALS AND METHODS

3.1. Overview

This chapter elaborates on the detailed methods, tools, and procedures used to analyze the research problems. It provides a very concise background of methodology that can fit the design, implementation, and evaluation of the proposed architecture. In more detail, it discusses the research strategy, method, approach, the data collection techniques, the research measures or metrics, the research procedure or technique, the simulation system, the method of testing and evaluation, the algorithm used. Most importantly, this section provides the academic justification for method selection to achieve the main goal of the research.

3.2. Research Method

Quantitative research was conducted to achieve the goals of the studies. The key feature of this type of research is that it is best suited to both the small data set and the large data set while its results are measurable and quantifiable. Its direct advantage, which also accounts for its fundamental difference from qualitative research, is that quantitative data is more efficient and able to test hypotheses by limiting the research scope and the characteristics of the participants' responses, i.e., are not subjective (interpretation of events by individuals may not be important)(Collis, J. and Hussey, 2014). In general, the reliability of quantitative research is based on the researcher's skills and abilities and also on the value of the data used, while the results must be considered accurate as the results are not derived from the researcher's personal opinions and perceptions.

3.3. Research Strategy

The study to support this thesis concerns applied types, but not new ones. Instead, there are several previous academic research papers on function scheduling in serverless computing. This research is conducted by reviewing several relevant literature papers on serverless computing environments in the domain of functional scheduling. To gather the appropriate resources accurately, the author has analyzed written articles, research papers and a forum on the identified problem and also visited the official technology pages to look at specifications, methods, technology stacks, conventions and archetypes. In addition, some

significant works of specialists and authors are reviewed. In conduct this work, the following brief steps apply to the research strategy.

- ✧ First, formulating the problem by reviewing different literature and documents.
- ✧ Secondly, collecting data from different guidelines, documents, and websites related to function scheduling.
- ✧ Thirdly, mention the research methodologies such as for systematic literature review, data collection method, function scheduling methods, proposed framework design method, the framework implementation method, the prototype evaluation method, and the design and implementation tools used for the research work.
- ✧ Then, designing function scheduling for a serverless framework.
- ✧ And then implementing the proposed framework.
- ✧ Finally, evaluate and test the functionality and usability of the proposed framework.

3.4. Research Approach

The investigative approach pursued for this research is the deductive one. According to this approach, researchers begin by examining an identified phenomenon and testing whether that theory is valid under certain circumstances. The reason for the deductive approach was that it takes into account the context in which the research effort is active, while also being best suited to small as well as large samples that produce qualitative data or results (the reliability of the research results cannot be questioned) (Leech, n.d.).

3.5. Data Collection Methods and Tools

Data Collection is a fundamental aspect of any analysis study. The data collection starts with determining what kind of data are needed to be followed by techniques of information collection. In this study, user input data will be collected from external sources and generated using simulation tools to develop and evaluate the proposed solution. The dataset and data in this case is a set of functions scheduled for the available container and these user inputs are computational functions based on (Kuntsevich et al., 2018). These functions are: primeNumber, which finds the nth prime number, factorial, which calculates the nth factorial, multiplyMatrix, which multiplies two matrices of size n, sleep function, which sleeps for n seconds. The reason we chose these functions is that firstly they are all CPU and memory intensive (not I/O intensive) and secondly, they are all sensitive to the inputs and resource settings. Based on this information, the effectiveness of the proposed solution will be evaluated and compared to other existing approaches.

3.6. Research Process

The algorithm that is developed in the proposed study uses a hybrid scheduling method for serverless function scheduling. By combining ETAS (an execution-time-aware scheduling algorithm) and Fifer (a function-aware container scaling and intelligent request batching algorithm), the function scheduling algorithm for the serverless environment is developed. The limitations or issues with these scheduling algorithms are addressed by this hybrid or combination, which enables us to create a strong algorithm for the serverless cloud computing system.

The hybrid algorithm functions as follows. Functions are dispatched to the worker nodes to execute in dedicated sandboxes. If there are not enough resources, functions are enqueued until the required resources are released. The queue scheduler (*ETAS*) determines which function should be executed when the required resources are released. The scheduler assigns function requests using their execution times, estimated by their previous execution's history, arrival times, and containers' status. First, the execution times of functions is predicted by profiling the execution times of the previous function executions, then schedules them on worker nodes based on the prediction. Every worker node has a load monitor to keep track of the request load and periodically sends updates to the load balancer. The load balancer decides the scale out factor (#containers) of every worker based on the predicted load and existing queuing delays for each worker node. *Fifer* algorithm is used to determine the number of containers needed at every worker node based on this queuing delays, which holds all the incoming function requests and predicted load.

3.7. Simulation Environment

There are many simulation toolkits for serverless environments, but the proposed function scheduling algorithm is simulated using Apache OpenWhisk framework. Because this toolkit is better suited to our selected needs. Following are the justifications for selecting Apache OpenWhisk:

Open Source: Apache Openwhisk is an open source serverless offering, which allows users to create their own functions without having to pay for extra services. This makes it an attractive option for developers and businesses looking to save money on their serverless deployments.

Flexibility: Apache Openwhisk's platform-agnostic nature allows it to run on multiple environments and operating systems, including Kubernetes, IBM Cloud Functions, and

Amazon Web Services (AWS). This makes it easier for teams of developers to use the same technology stack when working on projects where flexibility is essential.

Portable: Unlike many other serverless offerings that require code to be written in a specific language or framework, Apache Openwhisk allows users to write code in any language supported by the service. Additionally, the portability across different cloud environments offers significant advantages over traditional deployment models where developers need to re-write code for each environment.

Comprehensive APIs: The comprehensive set of REST APIs make it easy for developers to access various plugins and extensions that can help customize the implementation of their application as well as enable integrations with third parties such as databases and messaging queues like RabbitMQ or Kafka.

Scalability: Apache Openwhisk is highly scalable meaning that users can allocate more resources without worrying about resource contention issues caused by other tenants accessing the same environment at the same time.

OpenWhisk operates on three types of components: controllers, invokers and action containers. A controller manages other entities and routes actions invocations to invokers, acting as a load balancer if multiple invokers are available. With multiple invokers, the communication requires an intermediate event streaming system, such as Apache Kafka. A single invoker manages a single worker node acting as its node-level resource manager. A node hosts multiple action containers, which in turn execute function calls and return the results. The invoker is also responsible for assigning a function call, routed by the controller, to a specific action container. In a standard configuration, a single container does not execute multiple function calls in parallel.

When an invoker receives a new request and there are pending requests, the request is added to the queue; otherwise, the invoker tries to arrange a container on which the request can be executed immediately. First, the invoker tries to locate a free pool container matching the request. A free pool container is a container that is initialized with the execution environment and the requested function. Such containers can be usually found if a particular function is executed repeatedly. If a free pool container is not available, the invoker tries to find a prewarm pool container. A prewarm pool container is initialized with the execution environment suitable for a group of functions, but the specific function has not yet been initialized. If no free or prewarm containers match the request, the invoker tries to create a

new container. This may be impossible if there is not enough free memory. In such a case, some non-matching free pool containers may be removed (evicted). If there are not enough free pool containers to be evicted, the action call is queued.

In the standard OpenWhisk, the controller forwards the request to the invoker. Although it is easy to implement, the load balancing of this method is more static because the decision to execute the request of a particular invoker cannot be simple. To avoid this, OpenWhisk is currently gradually switching to a new action assignment model based on global queues. In this new model, the controller does not decide which invokers will be responsible for executing actions. Instead, each invoker pulls requests from common Kafka queues: each function has its own global queue, and the invoker pulls requests from queues corresponding to its free pool containers. However, the controller still decides what kinds of containers will be created by specific invokers.

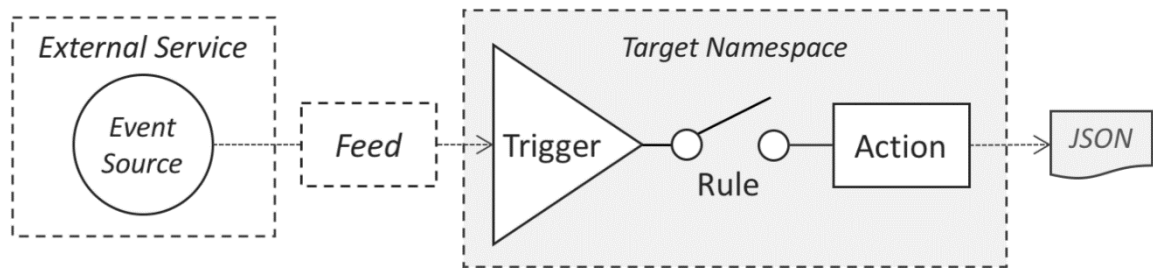


Figure 3.1: OpenWhisk programming model (Apache OpenWhisk:, n.d.)

Experimentation and Testing Tools

Our proposed solution is implemented on Apache OpenWhisk, an open-source serverless cloud platform. To receive and respond to requests, OpenWhisk provides a REST interface. The Controller component is responsible for parsing the HTTP method and calling the Invokers function that creates the Docker container.

3.8. Proposed Architecture

The general objective of this study is to develop an efficient algorithm for function scheduling in Serverless cloud computing. It works on both global and local scheduling schemes of worker nodes to improve the current performance of the scheduler by reducing response time and increasing throughput and resource efficiency to meet user's requirements of quality of service. The following diagram shows the architecture of the proposed solution.

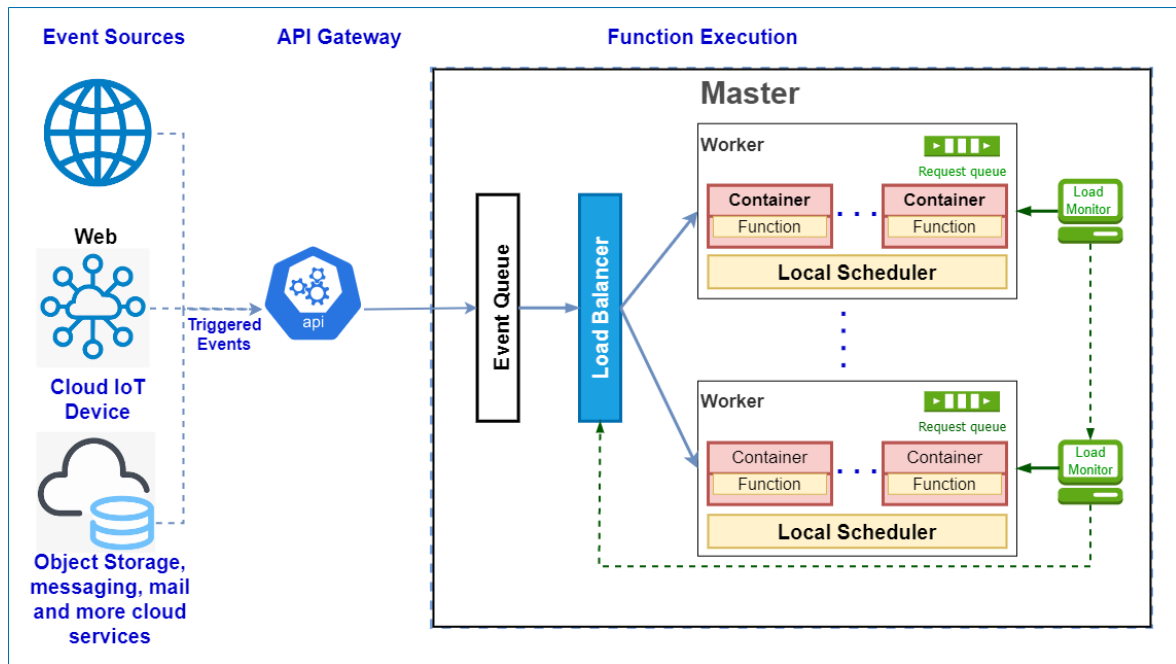


Figure 3.2: Proposed Architecture

(Modified components of the Architecture are in light blue and new components are highlighted in green).

Figure 3.2 shows how serverless systems will accept an event supplied over HTTP or from another cloud-based event source. The platform's internal components include a load balancer, workers, and an event queue. The system decides which action(s) should handle the event after receiving a request. Once the load balancer has retrieved the code of the action from a database, it passes the request to a worker.

The worker is responsible for following the procedures. When the worker receives the functions and there are enough resources for the function, it starts one or more instances of the resource-constrained function (depending on the number of requests) and sends the functions to the instance (usually containers) for execution. Otherwise, the functions will be queued until the required resources are released.

An efficient load balancer can prevent overloaded worker nodes. However, when there are plenty of invocations and burstiness, a large execution queue forms, which lengthens the time invocations must wait before being executed. In addition, while employing the least nodes possible to respond to invoked functions in order to improve resource utilization while maintaining Quality of Service (QoS), some functions might be left in node queues.

Therefore, the local scheduling strategy has a significant impact on the throughputs, resource usage, and average waiting times of functions for worker nodes.

Launching the instance requires initializing the container with the necessary libraries, which can cause an unwanted startup delay to complete, a process known as a "cold start". To avoid cold-start latency, cloud providers suspend instances after completion to reuse them for future invocations. When the action completes or reaches the maximum execution time, the worker collects the execution logs and provides the response to the user. Scale-to-zero allows users just to pay for the time and resources used when their functions are running.

The load monitor (LM) monitors scheduled requests to determine if there are any SLO violations due to queue delays. This is because there are not enough containers to handle all the requests in the queue. In this case, we estimate the additional containers needed for a worker node, knowing the free resources of the worker and the number of pending requests in the Queue, using the load monitor. The load balancer (LB) and load monitor (LM) are integrated with each worker so that the containers needed for each worker can be efficiently scaled reactively.

3.9. Proposed Algorithm

In this study, we proposed a hybrid serverless function scheduling algorithm that takes advantages of ETAS and Fifer algorithms. ETAS primary goal is to reduce response times. However, it falls short in terms of enhancing resource usage as well as reducing provider costs. The usage of slack estimation methodologies by Fifer, on the other hand, allows it to greatly reduce the number of containers needed for each function, which improves resource consumption and cluster energy efficiency. Despite the fact that slack-based request queuing considerably reduces the number of containers created, it still causes SLO violations due to cold-starts, particularly during dynamic load changes. ETAS, on the other hand, schedules on warm instances based on the estimated function execution time to prevent starvation and reduce cold starts.

