M.Sc. Engg. (CSE) Thesis

A Novel Architecture for Mitigating Cold Start Problem in Serverless Computing

Submitted by

Khondokar Solaiman

1017052017

Supervised by

Dr. Muhammad Abdullah Adnan

Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

January 2023

Dedicated to my parents

Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "A Novel Architecture for Mitigating Cold Start Problem in Serverless Computing", is the outcome of the investigation and research carried out by me under the supervision of Dr. Muhammad Abdullah Adnan, Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

[illegible]

Khondokar Solaiman
1017052017

The thesis titled "**A Novel Architecture for Mitigating Cold Start Problem in Serverless Computing**", submitted by Khondokar Solaiman, Student ID 1017052017, Session October 2017, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on January 24, 2023.

Board of Examiners

Acknowledgement

I am thankful to my thesis supervisor Dr. Muhammad Abdullah Adnan for his exceptional guidance and persistent pursue of excellence to push me in right direction. Moreover, I want to mention my thesis board members specially Dr. Mahmuda Naznin and Dr. Rifat Shahriyar and many reviewers for their valuable advises and suggestions to ameliorate my thesis.

Finally, I am grateful to Allah and would like to express my heartfelt gratitude to everyone to make this thesis possible even in this unprecedented COVID pandemic times.

Dhaka	Khondokar Solaiman
January 24, 2023	1017052017

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Features like no server maintenance, low cost, and elasticity are attracting developers to build applications utilizing the functions-as-a-service (FaaS) architecture on serverless platforms. Researchers have proposed various fixes to overcome the cold start problem, a well-known issue of serverless architecture. In this thesis, we address the cold start problem of the serverless platform. Our study shows that we can not get rid of cold start time because of the on-demand nature of the FaaS model. In addition, developers are adopting FaaS for critical stateful applications to retain the architecture's benefits.

Consequently, the need for an adaptive and intelligent fault-tolerant technique for FaaS architecture has transpired. It has become evident that the classic Retry technique is neither reliable nor efficient for handling such critical applications. Admittedly, mitigating cold start latency and achieving fault tolerance for serverless has remained an open problem. Therefore, we propose Warm Lambda Equipped Container (WLEC), a container management architecture that utilizes a modified Two Segmented Least Recently Used (S2LRU) structure called S2LRU++ inspired by the S2LRU* architecture of Pannier. Then, we evaluate WLEC in both AWS and Local VM environments with OpenLambda with six different metrics. Assessing the practical implication and performance, we realize the shortcomings of WLEC and decide to alleviate them. Subsequently, we have developed Advance Warm Lambda Equipped Container (AdWaLEC) that addresses cold start latency and fault tolerance with replication-based strategies.

Moreover, AdWaLEC redesigns WLEC-one of the cold start mitigation solutions, by improving container replication of template container list utilizing Spearman's rank correlation coefficient to reduce cold start delay. Next, it also incorporates a reliable fault tolerance scheme for the ubiquitous serverless model called Warm Lambda Container Replication (WaLCoR) that leverages container-associated logs. Further, we have implemented AdWaLEC in the OpenLambda platform, evaluated it with Azure Function trace with seven different metrics, and compared it with four different approaches. For instance, AdWaLEC shows a 77.23% reduction in cold start frequency, a 27.36% decrease in cold start latency, and a 29.95% increase in invocation per container compared to the baseline of OpenLambda.

Chapter 1

Introduction

The advancement of cloud computing now makes it possible to define the new concept of serverless computing. It often refers to developers running applications without managing any server dependencies. They do not require any provisioning scheme or management of servers. Serverless computing has become one of the most popular emerging technology for modern cloud-based systems. The serverless architecture lets us focus on the core product and business logic instead of responsibilities like operating system (OS) access control, patching, provisioning, right-sizing, scaling, and availability. Different versions of the service models, like Infrastructure as a service (IaaS), Software as a service (SaaS), and Platform as a service (PaaS), are provided by the cloud providers. Serverless is in the fourth category: Functions as a service (FaaS) [26], based on runtime-level virtualization. In the FaaS model, the developers need not think about the runtime that the code uses. They need to write code based on the platform and use a way that the platform supports and uses the library provided by the platform. Such attractive features have caught developers' attention and are gaining popularity in application development and deployment. FaaS service providers include Amazon Lambda [2], Google Cloud Functions [3], Azure Functions [5], OpenLambda [51], Apache OpenWhisk [1], OpenFaaS [6] etc.

1.1 Inception of Serverless

In earlier settings, applications needed their environment, including hardware and software. The introduction of virtualization has made it possible to share the same hardware for multiple operating systems known as Virtual Machine (VM) [17]. Later container-based virtualization allowed us to use the same OS and H/W for different applications. The repackaging of Unix-style processes combined with distribution tools has gained huge favor among developers, known as Docker [41]. However, the Lambda model is the latest virtualization model that allows sharing of runtime resources. It has revolutionized the concept of Serverless Computing. With the Lambda model, applications are seen as a set of functions called lambda functions instead of a collection of servers and developers. Different lambda handlers are written to handle different kinds of requests. As we browse available lambda services, we find that lack of complete control over the lambda model hinders the implementation of the lambda model in real life. All the core sections: request distribution, code files, and code executions- are hidden from the developer because of security, consistency, resource management, and performance issues [113]. The lack of control in code execution on specific containers results in increased start-up time, also known as the cold start problem.