By combining advantages of these two algorithms, a hybrid approach improves both response times and the use of containers to achieve high resource efficiency without compromising SLOs. The hybrid approach uses proactive container provisioning by forecasting functions execution time based on the history of their prior execution, arrival times, and container status to reduce response time and cold starts. Additionally, the

container is scaled in and out reactively by monitoring the workload through load monitor (LM) on the worker node to reduce SLO violations due to by queuing delays and starvation.

3.9.1. Assumptions

The following assumptions underlying in our proposed scheduling scheme. First of all, we have considered a system under a high workload. The remaining assumptions are as follows:

1. There are multiple clients in the system.
2. Each function has its own resource setting (the resource setting is the number of resources allocated to the function sandbox).
3. A function can be called with different inputs, and its execution time can be affected based on its input.
4. Function execution times can be affected by their resource setting.
5. Functions are compute-intensive and require no network or I/O.
6. Functions do not have priority among themselves.
7. Functions are independent and we do not distinguish between a function called by an external request (from outside the platform) or an internal one (called by other functions).

3.9.2. Execution time prediction

Predicting the execution times of functions is achieved by profiling the execution times of previous executions. According to the above assumptions, function input, and resource setting can affect execution time. Therefore, to increase accuracy, different records were considered for each function, resource setting, and input. Therefore, whenever a function is invoked by a specific input, the predicted execution time based on the function name, resource setting, and input is returned. In other words, there is a record containing the estimated execution time of the function for each tuple containing username, function name, resource setting, and input.

To profile the execution times, two scenarios are taken into consideration. The time is calculated and saved after executing a requested function. There is no record of an invoked action in the first scenario. As a result, the calculated time is recorded as a prediction for the following request. But if a record exists, the average of the old record and the new one is determined and substituted in accordance with an exponential average (Equation 1).

$$\tau_{n+1} = \alpha \tau_n + (1 - \alpha) \tau_n, 0 \leq \alpha \leq 1 \text{-----Equation 3-1}$$

The value of t_n contains the last execution time, while τ_n stores the past and τ_{n+1} , is the predicted time for the next invocation. The higher the value α , the closer the predicted value is to recent history. So, we assume $\alpha = \frac{1}{2}$ so that recent history and past history are weighted equally.

In addition, for each action with a given resource setting, the weighted average of all the predicted times of different inputs is calculated and updated in the cascade. This weighted average is used if there is no estimation record for an invoked function with a specific input. Saving the predicted time in the database is the next step.

One of the assumptions we made in (Section 3.9.1) was that there are clients with different functions that require different resource settings that are invoked with different inputs. As a result, there are separate records for each action and input. Therefore, we considered a two-level key-value structure. At the first level, for each action there is a key, and the value is again a key-value structure. We use a hash function to create the keys, so for the first level, the tuple $\langle Username, Function Name, Resource Setting \rangle$ is used as the hash value. For the second level, the key is created by applying the hash function to each input. Figure 3.3 illustrates the scheme for storing the previous execution times of functions. An entry is considered for each resource setting, which contains different times depending on the input.

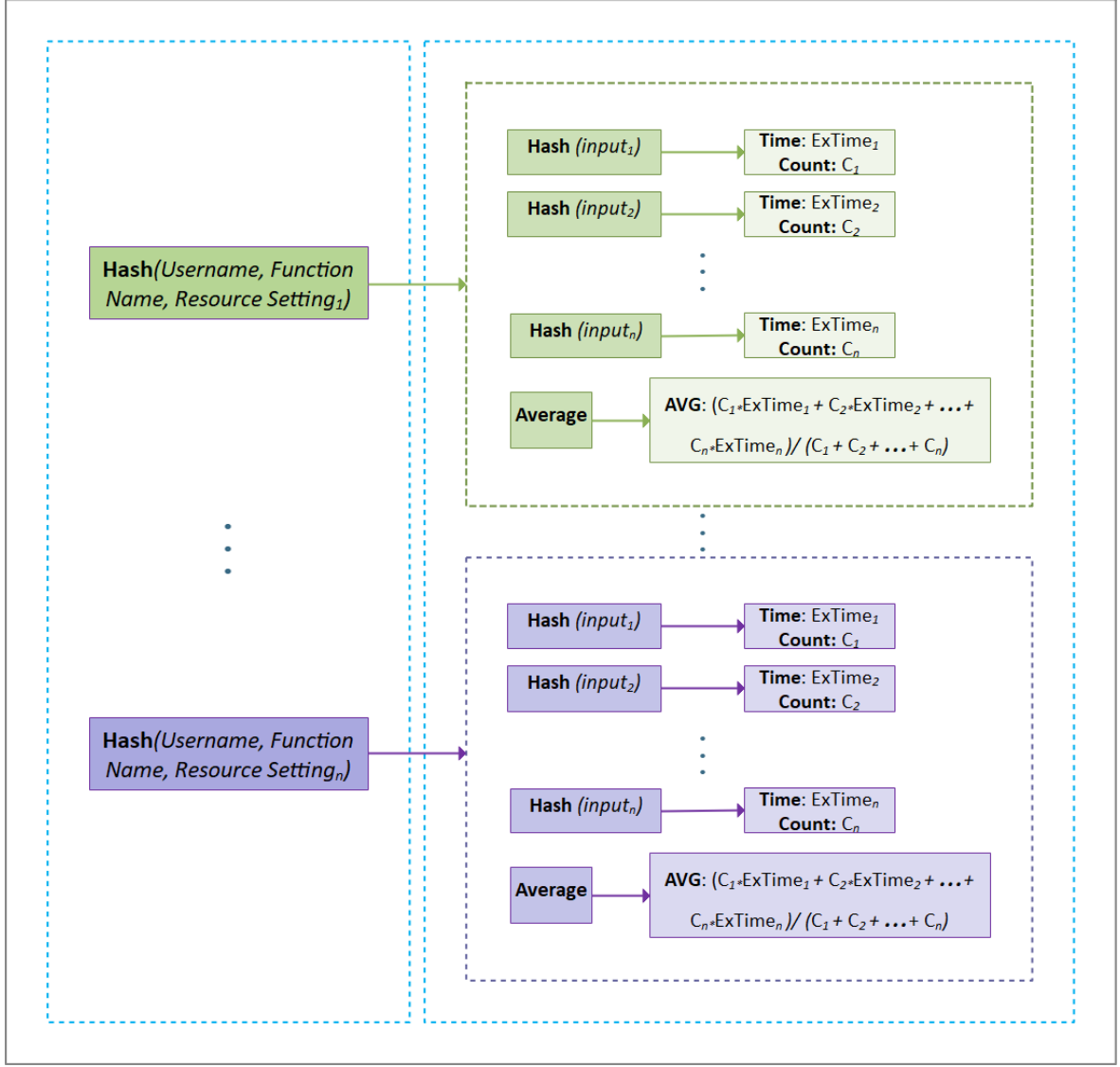


Figure 3.3: A key-value storing structure for storing a function execution time

3.9.3. Scheduling

The scheduling discipline determines the order of function execution, which has a significant impact on worker node latencies and throughputs. When a function grabs a container, retrieving the container is inefficient due to cold start and loss of execution state. For this reason, we choose a non-preemptive scheduling discipline.

Algorithm 1: Execution time aware function scheduling algorithm

Require: A new function ($f_i \in F, i \in N$) which is arrived at the Worker node ($w_p \in W, p \in N$). And q_p is the queue in w_p

- 1: *FunctionExecute* (f_i)
- 2: **if** (f_i) == f^c or *Empty* (q_p): **then**
- 3: **if** $FR_p^w \geq R_i^f$: **then**
- 4: **if** (f_i) $\in$ (q_p): **then**

```

5:      Dequeue ( $f_i$ )
6:  end if
7:       $c_i = \text{CreateContainer}$  ( $f_i$ )
8:  InvokeContainer ( $c_i, f_i$ )
9:  if notEmpty ( $q_p$ ): then
10:       $f^c \leftarrow \text{Min}(q_p, T^f)$ 
11:      Execute ( $f^c$ )
12:  end if
13:  else
14:      if Empty ( $q_p$ ): then
15:           $T_i^f \leftarrow \text{Time\_now}() + T_i^e$ 
16:          Enqueue ( $f_i$ )
17:           $f^c \leftarrow (f_i)$ 
18:          Execute ( $f^c$ )
19:  else
20:      if Empty ( $WC_p$ ): then
21:          if Empty ( $FR_p$ ): then
22:              Execute ( $f^c$ )
23:          else
24:              for funcinqp: do
25:                  if  $T_{func}^a < T_{fc}^a \ \& \ FR_p \leq R_{func}^f$  : then
26:                       $f^c \leftarrow func$ 
27:                      Execute ( $f^c$ )
28:                      Break
29:                  end if
30:              end for
31:          end if
32:  else
33:      for funcinqp: do
34:          if  $T_{func}^a < T_{fc}^a \ \& \ \text{WarmContainerExist}(WC_p, func)$ : then
35:               $f^c \leftarrow func$ 
36:              Execute ( $f^c$ )
37:              Break
38:          end if
39:      end for
40:  end if
41:  end if
42:  end if
43:  else
44:       $T_i^f \leftarrow \text{Time\_now}() + T_i^e$ 
45:      Enqueue ( $f_i$ )
46:  end if
47:  END Function

```

Table 3.1: The summary of the notations used in the mathematical model

Symbols	Description
$U = \{a_1, \dots, a_n\}, n \in N$	Set of users in the system
$F = \{f_1, \dots, f_m\}, m \in N$	Set of functions in the system
$R_i^f, i \in \{1, \dots, m\}, m \in N$	Required resource of function _{<i>i</i>}
$T_i^e, i \in \{1, \dots, m\}, m \in N$	Execution time of function _{<i>i</i>} in <i>ms</i>
$W = \{w_1, \dots, w_k\}, k \in N$	Set of worker nodes in the system
$Q = \{q_1, \dots, q_k\}, k \in N$	The set of queues in the system. And q_i belongs to w_i
B_size	Number of requests which can be queued in a container
$T_{w_i}^e$	Total execution time of worker _{<i>i</i>}
$R_i^w, i \in \{1, \dots, k\}, k \in N$	Total resource capacity of worker _{<i>i</i>}
$FR_i^w, i \in \{1, \dots, k\}, k \in N$	Total free resources of worker _{<i>i</i>}
$WC_i, i \in \{1, \dots, k\}, k \in N$	Pool of warm containers in worker _{<i>i</i>}
$T_i^a, i \in \{1, \dots, m\}, m \in N$	Arrival time of request for function _{<i>i</i>} (based on origin of time) in <i>ms</i>
PQ_{len}	Number of pending requests in the queue
D_f	Queuing delay threshold
C_d	Cold start
$T_i^f = T_i^a + T_i^e, i \in \{1, \dots, m\}, m \in N$	Ideal finish time of executing function _{<i>i</i>} without delay
f^c	A candidate function that scheduler selects to execute after releasing the required resources

As we have already indicated, functions are dispatched to the worker nodes to run in dedicated sandboxes. If there are insufficient resources, functions are queued until the required resources are released. The queue scheduler determines what function to run when the required resources are freed. The proposed scheduling scheme decides the next function according to the following steps:

1. When a function (f_i) arrives in a worker, its predicted execution time is fetched (T_i^e)
2. Before enqueueing the function (f_i), T_i^f is calculated and stored with the other function's data.
3. The scheduler chooses a function with the lowest T^f and takes it as a candidate function (f^c) when it needs to select the next function from the queue.
4. Now if there are enough resources to run the f^c , it is executed. Otherwise, ETLAS looks for a function that has a warm sandbox or accessible required resources. First it checks if there is a warm sandbox for any function in the queue that arrived earlier than f^c and if a function is found it is executed before f^c , otherwise the scheduler checks if there are enough resources for each function that arrived in the queue before f^c , and if a function is found, executes it before f^c . Else, f^c is executed.

Algorithm 2 *Worker nodes function aware container scaling algorithm*

```

1: Procedure DYNAMIC_REACTIVE_SCALING(WORKERS)
2:   for worker in  $\forall$ workers do
3:     delay  $\leftarrow$  Calculate_Delay(last_10s_jobs)
4:     if delay  $\geq$  worker.slack then
5:       est_containers  $\leftarrow$  Estimate_Container()
6:       worker.containers.append(est_containers)
7:     end if
8:   end for
9: end procedure
10: procedure ESTIMATE_CONTAINERS(WORKERS, PQ_LEN)
11:   total_delay  $\leftarrow$  PQ_len * worker.resp_latency
12:   current_req  $\leftarrow$  len(worker.containers) * batchSize
13:   delay_factor  $\leftarrow$  total_delay/current_req
14:   if delay_factor  $\geq$  cold_start then
15:     est_containers  $\leftarrow$  (PQ_LEN - current_req)
16:     return est_containers
17:   end if
18: end procedure

```

User applications send requests to the system which are handled by the cluster scheduler. The scheduler dispatches the requests to queues. From there, requests make it to the containers depending on the available free slots. Request queue has a load monitor to keep

track of the request load and periodically sends updates to the load balancer. The load balancer decides the scale out factor based on the predicted load and existing queuing delays.

3.9.4. Load Balancing

Our proposed solution utilizes a request queue, which holds all the incoming tasks. We design a load balancer (LB) along with a load monitor that are integrated to each worker for efficiently scaling containers for the function requests. Since we know the execution time of functions and total free resources of worker the load balancer can calculate the batch size (B_size) for each worker.

$$B_size \leftarrow \frac{FR_i^w}{T_{w_i}^e} \text{-----Equation 3-2}$$

Dynamic Reactive Scaling Policy: To accurately determine the number of containers needed at every worker which is a function of B_size and queue length, we need to periodically measure the queuing delay due to batching of requests. In our case the number of requests which can be queued in a container is defined as the batch size (B_size) of the container. As shown in Algorithm 1a, the Load Monitor (LM) checks each worker's scheduled requests for the last 10 seconds of a given monitoring interval to check for SLO violations caused by queuing delays. This is due to the fact that not enough containers are available to handle all of the requests in the queue. In this regard, additional containers are created after we estimate the required number of containers using the *Estimate_Containers* method (Algorithm 2). By knowing the B_size and number of pending requests in the Queue (PQ_{len}), the function can estimate the number of containers (N_c).

$$N_c = PQ_{len} / B_size \text{-----Equation 3-3}$$

In case the time taken to service the requests by queuing on existing containers is lower than the cold-start delays, spawning a new container would be deemed unnecessary. Therefore, the function takes into account, the delay incurred in terms of queuing the request for a longer time vs cold-start (C_d). The queuing delay threshold D_f for worker W is calculated using total number of requests that can be executed without violating SLOs (L), and total time required to satisfy all pending requests (T_d) as shown on Equation below:

$$L = \sum_{i=1}^N Bsize_i \text{-----Equation 3-4}$$

$$T_d = PQ_{len} * W_r \text{-----Equation 3-5}$$

$$D_f = \frac{T_d}{L} \text{-----} \text{Equation 3-6}$$

where W_r is the per worker response latency, N is the number of containers in W . W_r for a worker is defined as the sum of its allocated slack and execution time. If $D_f > C_d$, then LB spawn additional containers (N_c) for each worker.

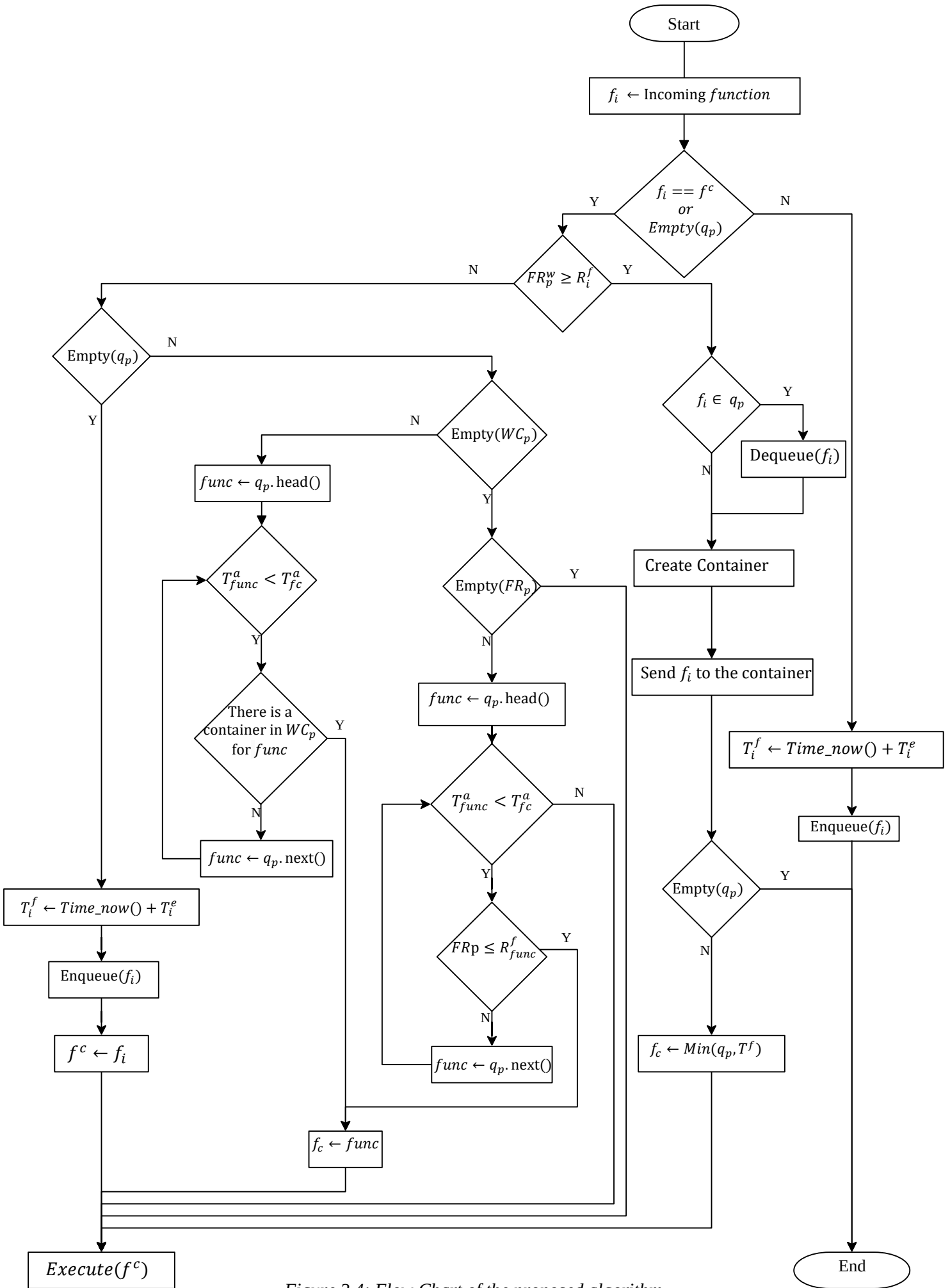


Figure 3.4: Flow Chart of the proposed algorithm

The ETLAS workflow is shown in Fig. 3.4. As a result, the entered function (f_i) can only be executed if either the queue is empty or it is the candidate function (f^c), which is chosen by the scheduler. Otherwise, related T_i^f is computed and attached to f_i and queued (step 2).

Next, the availability of resources to execute f_i is determined. If so, f_i is placed in the designated container and, if the queue is not empty, the following function is chosen as the candidate function using the minimum T^f (step 3). Otherwise, the scheduler determines whether the queue is empty (meaning f_i is the first function that is enqueued and its T^f should be calculated) or not (meaning the function is f^c) in the event that there are no resources available to execute f_i . As a result, step 4 is followed to find a warm instance or free resources for another function in the queue.

According to Table 3.1, T_i^f is the summation of arrival time (T_i^a) and the predicted execution time (T_i^e) of f_i . Therefore, ETLAS selects a function with minimum T^f . In other words, the function with the shortest execution time among those called almost simultaneously is chosen, while the function with the earliest arrival time among those invoked at larger intervals is chosen. Because of this, the larger functions are not starved of resources and the smaller functions are not stuck behind larger ones in terms of execution time.

3.10. Research Metrics

This research will analyze the advanced techniques, and algorithms in order to design and evaluate the proposed algorithm in terms of selected metrics. The criteria to evaluate the proposed approach are:

- ✧ **Resource utilization-** is a measure of how much of the available cloud infrastructure and services are being used.
- ✧ **Average response time-** is the time it takes for a user request to be processed and a response returned. It is typically measured in milliseconds, with lower values indicating better performance.
- ✧ **Throughput-** refers to the amount of data that can be transferred from a cloud system to its user within a certain period of time.
- ✧ **Average waiting time** - is the average time a request waits in queue before an invoker answers.
- ✧ **Cold starts** - refer to the delays that can occur when an idle function instance has to be initialized and spun up in order to respond to an incoming request.

The scheduling algorithm tries to maximize and minimize the following:

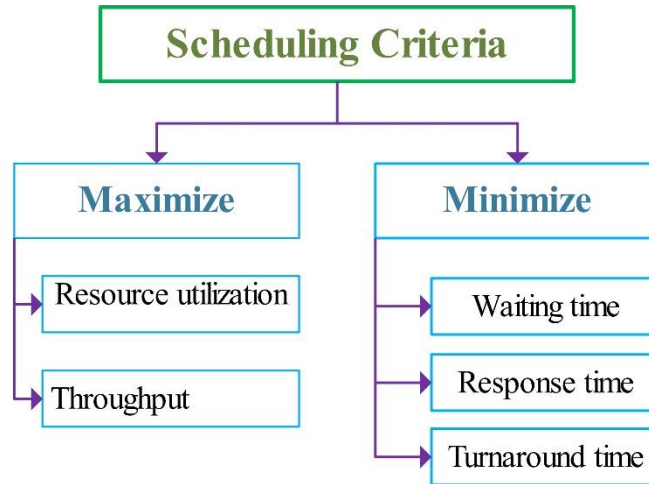


Figure 3.5: Our Scheduling Criteria

3.11. Testing and Methods of Evaluation

The proposed method is examined and assessed using the mentioned evaluation metric. To determine whether stated purpose the researches was achieved or not, the projected approach is also compared to other peer algorithms and the original algorithm. The final document is then created in this manner after the result has been documented using the graph and other techniques.

3.12. Development Tools and Techniques

For implementation and testing purposes, different types of tools are applied to implement the logic into a solution. The list is shown as follows.

❖ Implementation Techniques and Tools

Visual Studio Code is a very powerful code-focused development environment expressly designed to make it easier to write web, mobile, and cloud applications using languages that are available to different development platforms and to support the application development life cycle with a built-in debugger and integrated support for the popular Git version control engine(del Sole, 2021).

The main advantages of Visual Studio Code are:

- ✧ IntelliSense Auto-Completion - this feature provides intelligent auto-completion based on language semantics to help speed up development tasks.

- ✧ Cross-Platform Support - It can be used on Windows, MacOS, and Linux operating systems which makes it convenient for developers who use different platforms or have multiple machines at home or work.
- ✧ Built-in Debugging - Its powerful debug tool allows users to quickly identify bugs in their application code so they can be fixed more efficiently with fewer headaches.
- ✧ Large Ecosystem of Extensions and Add-ons - Visual Studio Code has an extensive library of extensions that enable developers to customize the IDE according to their specific needs.
- ✧ Separate Workspaces for Different Projects - This feature helps organize projects into separate workspaces allowing the user quickly switch between them when needed during development workflows

CHAPTER FOUR

RESULTS AND DISCUSSIONS

4.1. Introduction

In terms of effective resource usage, average latency, response time, throughput, and cold starts, the impact of our proposed approach is evaluated. In addition, the proposed algorithm is compared to scheduling benchmark algorithms such as SJF, FCFS. We also compared our proposed approach to the execution-time-aware method called ETAS.

4.2. Implementation of Hybrid Algorithm

We used Apache OpenWhisk, an open-source, serverless cloud platform, to implement and evaluate ETLAS. The implementation of ETLAS and other systems is explained in the following sections. We then describe our test setup and our benchmarks and finally report on the results of our experiments.

The implementation of the proposed hybrid solution is in the invoker component of Apache OpenWhisk platform (Sect. 2.7.4). To evaluate the algorithm, we have designed a new benchmark suit with four different functions based on benchmarks presented by (Kuntsevich et al., 2018) in both Python and PHP. In order to study the behavior of the proposed approach (ETLAS) under various workloads, we have taken into consideration two classes of experiments. The impacts of various factors were analyzed separately in the first class of experiments, whereas the combined effects of all the factors were evaluated in the second class. The scheduling methods SJF (Shortest Job First) (Silberschatz et al., 2018) and MQS (Multi Queue Scheduling) (Karthick et al., 2014) have also been implemented in OpenWhisk in order to give a credible evaluation. In our experiments, we examined different scheduling approaches using four fundamental measured parameters: worker throughput, average waiting time, resource utilization, and the frequency of using warm instances. According to the capabilities of our system, three separate users are created, and each function is allocated to each user with a different resource setting and memory allocation.

4.3. Experimental setup

According to the OpenWhisk architecture, all invokers hosting the serverless function containers are copies of the invoker component statements (Sect. 2.7.4). Since we are looking for the average waiting times of functions in the queue, throughput, resource utilization, and

the rate of reuse of warm containers within the invokers, the following experiments are performed with a single invoker, with respect that all invokers are homogeneous.

4.4. Working Environments

We have deployed and configured the Apache OpenWhisk with the default settings on a server machine with two 12 cores of Intel Xeon CPUs with 2.67 GHz, and 8 GB of memory. We have used Ubuntu 22.04 kernel version 5.15 (Linux-generic) as an operating system. Requests have been generated within the same host.

Furthermore, a logger has been implemented in the ContainerPool to record the details of each called action and the queue status, and a watcher to check the invoker's memory usage. A Python program is used to evaluate the logs once they have been put in the database. The logger records the username, the function name, the memory use, the arrival time, the queue number, the queue length, the exit time, the queue wait time, the total wait time (out of queue), the container status, and the estimated and real execution times for each function. The watcher is designed to store the total memory usage of the system per second. Average wait time, average execution time, throughput, container states, and memory usage are calculated from these parameters. Three categories of experiments were provided to evaluate and study the behavior of the proposed solution (ETLAS) and allow a reliable comparison between ETLAS and other schemes. In the following section, each section is explained and the related results are reported and discussed.

4.5. Experiments and results

Time intervals of invocations, execution times, workload, cold start, and function resource settings are the factors that affect function wait times. To allow for an accurate assessment, we defined two different experiments to calculate the impact of each factor. In the first class, each factor was measured separately, and in the second, the effect of combining all factors was examined.

4.5.1. Experiment One

As previously mentioned, several factors contribute to function waiting times. Therefore, in each test in this category, we examine the effects of the average time intervals of calls, average execution times of functions, call rate, cold start, and function resource settings. All scheduling schemes, except FCFS, must be aware of function execution times. Therefore, we use the sleep function whose execution time is known in advance to eliminate the prediction error. To keep the number of containers and resource settings constant, we used a

user and action with 512MB of memory. To avoid the cold start, delay effect, all sandboxes were started before the start of the test. As a result, all containers are equal and warm due to the use of one action. The exponential distribution is used to generate the time intervals and execution times of functions. Each test is repeated 25 times for each scheme.

✧ Time interval impact

In this test, we look for the effects of different time intervals on the average waiting times in each scheme. Therefore, the number of requests is constant and equal to 25, which means that 25 requests are sent in each iteration. The average execution times of functions are also constant, which results from the exponential distribution with $\theta = 30$ s. The time intervals are generated according to exponential distribution ($\theta = 0, 2, 4, 6$ s).