1.1.1 Cold Start Problem in Serverless

Specifically, the cold start mainly refers to the time delay from receiving a lambda request in the application to the start of executing code for that request. Alternatively, the time of preparing a container to serve any request is called cold start. As a result, the absence of any built-in mechanisms to counter cold start leads to the same problem for all lambda models in the industry. Thus, the only possible way out from the developers' end is to send continuous hello request actively to keep those lambda containers warm.

Serverless service providers have been working on the issue since the inception of the serverless concept. However, no generalized approach has been attained yet. Nonetheless, platform-specific solutions are surfacing from time to time. AWS-one of the major serverless service provider-has recently proposed SnapStart [100] which shows promises to finally solve the ever-persistent issue. In addition, no extra cost for SnapStart also makes it attractive to adopt for serverless applications. On the other hand, since it is particularly crafted keeping in mind the lambda architecture, the effectiveness of SnapStart for other service providers remains a question for the future. Though SnapStart service may be expanded for other functions, it should be noted that it is only available for Java 11 functions till now (as of writing this thesis). Therefore, its impact and scope are heavily limited to a small subset of applications right now. Similarly, the lack of proper mechanisms for handling fault tolerance, uniqueness, randomness, dynamic network connections, and ephemeral data makes a strong case for further study in this field to find better solutions. Admittedly, SnapStart provides a strong baseline for future research for other similar service providers such as OpenWhisk, Google Cloud, and Azure functions.

Furthermore, researchers have proposed various methods to reduce cold start delay in [89], [29], [73], [102], [45]. Similar replication-based approaches of recent times include time series forecasting model [56], retention-aware container caching [91], managing pool of function instances model [75], introducing new scheduler [116], function start-up reduction [104], prediction based warm up strategy [132]. However, previous works ignore disparate and unpredictable request patterns, adaptivity, and platform abstraction and suffer from high memory costs. Additionally, none of the aforementioned methods intends to facilitate fault tolerance along with the cold start mitigation technique. Thus, cold start remains a challenging problem even now for FaaS service providers. In our thesis, we aim to develop a platform independent cold start mitigation scheme that

1.2 Formulation of WLEC

In addition, after the introduction of OpenLambda-an open-source lambda model implementation, developers and researchers now have more control over the architecture of the lambda model than ever before [51]. Here, we propose WLEC to counter the cold start problem for the conventional lambda model and implement it in the OpenLambda Platform. Pannier [69], a container-based caching policy, encourages the architectural idea for WLEC.

1.2.1 Key Features

WLEC uses a modified version of the Two Segmented Least Recently Used (S2LRU++) instead of the standard two-queue S2LRU model. The specialty of S2LRU++ is that it is a container-ware S2LRU model and uses three queues rather than two. The three queues are cold, warm, and the template queue in S2LRU++. These queues hold containers that run the lambda functions. Containers are placed in these queues based on their usage, invocation time, wake-up time, state, and other parameters and are managed by the CMS (Container Management Service). CMS continuously monitors the state of the containers and can move, destroy and initialize them accordingly. CMS consists of three main functions: *Initialization* initiates a container with corresponding flags and variables, *QueuePlacing* handles the transition of a container from one queue to another, and *OnRequest* handles the request selecting the best container to serve the request at any given time.

1.2.2 Major Findings

We test the performance of WLEC architecture from our comparison of six different metrics with the ubiquitous lambda model. We define these metrics and compute them for the OpenLambda platform in the Local VM and AWS VM setup. By further computation of these metrics, we show that WLEC reduces average cold start invocation cases by 31% and the average duration of cold start by 23.5% in AWS VM. For memory consumption, we find WLEC architecture consumes about half of the memory of traditional All-Warm methods. Besides, we observe about a 31.25% increase in invocations per container while a 70.2% reduction in start-up latency compared to the fresh start scenario for AWS VM. We also find that Advance WLEC reduces the number of cold start invocation cases by 12.68% and the average duration of cold start invocation by 25.09% in AWS VM compared to the OpenLambda baseline. In the case of memory consumption, we find Advance WLEC architecture consumes about 18.04% less memory than the traditional

All-Warm method. Besides, we find about an 18.19% increase for invocations per container while an 18.97% reduction for start-up latency compared to the fresh start scenario for AWS VM.

1.2.3 Design Flaws

However, the number of redundant containers of each type of function in WLEC was static and designed as one-time user-defined parameters. Hence, it considers that the user has prior knowledge regarding the requests' nature, like frequencies and response times. However, it may be possible that the system owner does not have any prior understanding of request frequencies, or it is hard to predict the request pattern. In such cases, optimization of WLEC can be impossible for naive system owners. Thus, it will be reasonable to introduce an intelligent method that will dynamically determine the replication number for each type of container in runtime. Besides, WLEC suffers from a single point of failure issue since the failure of WLEC will halt the system from serving any request.