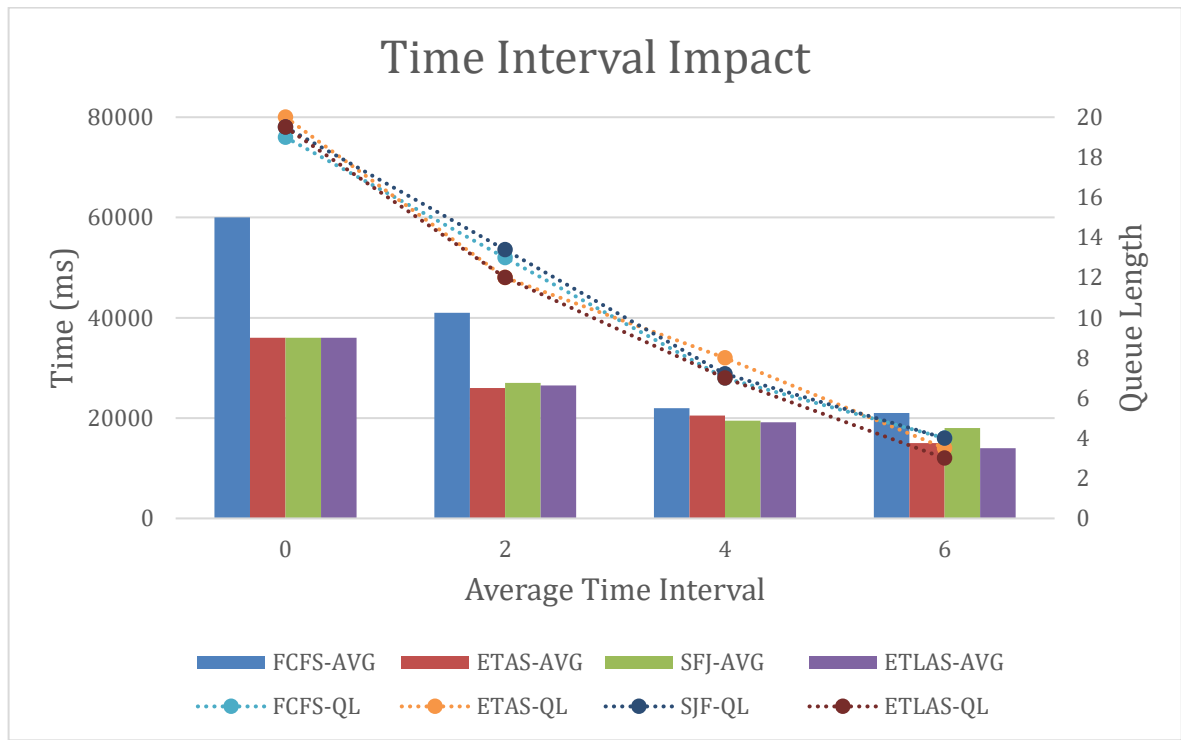


Figure 4.1: Average waiting times and standard deviations for queued functions with maximum queue length at various average time intervals

Figure 4.1 shows the average waiting times of actions and the standard deviations along with the maximum queue lengths in each scheme. Given that these times are for the waiting times of functions in the queue. ETLAS, SJF and ETAS behave the same with simultaneous requests ($\theta = 0$). Because with the same arrival times, T^f is equal to T^e (Table 3.1). With the increase in time intervals, SJF has a lower waiting time than ETLAS ($\theta = 4$). In contrast, when ($\theta = 6$), it seems that ETLAS has less waiting time. Because SJF always gives priority

to functions with lower execution times, it can impose long waiting times on functions with higher execution times, resulting in an increase in average waiting times. The results also show that ETLAS has the lowest standard deviation compared to other schemes, showing that no function encounters high latency.

✧ Execution time impact

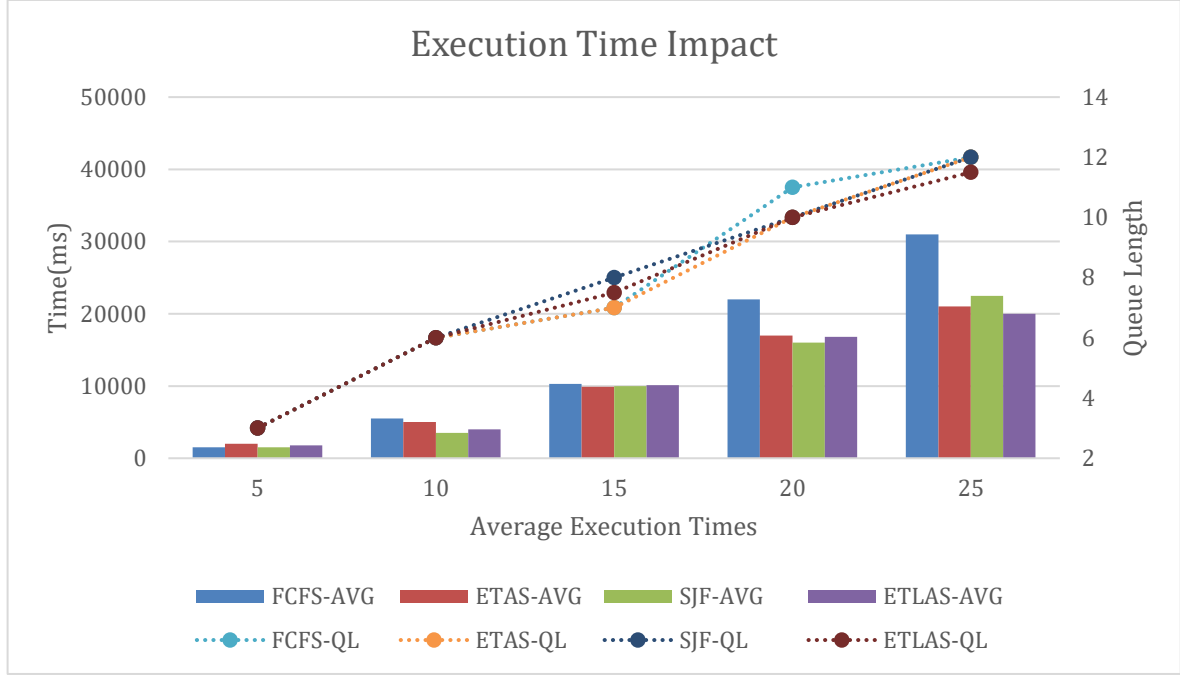


Figure 4.2: Average waiting times and standard deviations for queued functions with maximum queue length in various average execution times

Execution time variation is another factor affecting waiting times of the invocations. In this test, the number of requests in each iteration is 25 and the average time intervals are constant ($\theta = 2$ s), while the average execution times are increasing ($\theta = 5, 10, 15, 20, 25$ s). A look at Figure 4.2 shows that ETLAS has the lowest waiting times, especially with higher average execution times. At $\theta = 5$, the maximum number of functions in the queue is about 1, which is not enough to compare the schemes. As expected, with the increase in execution times, ETLAS and ETAS outperform other schemes, while FCFS encounters higher latency. At $\theta = 25$, ETLAS shows lower latency than ETAS. As execution times increase, some functions will have longer execution times. Therefore, as SJF, prioritizes the functions with shorter execution times, the longer ones would suffer from starvation. In addition, the results also show that the longer the execution times, the lower the standard deviation in ETLAS, which means that ETLAS tries to treat all actions fairly.

✧ Workload impact

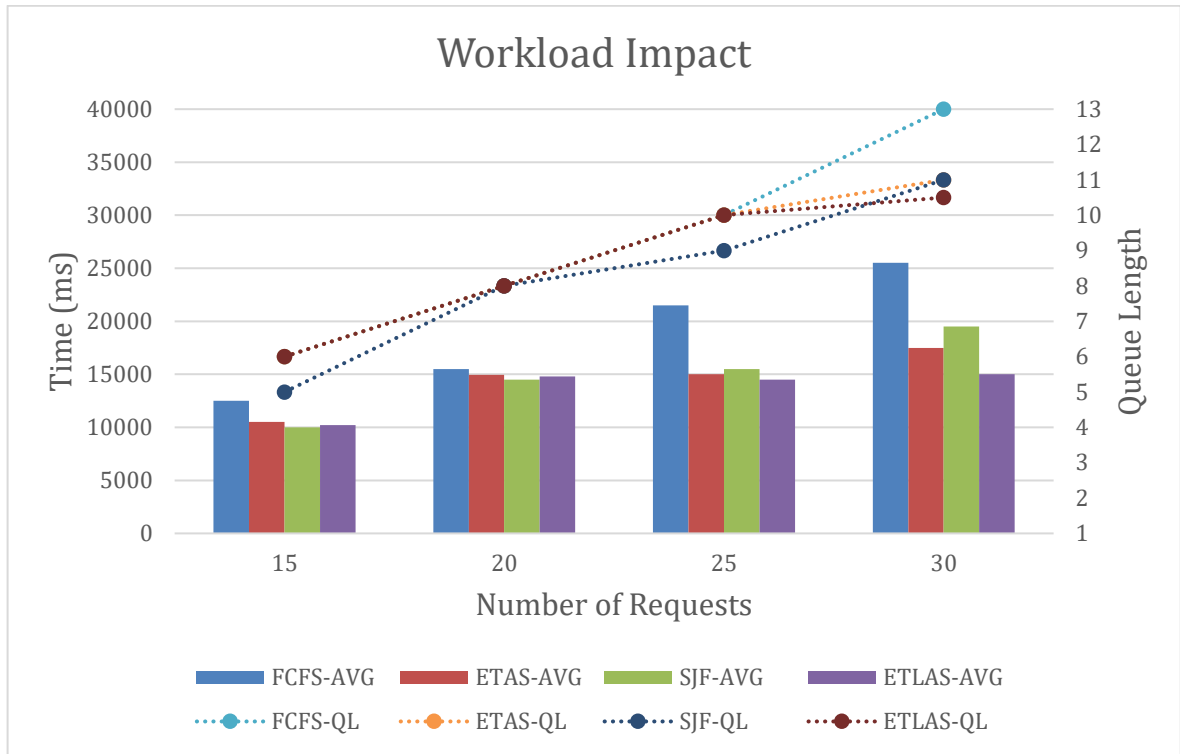


Figure 4.3: Average waiting times and standard deviations for queued actions with a maximum queue length in different workloads in each scheme

Figure 4.3 shows that FCFS has the highest latency as the request load increases, while ETLAS performs better than other schemes. Additionally, SJF has the most standard deviations in latency, while ETLAS has the lowest. As mentioned in previous tests, ETLAS avoids long latency for the functions by considering the arrival time when choosing the next function as a candidate function.

4.5.2. Experiment Two

This test combines all the factors that affect function call latencies. Also, the other functions are used (Section 3.5) whose execution times are unknown in advance. ETLAS consists of two main parts, prediction and scheduling. Therefore, before the tests of this class are performed, the prediction part is examined and the results are reported.

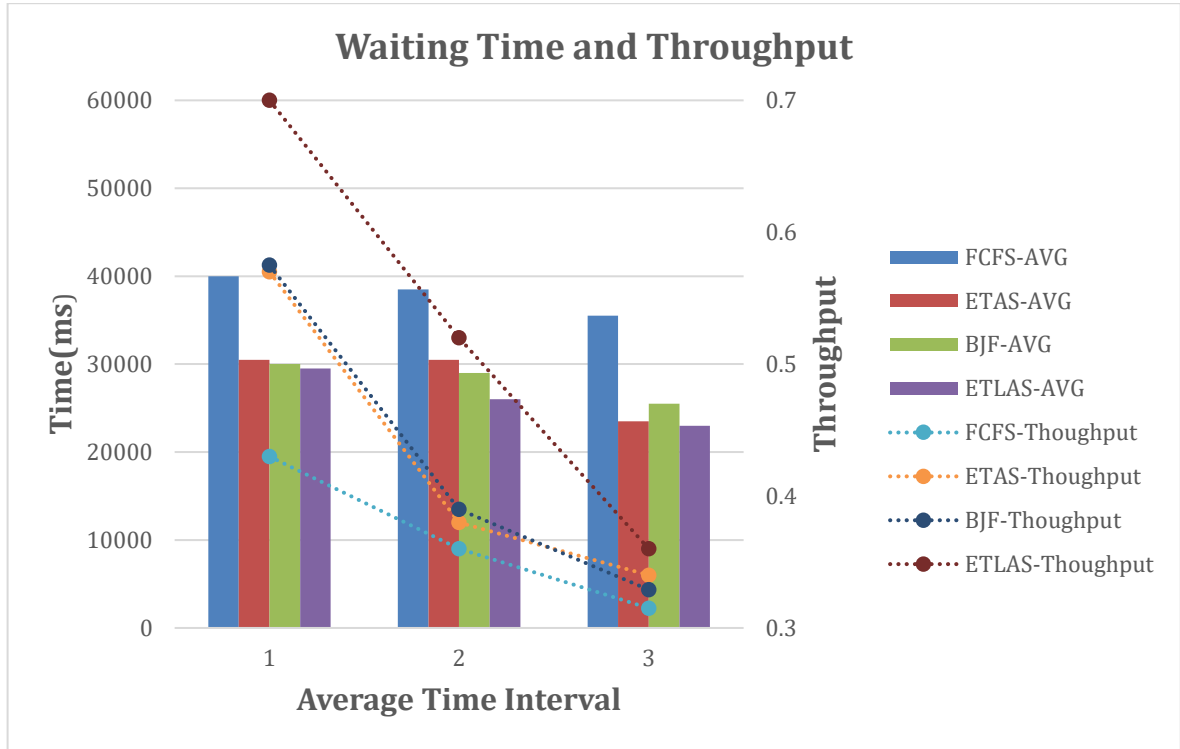


Figure 4.4: Average waiting times and standard deviations for queued actions and invoker throughput at various time intervals and schemes

Figure 4.4 illustrates the average execution times of the called functions in each test and scheme. To provide for a situation where functions are forced to remain in the queue, the average execution times increase with an increase in the time interval. However, with increasing time intervals, the length of the queue decreases slightly.

✧ Cold start and resource setting impacts

When an action is invoked and there is an execution instance, it is immediately associated (warm) with the action. Otherwise, a container is created (cold start). In case there are not enough resources to create a container and one or more containers from paused containers (if any) are terminated to free enough resources for the new (recreated) container. And if there are not enough resources for the paused container, the action is queued.

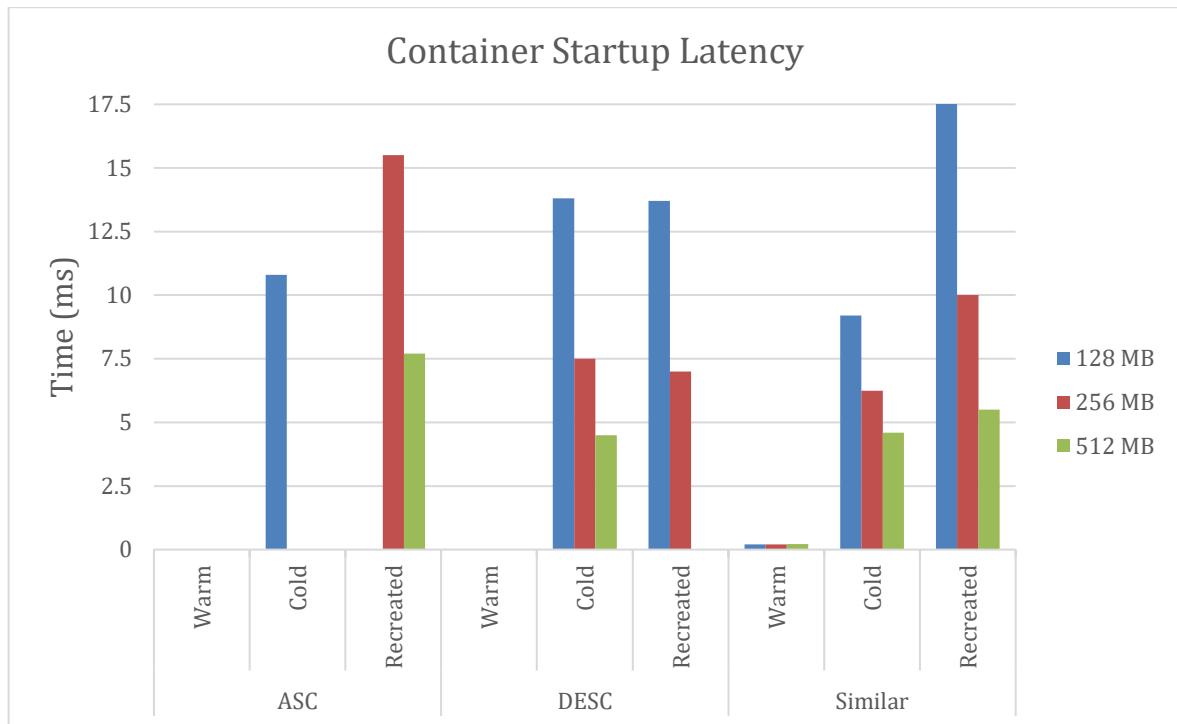


Figure 4.5: Container startup latency in different states with different sizes of memory in three invocation patterns

Figure 4.5 shows container startup latency in different states and memory sizes. We established three distinct scenarios, namely Similar, ASC, and DES, to test startup latency. In a similar test, startup latencies are calculated based on function memory, and a function with the same memory is replaced (recreated) with the previous one. It is clear that the warm container has by far the lowest latency while recreating a container imposes the longest latency.

With ASC and DES, containers are replaced by a series of containers with different memory sizes. In ASC, all resources are initially occupied with 128 MB containers. Thereafter, 256MB and 512MB functions are called, respectively, while the reverse procedure is followed for DES. Figure 9 shows that recreating the 256MB containers encounters higher latency in the ASC test than in the DES test. In fact, the ASC test to initialize a 256MB container should terminate two 128MB containers, while the DES would terminate a 512MB container. Overall, we can observe that startup times for containers vary and rely on the situation. It is obvious that the size of memory directly influences startup time.

This test examines the influence of cold start and different resource settings. This uses different users, roles, and resource settings, and containers don't get warm at first. Three different users are used along with sleep functions in Python and PHP languages with

memory sizes of 128, 256, and 512 MB. The number of requests is 40, the average time between requests is 2 s ($\theta = 2$), and the average execution time is 20 s ($\theta = 20$). On each scheme, the test is run 40 times.

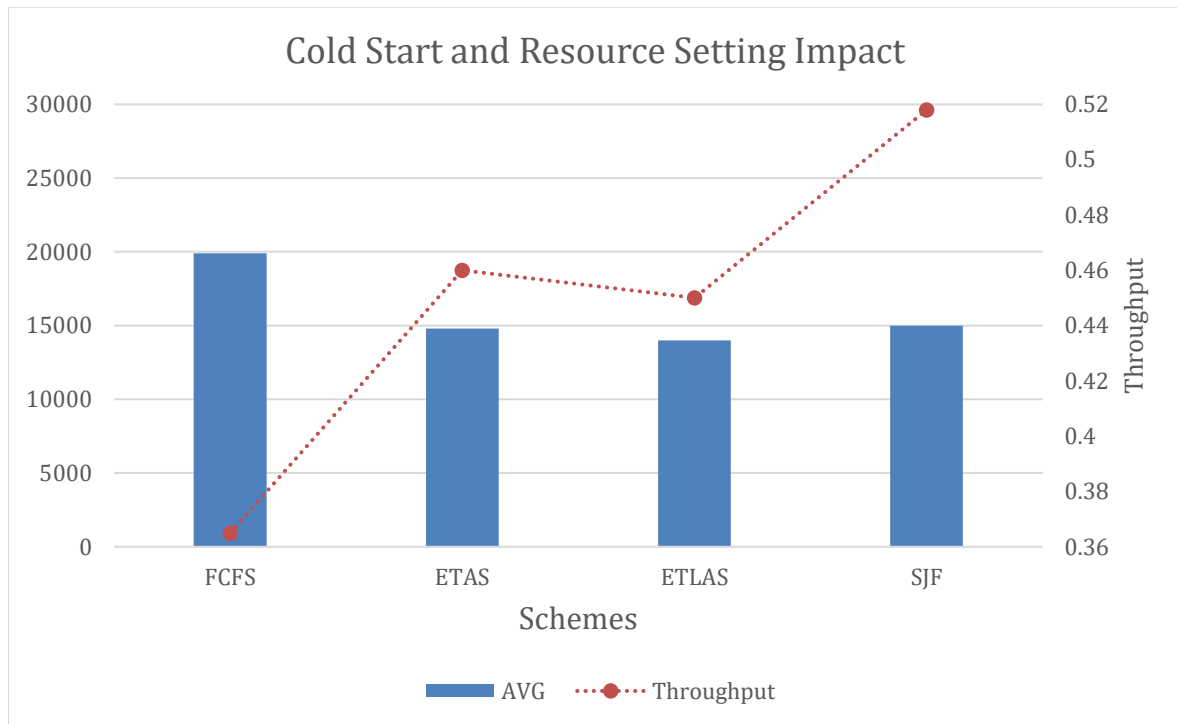


Figure 4.6: Average waiting times and standard deviations for actions in the queue and throughput of the invoker in each scheme

Figure 4.6 shows the average waiting times and standard deviation for queued actions and invoker throughput in each scheme. ETLAS, ETAS, and SJF require roughly equal latency, while SJF has the largest standard deviation and throughput. Throughput shows the amount of work done in one second. FCFS has the longest latency with the lowest throughput.

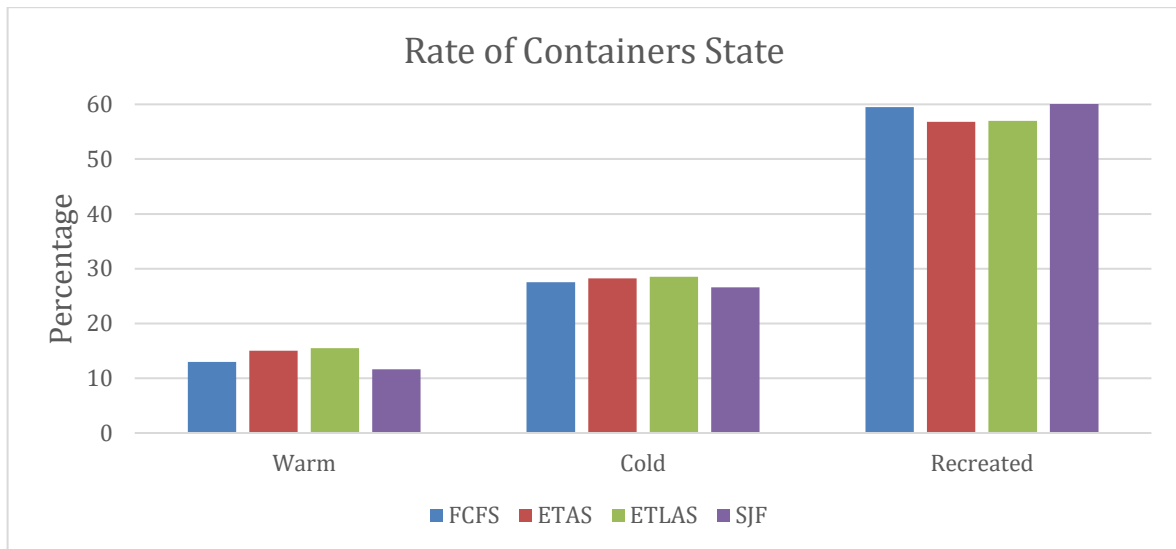


Figure 4.7: Rate of containers states in each scheme

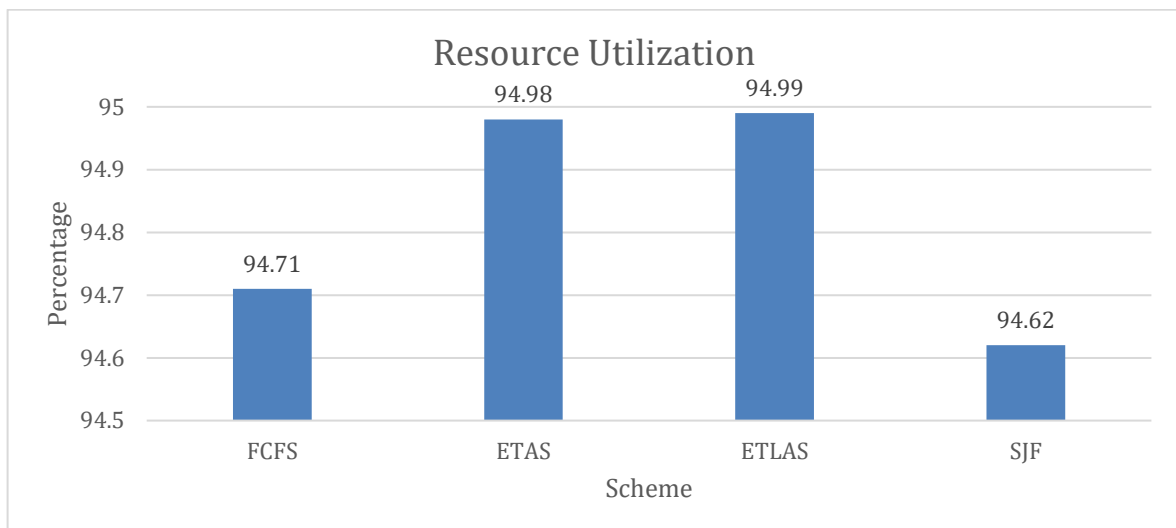


Figure 4.8: Rate of resource utilization in each scheme

Figure 4.7 and 4.8 shows the rate of container states and resource utilization in each scheme respectively. Looking at the charts, ETLAS shows the greatest warm container reuse and resource utilization by prioritizing the functions with warm containers and the functions that have enough resources to run.

Table 4.1: Input, execution times, and invocations percentage of benchmark's functions

Inputs	Functions			Execution Time (s)	Invocations Percentage		
	primeNumber	factorial	multiplyMatrix		$\theta=1$	$\theta=2$	$\theta=3$
1	1×10^3	5×10^4	1×10^2	0.5	15	10	5
2	2×10^3	10×10^4	2×10^2	2	25	15	10
3	4×10^3	15×10^4	3×10^2	5	25	25	20
4	6×10^3	20×10^4	4×10^2	10	20	25	25
5	8×10^3	25×10^4	5×10^2	30	10	20	25
6	10×10^3	30×10^4	6×10^2	60	5	10	15

Table 4.2: Results of Experiment for time interval (1s)

Time Interval	$\theta=1$			
Algorithms	FCFS	ETAS	ETLAS	SJF
Average waiting time(ms)	40436	31040	29849	29530
Standard deviation	20707	20823	21221	25458
Average execution time(ms)	17360	17271	16687	17213
Throughput	0.364	0.542	0.548	0.755
Resource utilization (%)	94.77	95.78	95.8	94.84
Warm container usage (%)	5.87	9.48	9.5	6.42
Containers startup latencies(ms)	10975	14622	13664	14501

Table 4.3: Results of Experiment for time interval (2s)

Time Interval	$\theta=2$			
Algorithms	FCFS	ETAS	ETLAS	SJF
Average waiting time(ms)	37736	30777	28321	25941
Standard deviation	19857	21738	21864	26976
Average execution time(ms)	24431	25804	24307	24300
Throughput	0.281	0.355	0.385	0.491
Resource utilization (%)	94.87	95.42	95.85	94.9
Warm container usage (%)	6.93	9.34	9.45	6.82
Containers startup latencies(ms)	9784	11252	10952	10769

Table 4.4: Results of Experiment for time interval (3s)

Time Interval	$\theta=3$			
Algorithms	FCFS	ETAS	ETLAS	SJF
Average waiting time(ms)	35459	23724	23103	26432
Standard deviation	16394	17577	17986	27265
Average execution time(ms)	29991	30992	30347	30933
Throughput	0.241	0.31	0.293	0.354
Resource utilization (%)	95.01	95.43	95.95	94.81
Warm container usage (%)	6.54	8.47	8.5	6.75
Containers startup latencies(ms)	8709	8760	8623	9063

Table 4.2, 4.3 and 4.4 demonstrate the summary of experiment results to compare the parameters precisely.