1.3 Consideration of Fault Tolerance

Further, an essential aspect of any system is fault tolerance (FT). Nevertheless, the sole stateless nature of serverless architecture has made it difficult to implement traditional fault-tolerant models. However, now the popular trend and adoption of serverless in critical applications have created the scope of thinking fault tolerance techniques for serverless architecture. Furthermore, we find that researchers are devising new strategies for adopting stateless serverless functions for stateful transaction operations like Cloudburst [109], programming models [37], PORTALS [107], Pheromone [135], CRUCIAL [22], FAASM [103]. On top of that, Serverless architecture is being adopted into other diversified and trending fields like machine learning, edge computing, etc. To illustrate, Distributed Deep Neural Network (DDNN) training framework-λDNN [131], Graph Neural Network (GNN) framework-Dorylus [114] exhibit researchers' interest in serverless. Therefore, research works regarding fault tolerance are getting more attention than ever. For instance, Beldi [136], Boki [57], Threshold-Based Adaptive Fault Tolerance (TBAFT) [96], Atomic Fault-Tolerant (AFT) shim [108] are some of the recent works are to mention a few. Yet, none of the approaches consider a cold start in its formulation resulting in hundreds of milliseconds of latency.

1.4 Conceptualization of AdWaLEC

Thus, We introduce Advance WLEC (AdWaLEC)-an extension scheme of WLEC architecture that facilitates dynamic replica containers and determines their amount based on the need during runtime.

1.4.1 Principal Aspects

We propose an adaptive container replication model based on one of the most well-known and robust ranking methods named Spearman's Rank Correlation Coefficient. The reason behind choosing this ranking method is that it makes no assumption about the ranks and can deal with many outliers. As our system needs to be able to deal with any request patterns, it will be the most effective in this case. We provide an alternative pathway with the baseline OpenLambda model to solve the single point of failure problem of WLEC. We also introduce a fault tolerance mechanism for AdWaLEC named WaLCoR to provide fault-tolerant attributes. The WaLCoR combines the traditional *Record and play* and *Checkpointing* method. Additionally, to mitigate a single point of failure problem, we keep another alternative path in Advance WLEC to serve the request during any failure.

1.4.2 Scope of AdWaLEC

AdWaLEC utilizes replication and redundancy to mitigate cold start and achieve fault tolerance simultaneously. As our scheme employs redundancy and replication to achieve fault tolerance, it replaces containers instead of fixing faulty containers. WaLCoR-a part of AdWaLEC-only considers container-related transient faults. To clarify, other common faults, such as hardware, network, and external service-related faults, are out of the scope of our study. However, WaLCoR takes advantage of the template container log to achieve consistency with other containers residing in different queues.

1.4.3 Major Takeaways

We evaluate AdWaLEC using seven different metrics with the OpenLambda platform by comparing it with four previously proposed well-known methods: baseline, all warm, SOCK, and WLEC. We compute each metric with a random selection of Azure functions from [102]. We find that AdWaLEC reduces cold start frequency compared to baseline and all warm approaches by 77.23% and 42.21%, respectively. AdWaLEC also reduces new container invocation by 13.84% compared to SOCK. We also get higher container invocation per container than baseline, all warm and SOCK by 29.95%, 30.39%, and 10.88%, respectively. In terms of maximum start-up latency, our proposed scheme shows 67.68% less latency than SOCK. However, SOCK consumes 16.14% less memory than AdWaLEC. However, AdWaLEC outperforms all warm in the case of memory consumption by 26.49%. Recovery latency metrics indicate that AdWaLEC ensures container recovery regardless of memory availability. AdWaLEC shows 82.6% and 85.43% less recovery latency compared to SOCK and all warm approaches. To conclude, we implement AdWaLEC on a real-time application to determine its impact in a pragmatic scenario.

1.5 Main Contributions of the Thesis

The main contributions of this thesis are as follows:

- Firstly, we discuss various reasons, contributors, and impacts of cold start in the OpenLambda platform. Besides, we explore the traditional approaches and recent studies to minimize cold start time for containers and their suitability in the lambda model.
- In addition, we propose an integrated, structured approach to counter the cold start problem. We develop WLEC, an S2LRU++ based architecture to handle the containers in a more structured fashion to ensure less start-up latency and better concurrency than the ubiquitous lambda model.
- After that, we describe the design, components, workflow, and implementation of WLEC in the OpenLambda platform and evaluate it using Local VM and AWS VM with different metrics.
- Further, considering the limitation of WLEC, we develop a multifaceted scheme named AdWaLEC-an improved WLEC architecture that facilitates not only the cold start mitigation but also fault tolerance for FaaS architecture.
- Next, we introduce an adaptive container replication model using Spearman's Rank Correlation Coefficient and integrate it with WLEC to mitigate cold start latency.
- Then, we solve the single point of failure problem for WLEC.
- Later, we present a reliable fault tolerance technique for serverless platforms named WaLCoR, the first of its kind.
- Subsequently, we implement AdWaLEC in the OpenLambda platform and evaluate it using Azure Function trace with seven different metrics. We provide an extensive performance comparison of AdWaLEC with baseline, all warm, SOCK, and WLEC approach.
- Finally, we employ WLEC and AdWaLEC with a real-time image resizing application from the Amazon Web Service (AWS) lambda pool and show the impact in a practical scenario.