For this test, three users, three Python functions: primeNumber, which returns the nth prime number, factorial, and multiplyMatrix as well as three resource settings are used (128 MB, 256 MB, and 512 MB). Hence, 27 distinct actions are employed. Average time intervals are 1, 2, and 3 s, and for each function, 6 different inputs are used, resulting in a range of execution times (about 0.5, 2, 5, 10, 30, and 60 s). The number of requests is fixed at 50. As a result, three tests are included in this category. Each test is run 30 times on each scheme. Table 4.1 lists the functions, inputs, their execution times, and invocations percentage in each time interval. The percentages are chosen to create a system overload. To properly compare the parameters, the results of Figs. 4.1–4.8 are shown in the table 4.2–4.4. As shown in Fig. 4.5, the memory, container image, and recreated state (recreating from larger (based on memory) container to smaller one or reverse) have all had an impact on the delay.

CHAPTER FIVE

CONCLUSIONS AND RECOMMENDATIONS

5.1. Conclusion

Scheduling workloads on appropriate host nodes based on resource requirements while making efficient use of existing resources is an important task that both developers and cloud providers or system owners are concerned with. When resource requirements exceed available resource capacity, scheduling also includes deciding the order in which applications will run. While the developer can demand specific QoS assurances, it is also crucial for the provider to properly manage resources, which is one of the main goals of resource management. In this study, we proposed a hybrid scheduling algorithm called ETLAS (Execution-Time and Load-Aware Scheduling) for serverless computing. It is a predictive execution-time aware scheduling and function-aware container scaling approach that benefits both providers and customers by reducing function response times by reducing average waiting times and improving worker node throughput and resource utilization. The execution times of functions are estimated by logging previous executions and the scheduler selects functions based on their arrival times and execution times to reduce the waiting times without starving them.

The algorithm was implemented in the worker node component of Apache OpenWhisk and designed a benchmark to provide an evaluation. In addition, to investigate the behavior of ETLAS and provide an acceptable evaluation other scheduling algorithm such as (Shortest-Job-First and Execution Time Aware) have been implemented alongside the OpenWhisk default worker scheduler scheme, which is First-Come-First-Serve, to be able to compare ETLAS with others. The experimental results demonstrate that ETLAS outperformed other scheduling schemes and achieved all objectives that we had set for it. ETLAS reduces the average waiting times of invocations by up to 34% and increases the throughput of the worker node by up to 42% in comparison with FCFS and also has lower average waiting times in comparison with SJF by up to 12% in higher time intervals of requests by avoiding starvation.

5.2. Contribution

Resource utilization - The proposed approach increases resource utilization for each individual worker by balancing and scaling in-out containers. Our proposed scheme

improves container utilization by prioritizing the scheduling of functions on a warm instance or free resources of functions. This results in an average decrease in the number of containers spawned and increases the usage of containers allocated to the worker node.

Response time - The proposed method reduces the function's response time by allocating functions based on their execution times and arrival time which reduces functions' average waiting time and increases throughput.

Cold starts - Since the warm container has the least latency by far while recreating a container imposes the highest latency, our proposed approach employs a warm container than other algorithm and hence have the least container startup latencies.

5.3. Future Works

In the future, we will study, develop, and apply other techniques to address additional scheduling problems. We recommended that another motivated researcher develop machine learning models to predict future workload in a serverless cloud computing environment. Furthermore, it is better to develop and test the proposed approach in a real environment than to implement it with a simulation framework.

Special Acknowledgment

This research project is funded by Adama Science and Technology University under the grant number:

ASTU/SM-R/501/22

Adama, Ethiopia

References

- Adzic, G., & Chatley, R. (2017). *Serverless computing: economic and architectural impact*. 884–889. <https://doi.org/10.1145/3106237.3117767>
- Alvarado, A. (2019). *Serverless vs. FaaS: A Beginner's Guide - Liquid Web*. <https://www.liquidweb.com/kb/serverless-vs-faas-a-beginners-guide/>
- Apache OpenWhisk: (n.d.). <https://openwhisk.apache.org/documentation.html>. Retrieved January 31, 2023, from <https://openwhisk.apache.org/documentation.html>
- Aumala, G., Boza, E., Ortiz-Aviles, L., Totoy, G., & Abad, C. (2019). Beyond load balancing: Package-aware scheduling for serverless platforms. *Proceedings - 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, 282–291. <https://doi.org/10.1109/CCGRID.2019.00042>
- Azure Functions: (n.d.). *Azure Functions – Serverless Apps and Computing | Microsoft Azure*. Retrieved March 21, 2022, from <https://azure.microsoft.com/en-us/services/functions/#overview>
- Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017). Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing*, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1
- Banaei, A., & Sharifi, M. (2021). ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform. In *Journal of Supercomputing* (Issue 0123456789). Springer US. <https://doi.org/10.1007/s11227-021-04057-z>
- Bavosa, A. (2019). *What is a Serverless Function? | PubNub*. <https://www.pubnub.com/blog/what-is-a-serverless-function/>
- Boza, E. F., Abad, C. L., Villavicencio, M., Quimba, S., & Plaza, J. A. (2018). Reserved, on demand or serverless: Model-based simulations for cloud budget planning. *2017 IEEE 2nd Ecuador Technical Chapters Meeting, ETCM 2017, 2017-January*, 1–6. <https://doi.org/10.1109/ETCM.2017.8247460>
- BROWN, S. (2020). *FaaS Vs. Serverless: Resolving the Dilemma*. <https://stackify.com/faas-vs-serverless-resolving-the-dilemma/>

- Calabrese, B., & Cannataro, M. (2015). Cloud Computing in Healthcare and Biomedicine. *Scalable Computing: Practice and Experience*, 16(1), 1–18. <https://doi.org/10.12694/SCPE.V16I1.1057>
- Carreira, J., Kohli, S., Bruno, R., & Fonseca, P. (2021). From warm to hot starts: Leveraging runtimes for the serverless era. *HotOS 2021 - Proceedings of the 2021 Workshop on Hot Topics in Operating Systems*, 58–64. <https://doi.org/10.1145/3458336.3465305>
- Castro, B. Y. P., Ishakian, V., Muthusamy, V., & Slominski, A. (2020). *The Rise of Serverless Computing*.
- Chaudhary, S., Somani, G., & Buyya, R. (2017). Research Advances in Cloud Computing. *Research Advances in Cloud Computing*, 18–36. <https://doi.org/10.1007/978-981-10-5026-8>
- Chou, D. (2018). *Cloud Service Models (IaaS, PaaS, SaaS)*. <https://dachou.github.io/2018/09/28/cloud-service-models.html>
- Chrissy Kidd, S. W. (2021). *Serverless vs Function-as-a-Service (FaaS): What's The Difference?* – BMC Software | Blogs. <https://www.bmc.com/blogs/serverless-faas/>
- Cloud Computing Market Size Report, 2022-2030. (n.d.). Retrieved December 19, 2022, from <https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry>
- Cloudflare: (n.d.). *What is Function-as-a-Service (FaaS)?* | Cloudflare. Retrieved March 22, 2022, from <https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/>
- Collis, J. and Hussey, R. (2014). *Praise for Business Research 4th edition* :
- Cooney, M. (2022). *Gartner: By 2025 half of enterprise IT spending will be for cloud* | Network World. <https://www.networkworld.com/article/3649791/gartner-by-2025-half-of-enterprise-it-spending-will-be-for-cloud.html>
- Das, A., Leaf, A., Varela, C. A., & Patterson, S. (2020). Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. *IEEE International Conference on Cloud Computing, CLOUD, 2020-Octob(v)*, 609–618. <https://doi.org/10.1109/CLOUD49709.2020.00090>

- Daw, N., Bellur, U., & Kulkarni, P. (2020). Xanadu: Mitigating cascading cold starts in serverless function chain deployments. *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*, 356–370. <https://doi.org/10.1145/3423211.3425690>
- de Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., & Zavattaro, G. (2022). A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling. *Proceedings - IEEE International Conference on Web Services, ICWS 2022*, 337–342. <https://doi.org/10.1109/ICWS55610.2022.00056>
- del Sole, A. (2021). Introducing Visual Studio Code. *Visual Studio Code Distilled*, 1–15. https://doi.org/10.1007/978-1-4842-6901-5_1
- Education, I. C. (2019). *What is FaaS (Function-as-a-Service)?* | IBM. <https://www.ibm.com/cloud/learn/faas>
- Fu, Y., Liu, L., Wang, H., Cheng, Y., & Chen, S. (2022). *SFS: Smart OS Scheduling for Serverless Functions*. 1–15. <http://arxiv.org/abs/2209.01709>
- Fuerst, A., & Sharma, P. (2021). FaasCache: Keeping serverless computing alive with greedy-dual caching. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 386–400. <https://doi.org/10.1145/3445814.3446757>
- Gartner: (n.d.). *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly \$600 Billion in 2023*. Retrieved December 19, 2022, from <https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023>
- Geng, X., Ma, O., Pei, Y., Xu, Z., Zeng, W., & Zou, J. (2018). Research on Early Warning System of Power Network Overloading under Serverless Architecture. *2nd IEEE Conference on Energy Internet and Energy System Integration, EI2 2018 - Proceedings*. <https://doi.org/10.1109/EI2.2018.8582355>
- GHOSH, A. (2019). *What is the Difference Between FaaS and Serverless?* <https://thecustomizewindows.com/2019/04/what-is-the-difference-between-faas-and-serverless/>

- Google Cloud Functions: (n.d.). *Cloud Functions* | Google Cloud. Retrieved March 21, 2022, from <https://cloud.google.com/functions/>
- Großmann, M., Ioannidis, C., & Le, D. T. (2019). Applicability of serverless computing in fog computing environments for IoT scenarios. *UCC 2019 Companion - Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 29–34. <https://doi.org/10.1145/3368235.3368834>
- Gunasekaran, J. R., Thinakaran, P., Kandemir, M. T., Urgaonkar, B., Kesidis, G., & Das, C. (2019). Spock: Exploiting serverless functions for SLO and cost aware resource procurement in public cloud. *IEEE International Conference on Cloud Computing, CLOUD, 2019-July*, 199–208. <https://doi.org/10.1109/CLOUD.2019.00043>
- Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., & Das, C. R. (2020a). Fifer: Tackling resource underutilization in the serverless era. *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*, 280–295. <https://doi.org/10.1145/3423211.3425683>
- Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., & Das, C. R. (2020b). Fifer: Tackling resource underutilization in the serverless era. *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*, 280–295. <https://doi.org/10.1145/3423211.3425683>
- Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., & Das, C. R. (2020c). Fifer: Tackling resource underutilization in the serverless era. *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*, 280–295. <https://doi.org/10.1145/3423211.3425683>
- Hassan, H. B., Barakat, S. A., & Sarhan, Q. I. (2021). Survey on serverless computing. *Journal of Cloud Computing*, 10(1). <https://doi.org/10.1186/s13677-021-00253-7>
- Hoseiny Farahabady, M. R., Lee, Y. C., Zomaya, A. Y., & Tari, Z. (2017). A QoS-aware resource allocation controller for function as a service (FaaS) platform. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10601 LNCS, 241–255. https://doi.org/10.1007/978-3-319-69035-3_17
- Hunhoff, E., Irshad, S., Thurimella, V., Tariq, A., & Rozner, E. (2020). Proactive Serverless Function Resource Management. *WOSC 2020 - Proceedings of the 2020 6th*

- International Workshop on Serverless Computing, Part of Middleware 2020*, 61–66.
<https://doi.org/10.1145/3429880.3430102>
- IBM Cloud Functions: (n.d.). *IBM Cloud Functions* | IBM. Retrieved March 18, 2022, from
<https://www.ibm.com/cloud/functions>
- Jambunathan, B., & Yoganathan, K. (2018). Architecture Decision on using Microservices or Serverless Functions with Containers. *Proceedings of the 2018 International Conference on Current Trends towards Converging Technologies, ICCTCT 2018*.
<https://doi.org/10.1109/ICCTCT.2018.8551035>
- Jia, Z., & Witchel, E. (2021). Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 152–166.
<https://doi.org/10.1145/3445814.3446701>
- Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. (2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing*.
- Kaffes, K., Yadwadkar, N. J., & Kozyrakis, C. (2019a). Centralized Core-granular Scheduling for Serverless Functions. *SoCC 2019 - Proceedings of the ACM Symposium on Cloud Computing*, 158–164. <https://doi.org/10.1145/3357223.3362709>
- Kaffes, K., Yadwadkar, N. J., & Kozyrakis, C. (2019b). Centralized Core-granular Scheduling for Serverless Functions. *SoCC 2019 - Proceedings of the ACM Symposium on Cloud Computing*, 158–164. <https://doi.org/10.1145/3357223.3362709>
- Karthick, A. v., Ramaraj, E., & Subramanian, R. G. (2014). An efficient multi queue job scheduling for cloud computing. *Proceedings - 2014 World Congress on Computing and Communication Technologies, WCCCT 2014*, 164–166.
<https://doi.org/10.1109/WCCCT.2014.8>
- Kim, Y. K., Hoseinyfarahabady, M. R., Lee, Y. C., & Zomaya, A. Y. (2020). *Automated Fine-Grained CPU Cap Control in Serverless Computing Platform*. 31(10), 2289–2301.