1.6 Objective of the Thesis

The main objective of this thesis can be divided into three main questions. Firstly, How to develop a scheme that can mitigate cold start latency significantly for ubiquitous serverless applications? Secondly, How to design a fault-tolerant serverless architecture for critical FaaS applications? Finally, How to formulate one combined solution that can answer the previous two questions ensuring reduced cold start latency and fault tolerance? In summary, our objective is to design a reliable and effective fault-tolerant scheme consolidated with an adaptive cold start mitigation strategy.

1.7 Major Challenges

Cold start mitigation has become a burning question for cloud researchers. Large and critical applications required dealing with chained requests and interlinked applications. Thus, a single request may require booting up multiple containers with functions and getting a response. Consequently, cold start latency may increase exponentially from a single request for clients. Dealing with such cases with a structured strategy will be a crucial design challenge. Similarly, conventional fault tolerance techniques like *Retry*, *Record and play*, *Checkpointing* are designed for stateful architectures. However, serverless applications are stateless by nature. Hence, developing a fault tolerance scheme for stateless applications would be a significant obstacle to overcome. Integrating both solutions into a single scheme can have an unknown effect on the overall architecture. To sum up, we must formulate a novel, robust, efficacious, and high-performance solution that can facilitate cold start mitigation and fault tolerance effectively and efficiently.

1.8 Thesis Organization

From here on, in Chapter 2, we discuss the motivations behind our work. In Chapter 6, we mention diverse and relevant previous research works that discuss the reduction of the cold start time, rank correlation, and fault tolerance for serverless platforms. Then we provide the necessary definitions and in-depth descriptions of most of the terms and components we used later in our work in Chapter 4. We describe Pannier, OpenLambda and its various components, Ranking correlation and correlation coefficients, Thundra, Cold Start, and the role of cache replacement policies in our work for container management. After that, in chapter 5, we describe WLEC in detail, along with all the components and CMS functions of WLEC. Later, we narrated Advance WLEC, its components, function, and fault tolerance in Chapter 5. Then we move on to Chapter 7, evaluating our schemes with the OpenLambda setup. Our performance comparison shows how WLEC can play a leading role in minimizing cold start time, whereas Advance

WLEC can be potent in providing fault tolerance for serverless architecture. Next, in Chapter 8, we look at some of the possible future works of our thesis, and finally, in Chapter 9, we conclude our thesis on a promising note.

Chapter 2

Background & Motivation

In this chapter, we discuss our motivations behind choosing cold start time minimization as our research topic. We analyze the current condition of the OpenLambda platform and determine the effects of the cold start time. We also explore established methods for handling cold start delays for serverless service providers.

2.1 Cloud Computing

The immense unpredictable demand for dynamic, real-time web service in the current web 2.0 era, along with the 4th industrial revolution, has paved the path for the rapid adoption of large-scale cloud computation services. The idea of virtualization and the need for on-demand dynamic resources for storage, computation, processing, and scaling pushed us to the cloud revolution in the IT field. Cloud computing can be considered an extension of the previous concept of "Grid Computing". However, the main focus of grid computing was grid middle-ware-based decentralized, distributed, parallel computer architecture. In contrast, cloud computing relies on virtualization, on-demand service, and automatic ultra-high scaling [127].

2.1.1 Definition

The term "Cloud" basically refers to the collection of computer resources that collectively provide millions of services to its users. In another definition, " A cloud is a pool of virtualized computer resources" [95]. The fundamental nature of interactive, middleware-free, and user-focused architecture design distinguishes a cloud from its antecedent grid architecture. So, Cloud computing is a computing model that utilizes these clouds(virtualized resources) and shares their information, software, and other resources with different devices over the internet to provide maximum computation power.

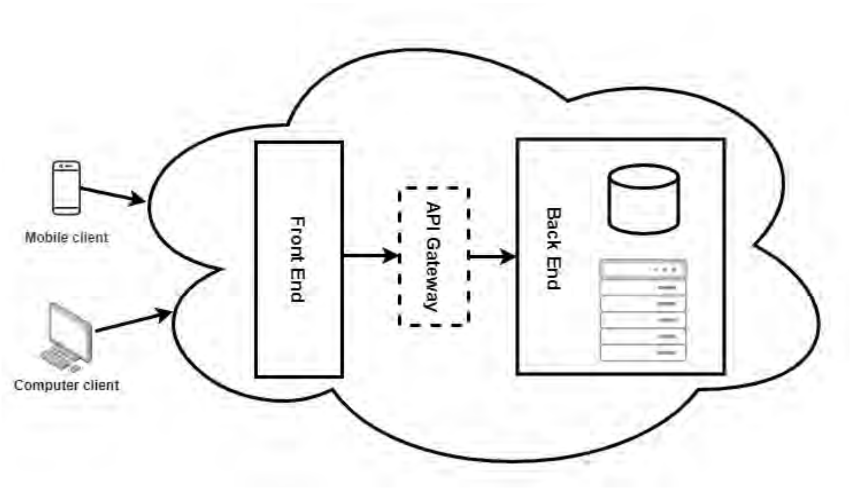

Figure 2.1: Cloud architecture

2.1.2 Basic Architecture

The basic architecture of cloud computing is shown in fig 2.1. A general cloud application has two main parts: *front end* and *back end*.

When a user wants a cloud service, he needs to go to the application home page, also known as *front end*. Users can access it via a mobile, desktop, or laptop client. From there, the user sends a request for the cloud service to the back end. The *front end* contains all the services available on the page. The request goes to the *back end* via an API designed by the service provider.

The second part, *back end*, contains all the resources and codes required to process the requested service and send a response to the *front end*. It contains resources like storage, database, codes for request processing, etc. The response is also sent using the same API that brought the request. All cloud applications follow this same architecture for customer service delivery.

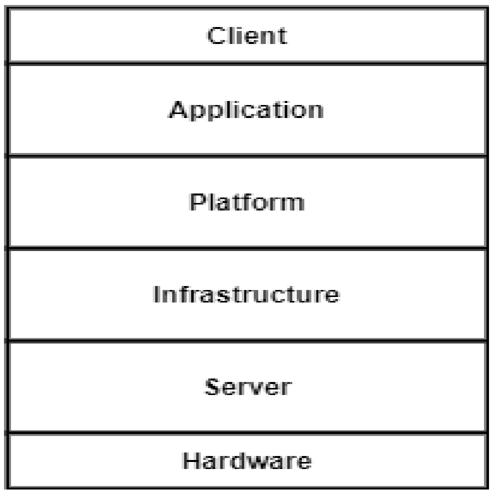

Figure 2.2: Layers of cloud

For cloud computing architecture, the cloud can be divided into six layers shown in fig 2.2 [55].

The lowest layer of the cloud consists of hardware like cables, plugs, switches, etc. The next layer above the hardware is the server. Cloud hosts a large number of servers. Servers are installed and maintained according to the need of the cloud owner. Servers vary from simple authentication servers with low resources to large data processing servers with thousands of cores of processors. The next layer is Infrastructure. Resources within a server or distributed among multiple servers can be combined using virtualization techniques. These packaged virtualized resource bundles are called infrastructures of the cloud. Each Infrastructure can hold one or multiple platforms. Hence, the platform layers come in. Here, it can host one or multiple applications. The application layer holds all the applications hosted in the cloud. Finally, the client layer provides simple client interfaces to connect with the applications of the application layer.

2.1.3 Cloud Deployment Models

There are various cloud deployment models based on their usage, requirement, and user base. We discuss the five most commonly used models. They are Public cloud, Private cloud, Hybrid cloud, Community cloud, and Virtual Private cloud. A representation of various cloud models is shown in fig 2.3.

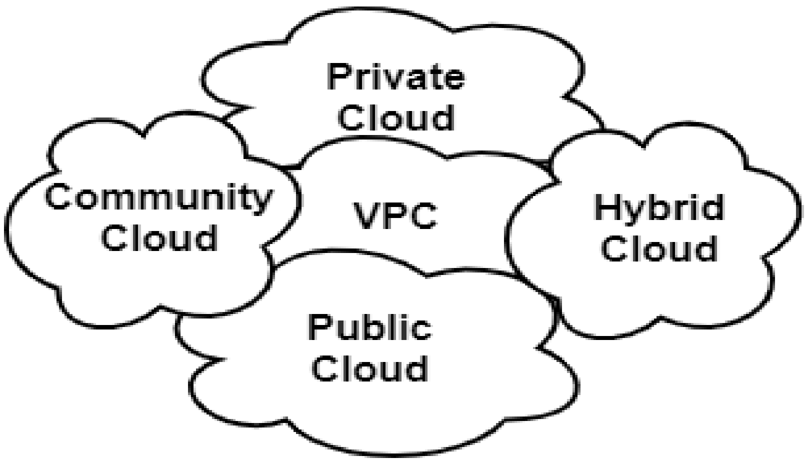

Figure 2.3: Cloud deployment models

Public Cloud

A cloud system developed and managed by service providers but open to being used by the general public is called a public cloud. It is also known as External Cloud, Multi-tenant Cloud. The main advantage of using the public cloud is that the users do not have to worry about cloud management while they can focus on the application. But a primary concern of the public cloud is security through data bridge on the public cloud is rare. It is especially appealing for small users and enterprises with no budget for large infrastructures like data centers. Users get the service via web browsers removing the necessity of installing any additional application for the service. Users pay only for the duration of service, known as the pay-per-use model. Some prominent public cloud providers include AWS, Azure, and Google Cloud.

Private Cloud

A private cloud is a special kind of cloud that is managed, maintained, and used exclusively only by the consumers of a single organization. It is also known as Internal Cloud, Enterprise Cloud, and On-Premise Cloud. Generally, if any organization has its own data center, a private cloud is the best option to utilize the already available Infrastructure and resources. It provides strong security, privacy, and governance over the data. However, the main drawback is the constant maintenance which can become too costly.

Community Cloud

When some organizations jointly develop and manage a cloud infrastructure that is used by the community from those organizations focusing on some common concerns is called community cloud. The cost of maintaining these clouds is spread among the members. So, it reduces the cost of maintenance significantly. Community clouds can also be managed by a third party if agreed upon. Typically, organizations with common shared interests become part of a community cloud.