- Kulkarni, S. G., Liu, G., Ramakrishnan, K. K., & Wood, T. (2019). Living on the edge: Serverless computing and the cost of failure resiliency. *IEEE Workshop on Local and Metropolitan Area Networks*, 2019-July. <https://doi.org/10.1109/LANMAN.2019.8846970>
- Kumar, P. (2019). *QoS and efficiency for FaaS platforms*.
- Kuntsevich, A., Nasirifard, P., & Jacobsen, H. A. (2018). Demo abstract: A distributed analysis and benchmarking framework for apache openwhisk serverless platform. *Middleware 2018 - Proceedings of the 2018 ACM/IFIP/USENIX Middleware Conference (Posters)*, 3–4. <https://doi.org/10.1145/3284014.3284016>
- Leech, A. J. (n.d.). Sampling Designs in Qualitative Research: Making the Sampling Process More Public. *The Qualitative Report*, 12(2), 238–254. <https://doi.org/10.46743/2160-3715/2007.1636>
- Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., & Pallickara, S. (n.d.). *Serverless Computing: An Investigation of Factors Influencing Microservice Performance*. <https://doi.org/10.1109/IC2E.2018.00039>
- Mampage, A., Karunasekera, S., & Buyya, R. (2021). A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Computing Surveys*, 1(1).
- Mampage, A., Karunasekera, S., & Buyya, R. (2022). A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Computing Surveys*, 54(11s), 1–36. <https://doi.org/10.1145/3510412>
- Mell, P., & Grance, T. (n.d.). *The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology*. <https://doi.org/10.6028/NIST.SP.800-145>
- Moravcik, M., Segec, P., & Kontsek, M. (2018). Overview of Cloud Computing Standards. *ICETA 2018 - 16th IEEE International Conference on Emerging ELearning Technologies and Applications, Proceedings*, 395–402. <https://doi.org/10.1109/ICETA.2018.8572237>

- Press release. (2013). *Amazon Web Services Announces Amazon Kinesis* | *Amazon.com, Inc.* - *Press Room*. <https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-announces-amazon-kinesis>
- Przybylski, B., Zuk, P., & Rządca, K. (2021). Data-driven scheduling in serverless computing to reduce response time. *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, 206–216. <https://doi.org/10.1109/CCGrid51090.2021.00030>
- Reza Hoseinyfarahabady, M., Zomaya, A. Y., & Tari, Z. (2018). A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *IEEE Transactions on Parallel and Distributed Systems*, 29(7), 1442–1455. <https://doi.org/10.1109/TPDS.2017.2779502>
- Sabyrkulov, E. (2021). *What are the Four Cloud Computing Service Delivery Models*. <https://maddevs.io/blog/an-overview-of-cloud-delivery-models/>
- Satyabrata, J. (n.d.). *Architecture of Cloud Computing - GeeksforGeeks*. Retrieved December 20, 2022, from <https://www.geeksforgeeks.org/architecture-of-cloud-computing/>
- Sen, J. (2013). Security and privacy issues in cloud computing. *Architectures and Protocols for Secure Information Technology Infrastructures*, iv, 1–45. <https://doi.org/10.4018/978-1-4666-4514-1.ch001>
- Shafiei, H., Khonsari, A., & Mousavi, P. (2022). Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys*, 1(1), 1–27. <https://doi.org/10.1145/3510611>
- Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., & Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020*, 205–218.
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th editi). <https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913>

- Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., & Akella, A. (2021). Atoll: A scalable low-latency serverless platform. *SoCC 2021 - Proceedings of the 2021 ACM Symposium on Cloud Computing*, 138–152. <https://doi.org/10.1145/3472883.3486981>
- Singhvi, A., Houck, K., Balasubramanian, A., Shaikh, M. D., Venkataraman, S., & Akella, A. (2019). *Archipelago: A Scalable Low-Latency Serverless Platform*.
- Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., & Tumanov, A. (2020). Cloudburst: Stateful functions as a service. *Proceedings of the VLDB Endowment*, 13(11), 2438–2452. <https://doi.org/10.14778/3407790.3407836>
- Stein, M. (2018). *The Serverless Scheduling Problem and NOAH*.
- Suresh, A., & Gandhi, A. (2019). FNSched: An efficient scheduler for serverless functions. *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*, 19–24. <https://doi.org/10.1145/3366623.3368136>
- Suresh, A., Somashekar, G., Varadarajan, A., Kakarla, V. R., Upadhyay, H., & Gandhi, A. (2020). ENSURE: Efficient scheduling and autonomous resource management in serverless environments. *Proceedings - 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020*, 1–10. <https://doi.org/10.1109/ACSOS49614.2020.00020>
- Tariq, A., Pahl, A., & Rozner, E. (n.d.). *Sequoia : Enabling Quality-of-Service in Serverless Computing*. 311–327.
- Theory, C., & Rountree, D. (2013). *The Basics of Cloud Computing : Understanding the Fundamentals of Cloud*.
- Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, 11(2), 233–247. <https://doi.org/10.1007/S11761-017-0208-Y/FIGURES/13>
- What Is Cloud Computing? Complex Guide for 2022*. (n.d.). Retrieved December 19, 2022, from <https://www.netguru.com/glossary/cloud-computing#deployment-models>

Yu, H., Irissappane, A. A., Wang, H., & Lloyd, W. J. (2021). FaaSRank: Learning to Schedule Functions in Serverless Platforms. *Proceedings - 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021*, 31–40. <https://doi.org/10.1109/ACSOS52086.2021.00023>

Appendix A List of sample codes

A. Creating Function

```
#!/usr/bin/env bash
```

```
wsk -i property set --auth $2  
wsk -i property set --namespace $1
```

```
wsk -i action delete prime-number  
wsk -i action create prime-number ../functions/js/prime-number/prime-  
number.js --web true -t 300000
```

```
wsk -i action delete matrix  
wsk -i action create matrix ../functions/js/matrix/matrix.js --web true -t  
300000 -m 128
```

```
wsk -i action delete db-get  
wsk -i action create db-get --kind nodejs:6 ../functions/js/db-  
get/action.zip --web true -t 300000
```

```
wsk -i action delete payload-size  
wsk -i action create payload-size ../functions/js/payload-size/payload-  
size.js --web true -t 300000
```

```
wsk -i action delete prime-number-java  
wsk -i action create prime-number-java ../functions/java/prime-number.jar --  
main PrimeNumber --web true -t 300000
```

```
wsk -i action delete matrix-java  
wsk -i action create matrix-java ../functions/java/matrix.jar --main Matrix  
--web true -t 300000 -m 128
```

B. Matrix function

```
package analysis.functions;  
import java.util.ArrayList;  
import java.util.List;
```

```
public class Matrix {  
    private static long[][] multiplyMatrices(long[][] m1, long[][] m2, int  
size) {  
        long[][] result = new long[size][size];  
        for (int i = 0; i < size; i++) {  
            for (int j = 0; j < size; j++) {  
                int sum = 0;  
                for (int k = 0; k < size; k++) {  
                    sum += m1[i][k] * m2[k][j];  
                }  
                result[i][j] = sum;  
            }  
        }  
        return result;  
    }  
    private static long[][] generateMatrix(int size) {  
        long[][] matrix = new long[size][size];  
        for (int i = 0; i < size; i++) {  
            for (int j = 0; j < size; j++) {
```

```

        long number = Math.round(117 + Math.random() * 40);
        matrix[i][j] = number;
    }
}
return matrix;
}
public static String multiplyRandomMatrices(int size, long ramLoader) {
    long[][] m1 = generateMatrix(size);
    long[][] m2 = generateMatrix(size);
    List<Object> fakeList = new ArrayList<>();
    for (int k = 0; k < ramLoader; k++) {
        fakeList.add(m1[k % 9].clone());
    }
    multiplyMatrices(m1, m2, size);

    return "Completed";
}
}

```

C. Prime number Function

```

function getNthPrimeNumber(n){
    var count = 0;
    var num = 2;
    while(count++ != n){
        num = getNextPrimeNumber(num);
    }
    return num;
}
function getNextPrimeNumber(n){
    for(var i = ++n; i < n*n; i++){
        if(isPrime(i)) return i
    }
    return 0;
}
function isPrime(n){
    for(var i = 2; i < n; i++)
        if (n%i===0)
            return false;
    return true;
}
function main(params) {
    var num = params.num;
    if (!num) {
        return {
            error: 'The input number must be specified'
        };
    }
    var result = getNthPrimeNumber(num);
    return {
        result: result
    };
}

```

D. ETLAS Scheduler

```
import random
import time
class Event:
    # Every event triggers a function invocation
    origin_zone = -1 # ID of the zone or region where the event is
    originated (zone 0 is the cloud; zones 1, 2, 3 are edge zones)
    assigned_zone = -1 # ID of the zone or region assigned to execute the
    event (zone 0 is the cloud; zones 1, 2, 3 are edge zones)
    init_time = 0 # Time when the request arrives to the serverless platform
    end_time = 0 # Time when the execution of the request ends in the
    serverless platform
    total_time = 0 # Total response time for the serverless platform point
    of view: total_time = end_time - init_time
    rtt = 0 # RTT observed for this request
    total_time_with_rtt = 0 # total_time + rtt --> response time for the
    client point of view
    resource_usage = []
class Resource:
    # A resource is a serverless instance used to execute one or more
    invocations of a function
    allocated_init_time = 0 # Time when the resource is allocated to execute
    an event (function invocation). The resource can be cold or warm
    allocated_end_time = 0 # Time when the resource ends the execution of
    the event (function invocation) and it is freed.
    idle_init_time = 0 # time when a warm resource start to be idle after
    booting or after ending the execution of a function invocation
    warmingup_ready = 0 # Time when a warming-up resource is ready to be
    used
    keep_warm = 0 # If keep_warm==1 it is a pre-warmed resource (should
    never be cold-down)
def create_event_list():
    file = open('event_list_sample.txt', 'w')
    interval = 1.0 / float(events_per_second)
    zone = 0
    for t in range(sim_time):
        for e in range(events_per_second):
            time = float(t) + float(e) * interval
            # print(time, zone)
            file.write("%.15f %i\n" % (time, zone))
            zone += 1
            if zone >= num_zones:
                zone = 0
    file.close()
def read_event_list():
    # This functions reads the file event_list.txt
    end_time=0
    # First we create an empty event list
    for i in range(max_sim_time):
        event_list.append([])
    file = open('event_list.txt', 'r')
    for line in file:
        data = []
        # We introduce in "data" the values of every line of the file
        # data[0] contains the time and data[1] contains the zone
        data = list(line.split(" "))
```

```

# Create a new event
event = Event()
event.init_time = float(data[0])
second = int(event.init_time)
if event.init_time > float(max_sim_time):
    print("ERROR!! Max. Simulation time exceeded")
    exit(-1)
    break
if event.init_time > end_time:
    end_time = event.init_time+1
event.origin_zone = int(data[1])
second = int(event.init_time)
event_list[second].append(event)
file.close()
return(end_time)
def initialize_resource_list():
    resource_id = 0
    for z in range(num_zones+1):
        idle_resources.append([])
        allocated_resources.append([])
        warmingup_resources.append([])
        resource_usage_time.append(0)
        resource_allocated_time.append(0) # Allocated time includes the
resource startup (cold or warm) + busy time (execution)
        resource_busy_time.append(0)
        resource_idle_time.append(0)
        number_of_cold_starts.append(0)
        if z < num_zones: # warm resources in edge zones
            prewarming = min(edge_prewarming, max_edge_resources)
        else:
            prewarming = cloud_prewarming
        for r in range(prewarming):
            resource = Resource()
            resource.boot_time = 0
            resource.idle_init_time = 0
            resource.keep_warm = 1
            idle_resources[z].append(resource)
            resource_usage_num_avg.append([])
            resource_usage_num_max.append([])
            for t in range(sim_time):
                resource_usage_num_avg[z].append(0)
                resource_usage_num_max[z].append(0)
def sched_policy_0(event):
    zone = event.origin_zone
    # First, we check if there are enough free resources in the local zone
    if len(allocated_resources[zone]) < max_edge_resources: # Schedule in
the local edge zone
        assigned_zone = zone
    else: # Schedule in the cloud zone
        assigned_zone = cloud_zone
    return assigned_zone
def sched_policy_1(event):
    zone = event.origin_zone
    # First, we check if there are enough free resources in the local zone
    assigned = 0
    if len(allocated_resources[zone]) < max_edge_resources: # There are free
resources in the local zone

```

```

        if len(idle_resources[zone]) > 1: # There are warm resources in the
local edge zone
            assigned_zone = zone
        elif len(idle_resources[num_zones]) > 1: # There are warm resources
in the cloud zone
            assigned_zone = cloud_zone
        else: # No warm resources available ==> then boot a cold resource
from local zone
            assigned_zone = zone
    else: # No free resources in local zone ==> Schedule in the cloud zone
        assigned_zone = cloud_zone
    return assigned_zone
def allocate_resource(event):
    global sim_end_time

    origin_zone = event.origin_zone
    assigned_zone = event.assigned_zone
    init_time = event.init_time
    cold_start = cold_start_time * (1 + random.uniform(-1, 1) *
cold_start_variance / 100)
    warm_start = warm_start_time * (1 + random.uniform(-1, 1) *
warm_start_variance / 100)
    exec = exec_time * (1 + random.uniform(-1, 1) * exec_time_variance /
100)

    if origin_zone == assigned_zone:
        rtt_latency = (edge_rtt_latency * (1 + random.uniform(-1, 1) *
edge_rtt_variance / 100))/1000
    else: # Schedule in the cloud zone
        rtt_latency = (cloud_rtt_latency * (1 + random.uniform(-1, 1) *
cloud_rtt_variance / 100))/1000

    # Look for an Idle (warm) resource in the assigned zone.
    if len(idle_resources[assigned_zone]) > 0:
        resource = idle_resources[assigned_zone].pop(0)
        end_time = init_time + float(exec + warm_start) / 1000.0
    # If no idle (warm) resource found, we first look for any warming-up
resource in the assigned zone.
    elif len(warmingup_resources[assigned_zone]) > 0:
        resource = warmingup_resources[assigned_zone].pop(0)
        end_time = max(resource.warmingup_ready, init_time) + float(exec +
warm_start) / 1000.0
    # If no idle nor warming-up resource found, the we start a new (cold)
resource in the assigned zone.
    else:
        resource = Resource()
        end_time = init_time + float(exec + cold_start) / 1000.0
        resource.boot_time = init_time
        resource.keep_warm = 0 # It is not a pre-warmed resource
        number_of_cold_starts[assigned_zone] =
number_of_cold_starts[assigned_zone] + 1

    resource.allocated_init_time = init_time
    resource.allocated_end_time = end_time
    allocated_resources[assigned_zone].append(resource)
    resource_allocated_time[assigned_zone] =
resource_allocated_time[assigned_zone] + end_time - init_time