Hybrid Cloud

A hybrid cloud is a unique cloud environment comprising private and public clouds. This kind of cloud is mainly used in enterprises because of its practicality in day-to-day use. On the one hand, it facilitates secure, private customer data. On the other hand, it can leverage external cloud resources as needed for large-scale computation. The accommodation of these two significant features has made the hybrid cloud the most popular choice among large organizations. It allows shifting workloads between private and public clouds when necessary, causing less cost than expanding the private cloud. It provides more flexibility and better options to businesses for data deployment.

In fig 2.4, we have shown an example of various communications and interactions of hybrid clouds. Enterprises have their private clouds, which directly communicate within the enterprise network. Then a public cloud also can connect with these private clouds. Enterprise networks also can directly communicate with the public cloud. But private cloud to public cloud communication should be happening with the help of a virtual private cloud to ensure data and network security.

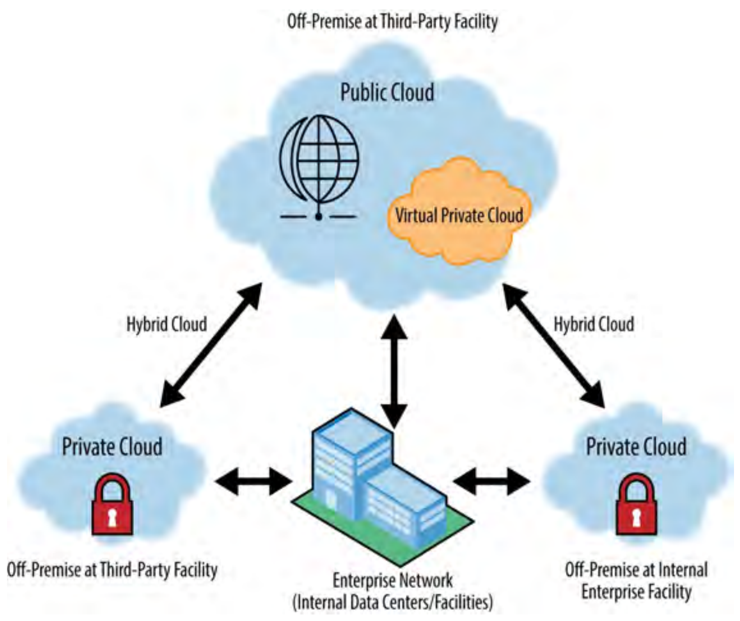

Figure 2.4: Interaction of different clouds

Multi-Cloud

The multi-Cloud deployment model involves multiple public or private cloud service providers or a combination of both. Here, multi-cloud service providers can engage multiple vendors for public and private cloud services. It provides redundancy and ensures a low chance of service interruption incidents like DDos, Disaster, or Power outrage issues. Users can choose a vendor based on the data requirement regardless of the public and private nature of the data. It eliminates the vendor lock-in limitation. The multi-cloud model offers high availability, low latency, flexibility, cost optimization, and risk mitigation. It guarantees high computing power and storage availability at the same time. However, dealing with multiple vendors and service providers may result in complex operational overhead and high attack surface area for system exploitation.

Virtual Private Cloud

Another recent cloud deployment model is Virtual Private Cloud(VPC), implemented by Amazon. It creates a secure bridge between users' private and service providers' public clouds. The security of the private cloud is conserved through a Virtual Private Network(VPN), and in a public cloud, end dedicated isolated resources are provisioned to provide data privacy.

2.1.4 Cloud Service Models

Cloud services are now considered a part of the larger Anything-as-a-Service model, also known as "XaaS". It means anything in a cloud environment, like storage, computing, network, database, logging, identity, mobility, content, etc., can be presented to a consumer as a service. Starting from the Infrastructure-as-a-Service(IaaS) to Data Recovery-as-a-Service(DRaaS) shown in fig 2.5, all these services are designed utilizing the typical cloud architecture of a simple request-response model without writing any extra codes. Most of these services are also open-sourced, meaning they are free to use and provide high flexibility, scalability, and easy-to-access features. Here we discuss some of the most widely known as-a-services.