```

```

    resource_busy_time[assigned_zone] = resource_busy_time[assigned_zone] +
float(exec)/1000.0
    event.end_time = end_time
    event.total_time = end_time - init_time
    event.rtt = rtt_latency
    event.total_time_with_rtt = event.total_time + event.rtt

    if sim_end_time < end_time:
        sim_end_time = end_time
    return event
def update_resource_list(time):
# This function checks:
# - if some allocated resource should pass to idle (warm)
# - if any booting (warming up) resources should pass to idle (warm)
# - if any idle resources should be shut-down (terminated)
    for z in range(num_zones+1):
        # Check if any allocated resources should pass to idle (warm)
        i = 0
        while i < len(allocated_resources[z]):
            if time > (allocated_resources[z][i].allocated_end_time +
reuse_interval/1000):
                resource = allocated_resources[z].pop(i) # the resource pass
to idle list
                resource.idle_init_time = resource.allocated_end_time +
reuse_interval/1000
                idle_resources[z].append(resource)
            else:
                i = i + 1
        # Check if any booting (warming up) resources should pass to idle
        i = 0
        while i < len(warmingup_resources[z]):
            if time > warmingup_resources[z][i].warmingup_ready:
                resource = warmingup_resources[z].pop(i) # the resource
pass to idle list
                resource.idle_init_time = resource.warmingup_ready
                idle_resources[z].append(resource)
            else:
                i = i + 1
        # Check if any idle resources should be shut-down (terminated)
according to pre-warming policies
        # Pre-warming policy 0 --> Fixed pre-warming ==> Pre-warmed
resources (with keep_warm==1) are never cold-down
        if prewarming_policy == 0:
            i = 0
            while i < len(idle_resources[z]):
                cold_down_time = idle_resources[z][i].idle_init_time +
float(keep_alive_interval) / 1000.0
                if (idle_resources[z][i].keep_warm == 0) and (time >
cold_down_time):
                    resource = idle_resources[z].pop(i) # the resource pass
to cold state and it is removed from any list
                    resource_usage_time[z] = resource_usage_time[z] +
(cold_down_time - resource.boot_time)
                else:
                    i = i + 1

```

```

        # Pre-warming policy 1 --> Adaptive pre-warming ==> pre-warm
resources on every zone to reach prewarming value
        if prewarming_policy == 1:
            if z < num_zones:
                prewarming = min(edge_prewarming, max_edge_resources)
            else:
                prewarming = cloud_prewarming
                diff = prewarming - (len(idle_resources[z]) +
len(warmingup_resources[z]))
                if diff < 0: # Check if some idle resource can be shut-down
(terminated)
                    max_cold_down = - diff # no more than max_cold_down
resources should be cold down to guarantee prewarming
                    cold_down = 0
                    i = 0
                    while i < len(idle_resources[z]):
                        cold_down_time = idle_resources[z][i].idle_init_time +
float(keep_alive_interval) / 1000.0
                        if (idle_resources[z][i].keep_warm == 0) and (time >
cold_down_time):
                            resource = idle_resources[z].pop(i) # the resource
pass to cold state and it is remove from any list
                            resource_usage_time[z] = resource_usage_time[z] +
(cold_down_time - resource.boot_time)
                            # resource_idle_time[z] = resource_idle_time[z] +
(time - resource.idle_init_time)
                            cold_down = cold_down + 1
                        else:
                            i = i + 1
                    if cold_down >= max_cold_down:
                        break
                if diff > 0: # if diff > 0 ==> No enough warm resources, then
warm up "diff" new resources
                    for i in range(diff):
                        resource = Resource()
                        resource.boot_time = time
                        resource.keep_warm = 0
                        resource.warmingup_ready = time + prewarming_time * (1 +
random.uniform(-1, 1) * cold_start_variance / 100) / 1000.0
                        warmingup_resources[z].append(resource)
def process_event(time, event_num):
    # This function is responsible for processing each event
    event=event_list[time][event_num]
    if sched_policy == 1:
        # Policy 1 --> schedule first where there are warm resources (cloud
or edge), if not, in the local zone
        #assigned_zone, end_time = sched_policy_1(origin_zone, init_time)
        event.assigned_zone = sched_policy_1(event)
    else:
        # Policy 0 --> schedule first in the local zone
        #assigned_zone, end_time = sched_policy_0(origin_zone, init_time)
        event.assigned_zone = sched_policy_0(event)
    update_resource_list(event.init_time)
    event = allocate_resource(event)
    event_list[time][event_num] = event
    # Compute how many resources are being used when this event is processed

```

```

    # then add this number to the total number of resources used in this
    zone, to compute, later, the average
    for z in range(num_zones+1):
        num_used_resources = len(warmingup_resources[z]) +
len(idle_resources[z]) + len(allocated_resources[z])
        resource_usage_num_avg[z][time] = resource_usage_num_avg[z][time] +
num_used_resources
        resource_usage_num_max[z][time] = num_used_resources
def compute_resource_usage():
# This function is responsible for computing and print the resource usage
time
    # Compute first resource usage time per zone
    total_usage_time = 0
    total_allocated_time = 0
    total_idle_time = 0
    total_busy_time = 0
    total_number_of_cold_starts = 0
    # Update the resource usage with the resources that remain idle or
warming-up at the end of the simulation
    for z in range(num_zones+1):
        for i in range(len(idle_resources[z])):
            resource_usage_time[z] = resource_usage_time[z] + (sim_end_time
- idle_resources[z][i].boot_time)
        for i in range(len(warmingup_resources[z])):
            resource_usage_time[z] = resource_usage_time[z] + (sim_end_time
- warmingup_resources[z][i].boot_time)
        # print("Resource usage time for zone", z, "-->",
resource_usage_time[z], "s")
        resource_usage_time[z] = float(resource_usage_time[z])/3600.0 #
Pass seconds to hours
        resource_allocated_time[z]
= float(resource_allocated_time[z])/3600.0 # Pass seconds to hours
        resource_busy_time[z] = float(resource_busy_time[z])/3600.0 # Pass
seconds to hours
        resource_idle_time[z] = resource_usage_time[z] -
resource_allocated_time[z]
        total_usage_time = total_usage_time + resource_usage_time[z] # in
hours
        total_allocated_time = total_allocated_time +
resource_allocated_time[z] # in hours
        total_idle_time = total_idle_time + resource_idle_time[z] # in
hours
        total_busy_time = total_busy_time + resource_busy_time[z] # in
hours
        total_startup_time = total_allocated_time - total_busy_time # in
hours
        total_number_of_cold_starts = total_number_of_cold_starts +
number_of_cold_starts[z]
    for z in range(num_zones + 1):
        resource_usage_time[z] = round(resource_usage_time[z], 2)
        resource_allocated_time[z] = round(resource_allocated_time[z], 2)
        resource_idle_time[z] = round(resource_idle_time[z], 2)
    print("\n")
    print("Total Resource usage time per zone:")
    print(resource_usage_time, "hours") # in hours
    print("Total resource allocated time per zone (includes busy + start-up
times):")

```

```

print(resource_allocated_time, "hours") # in hours
print("Total resource idle time per zone:")
print(resource_idle_time, "hours") # in hours
print("\n")
print("Total resource usage time -->",
"{:.2f}".format(total_usage_time), "hours")
print("Total resource startup time -->",
"{:.2f}".format(total_startup_time), "hours")
print("Total resource busy time -->", "{:.2f}".format(total_busy_time),
"hours")
print("Total resource idle time -->", "{:.2f}".format(total_idle_time),
"hours")
print("\nTotal number of cold starts -->", total_number_of_cold_starts)
# Compute then number of resources used, in average, every second, per
zone
average_per_zone = []
total = 0
max = 0
for z in range(num_zones+1):
    average_per_zone.append(0)
    for t in range(sim_time):
        if len(event_list[t]) > 0:
            resource_usage_num_avg[z][t] = resource_usage_num_avg[z][t]
/ len(event_list[t])
        else:
            resource_usage_num_avg[z][t] = 0
            average_per_zone[z] = average_per_zone[z] +
resource_usage_num_avg[z][t]
            if resource_usage_num_max[z][t] > max:
                max = resource_usage_num_max[z][t]
            average_per_zone[z] = average_per_zone[z] / sim_time
            total = total + average_per_zone[z]
    print("Average number of resources used per second -->",
"{:.2f}".format(total))
    print("Max number of simultaneous resources used -->", max)
def simulation():
# This functions process one-by-one all the events in the event list
    total_time = 0.0 # Sum of response times for all tasks from the
serverless platform point of view
    client_total_time = 0.0 # Sum of response times for all tasks from the
client point of view
    min_time = 99999.0 # Task with minimum response time
    max_time = 0.0 # Task with maximum response time
    min_client_time = 99999.0 # Task with minimum response time
    max_client_time = 0.0 # Task with maximum response time
    total_tasks = 0
    for t in range(sim_time):
        for event_num in range(len(event_list[t])):
            process_event(t, event_num)
            total_tasks = total_tasks + 1
            task_time = event_list[t][event_num].total_time
            client_task_time = event_list[t][event_num].total_time_with_rtt
            total_time = total_time + task_time
            client_total_time = client_total_time + client_task_time
            if task_time < min_time:
                min_time = task_time
            if task_time > max_time:

```

```

        max_time = task_time
    if client_task_time < min_client_time:
        min_client_time = client_task_time
    if client_task_time > max_client_time:
        max_client_time = client_task_time
    minutes = int(t / 60)
    seconds = t - (minutes * 60)
    if seconds == 59:
        print(" -- simulated", minutes + 1, "min.")
    average_time = 1000.0 * total_time / float(total_tasks)
    min_time = 1000.0 * min_time
    max_time = 1000.0 * max_time
    average_client_time = 1000.0 * client_total_time / float(total_tasks)
    min_client_time = 1000.0 * min_client_time
    max_client_time = 1000.0 * max_client_time
    update_resource_list(sim_end_time)
    compute_resource_usage()
    print("\nSimulation end time:", sim_end_time)
    print("\nServerless platform response time:")
    print(" - Average task response time --> ",
"{:.2f}".format(average_time), "ms")
    print(" - Min. task response time --> ", "{:.2f}".format(min_time),
"ms")
    print(" - Max. task response time --> ", "{:.2f}".format(max_time),
"ms")
    print("\nClient-side response time:")
    print(" - Average task response time --> ",
"{:.2f}".format(average_client_time), "ms")
    print(" - Min. task response time --> ",
"{:.2f}".format(min_client_time), "ms")
    print(" - Max. task response time --> ",
"{:.2f}".format(max_client_time), "ms")
def store_output():
    file = open('event_output.txt', 'w')
    file2 = open('response_time.txt', 'w')
    n = 0
    for t in range(sim_time):
        for event in event_list[t]:
            file.write("Event %i --> Init: %.2f s. - End: %.2f - Total %.1f
ms (For client --> Total %.1f ms)\n" % (n, event.init_time, event.end_time,
1000*event.total_time, 1000*event.total_time_with_rtt))
            file2.write("%.2f\n" % (1000*event.total_time))
            n += 1
    file.close()
    file2.close()
# SIMULATION TIME
max_sim_time = 24 * 60 * 60 # Max sim time = 1 day
sim_end_time = 0 # Time when the last event finish its execution

#INPUT DATA
create_input_data = 0 # If 0 --> read default input data file
(event_list.txt) # If 1 --> create a new input data
file (event_list.txt) according to the next parameters: sim_time and
events_per_second
sim_time = 10 # If create_input_data==1, use this sim_time (in
seconds)

```

```

events_per_second = 100 # Create an input event_list.txt file with this
number of events_per_second (uniformly distributed) and sim_time duration
num_zones = 5
cloud_zone = num_zones
sched_policy = 1
prewarming_policy = 0

# Num. of resources prewarmed:
edge_prewarming = 0 # resources prewarmed per edge zone
cloud_prewarming = 160 # resources prewarmed in the central cloud
# Keep_alive period (for instance reuse mechanism)
keep_alive_interval = 30000 # If a warm resource is unused for this period
(in ms), the resource is cold down
reuse_interval = 0 # When a busy resource passes to idle, it can be reused
after this interval
# Resource limit
max_edge_resources = 1000000 # Max. number of resources available at each
edge zone
# DELAYS
exec_time = 2053 # Execution delay in ms (guessed value, it depends on the
application)
cold_start_time = 6090 # Average cold start delay in ms
warm_start_time = 71 # Average warm start delay in ms
prewarming_time = cold_start_time - warm_start_time # Time needed for
booting pre-warmed instances (we assume that it is equal to cold_start -
warm_start)
exec_time_variance = 1 # Variance (%) applied to cold start
cold_start_variance = 1 # Variance (%) applied to cold start
warm_start_variance = 4 # Variance (%) applied to warm start
edge_rtt_latency = 62 # RTT latencies (ms) between users and resources from
local edge zone
edge_rtt_variance = 15 # Variance (%) applied to edge RTT latencies
cloud_rtt_latency = 851 # RTT latencies (ms) between users and resources
from central cloud
cloud_rtt_variance = 5 # Variance (%) applied to cloud RTT latencies
# OTHER GLOBAL VARIABLES
# Cloud is assumed to be the last zone
# (e.g. for 4 zones: zones #0, #1, #2, #3 are the edge zones; and zone #4
is the cloud)
event_list = []
resource_list = []
idle_resources = [] # List of idle (warm) resources. One list per zone z.
E.g.: idle_resources[z][r]
allocated_resources = [] # List of allocated resources. One list per zone z.
E.g.: allocated_resources[z][r]
warmingup_resources = [] # List of warming up resources. One list per zone
z. E.g.: warmingup_resources[z][r]
resource_usage_time = [] # Total resource usage time per zone (idle + busy +
startup states)
resource_allocated_time = [] # Total resource allocated time per zone
(includes busy and startup times)
resource_busy_time = [] # Total resource busy time per zone (include only
execution time)
resource_idle_time = [] # Total idle (warm) resource time per zone (only
idle state)
resource_usage_num_avg = [] # Average number of resources used every second
per zone (warm + allocated + warmingup)

```

```

resource_usage_num_max = [] # Max number of resources used every second per
zone (warm + allocated + warmingup)
number_of_cold_starts = [] # Number of cold starts per zone
# START THE MAIN CODE
print("Reading input files .....")
now = time.time()
sim_time = int(read_event_list())
initialize_resource_list()
elapsed_time = time.time() - now
print("Done (", "{:.2f}".format(elapsed_time), "s.)")
print("Starting Simulation .....")
now = time.time()
# Perform the simulation
simulation()
elapsed_time = time.time() - now
print("(Simulator run time:", "{:.2f}".format(elapsed_time), "s.)")
store_output()

```