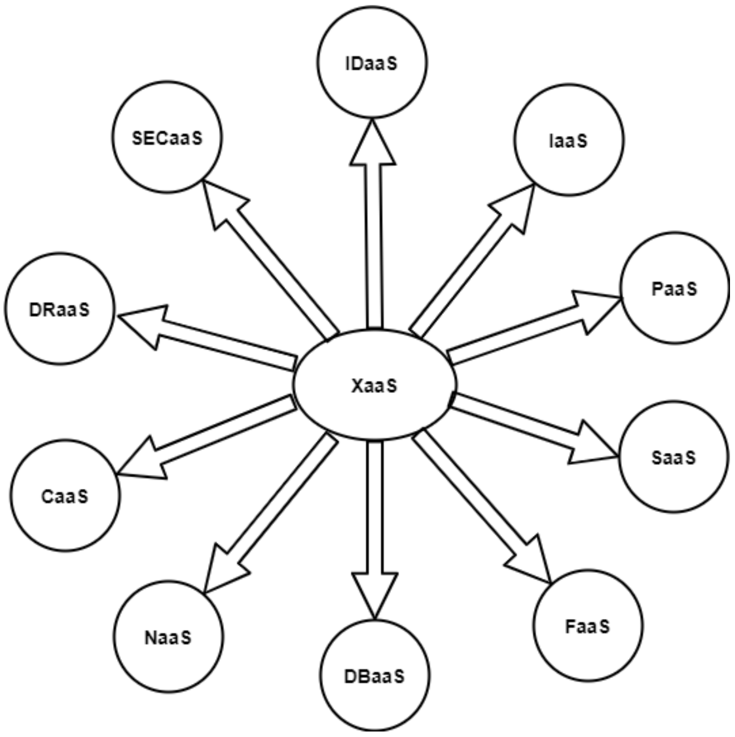

Figure 2.5: Cloud service models

IaaS

IaaS stands for Infrastructure-as-a-service. In this service model, service providers allocate virtual resources like storage, processor, networks, and other resources upon the request of users. Users utilize these resources according to their needs and emancipate them after usage. Users are only billed for the time that the resources were in use. The main advantage here, this model removes users' need for infrastructure amelioration for ephemeral demands. Users do not need to intervene in the resource management and control of the resources. Utilizing the distributed computing systems of the service provider, consumers can achieve the fastest service delivery rather than waiting for the tasks to finish one by one until the end. Thus, providing ultra-high scalability and concurrent execution for large and complex computations. IaaS service examples include GoGrid, Amazon EC2, Amazon S3, etc.

In fig 2.6, we find a general concept of the IaaS service model. Here, various clients are connected with small-scale cloud data centers managed by the broker. These small-scale cloud data centers are called cloudlets. The broker maintains the cloudlets with the help of cloud info service. The broker is connected to a larger data center that hosts many VMs. The service provider manages all the cloudlets, the broker interfaces, and the data center behind them.

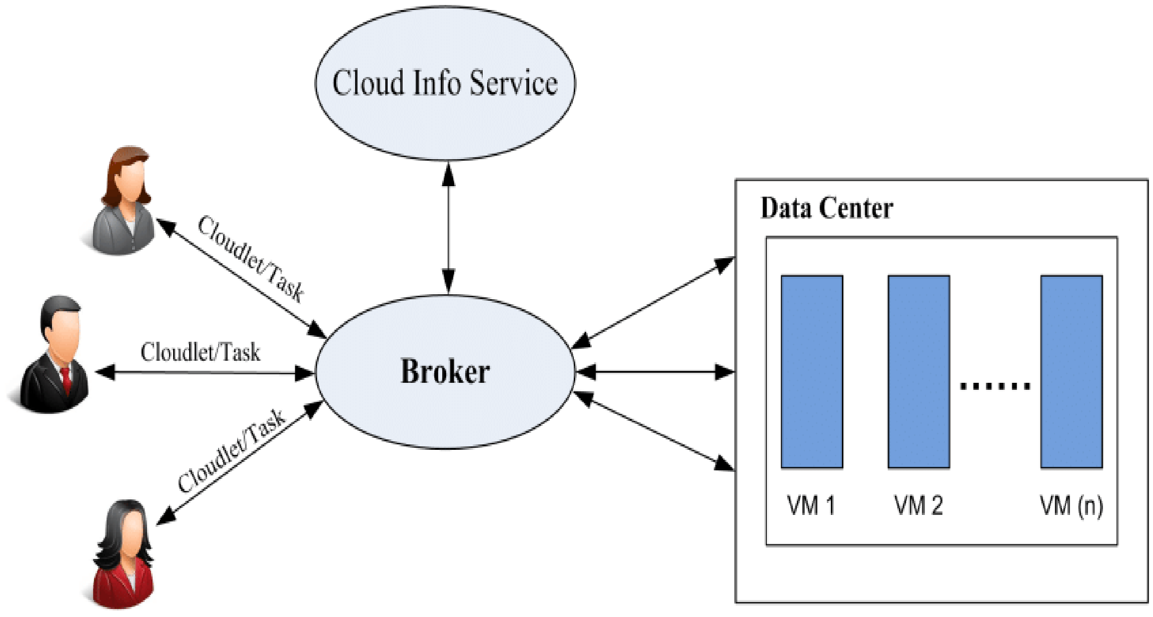

Figure 2.6: Infrastructure as a service model

PaaS

The Platform-as-a-Service model is a service delivery model where service providers offer tools and software for the developers to build their services by combining those resources. It removes the need to buy new software and tools for the development process, and developers do not require to worry about the issues related to software life cycles like version management of the platform, upgrading the platform, libraries, packages, and codes according to it. It becomes the service provider's responsibility to grant the opportunity to developers to focus on the application services. Services of PaaS include developing, testing, deploying, hosting, versioning, monitoring, etc. It also allows developers to run multiple versions of a system simultaneously. However, a significant drawback of PaaS is the need for more tools to migrate old services from on-premise to the PaaS platform and support only a small bunch of programming languages. Microsoft Azure, Google Sites, SAP Cloud Platform, etc.

The basic architecture of the PaaS service is shown in fig 2.7. Here, we can see the service model of the popular PaaS service provider Heroku. It helps developers with a container management ecosystem to make the software delivery process more app-centric. Developers utilize all the tools and platform features to create the desired application and services. The platform handles all the maintenance, versioning, load balancing, and package management tasks. It facilitates developers to focus on application creation and delivery process. Heruko keeps the application up-to-date by managing the patching, upgradation, and other maintenance-related tasks.

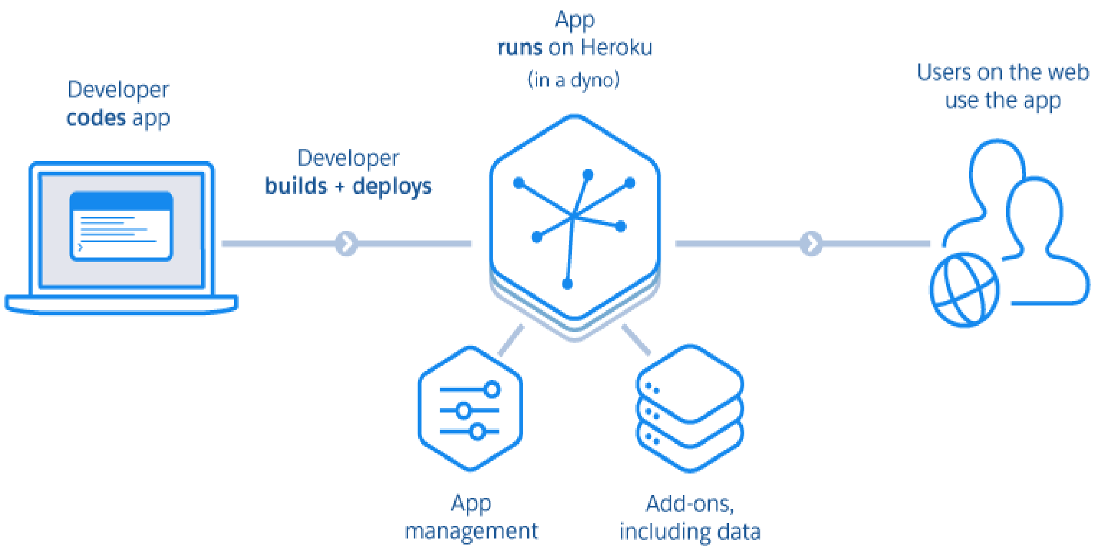

Figure 2.7: Platform as a service model [52]

SaaS

A Software-as-a-Service model is a software licensing and delivery model where users use licensed software on a subscription basis. Users use the software from their devices' web browsers. Though researchers argue that it goes against the free software model, it has gained massive popularity among consumers. The easy access feature removes any need to install software and hardware requirements. That's why it's also called On-demand software, Hosted Software, and Web-based Software. Users can customize the configuration based on their needs. However, these settings are based on predefined customization options of the software. It also enables distributed computing features of the software back end. The major drawback of SaaS can be latency, over-dependency on internet speed, and data security. An example includes Salesforce.com, Gmail, Facebook, etc.

We have shown a fundamental structure of SaaS in fig 2.8. An application of SaaS follows a subscribed-based policy. The application is hosted in the cloud and available to the users through an application interface. Here, we see that behind the application lie servers, databases, and codes. The vendor also manages all these components and hosts in the cloud. The vendor manages all the servers and handles scaling, load balancing, and versioning. Users only access the application.

FaaS

Function-as-a-Service is the newest service model among the cloud service. In this model, service providers allow users to design and run their application functionalities utilizing the cloud resources provided by the service provider without going through the complexity of building

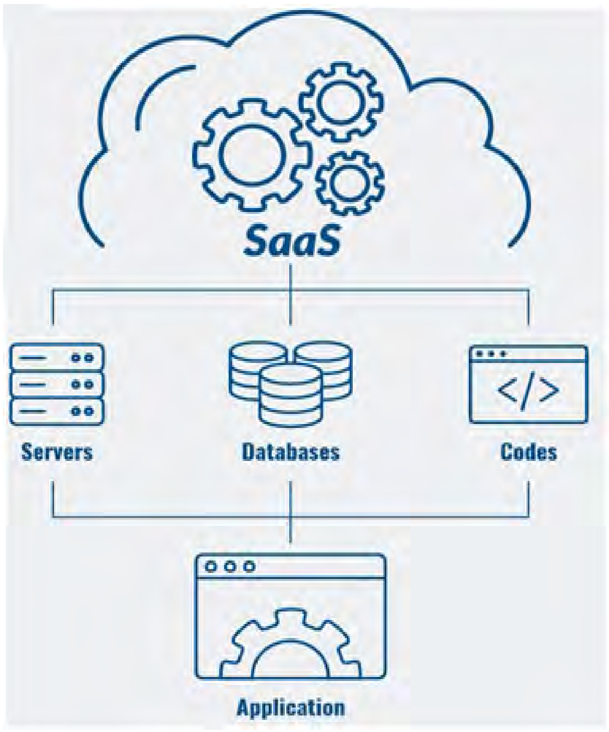

Figure 2.8: Software as a service model

and maintaining the Infrastructure for the development process. Unlike PaaS, FaaS does not require a constantly running process. The FaaS model only starts a process upon getting a request for specific function execution. As a result, processing initial requests may take several seconds, but consequent requests are processed within milliseconds. Users only need to pay for the function execution time resulting in lower cost and higher scalability, allowing cost-free idle time. Amazon Lambda, OpenLambda, and Microsoft Azure Functions are some examples of FaaS services.

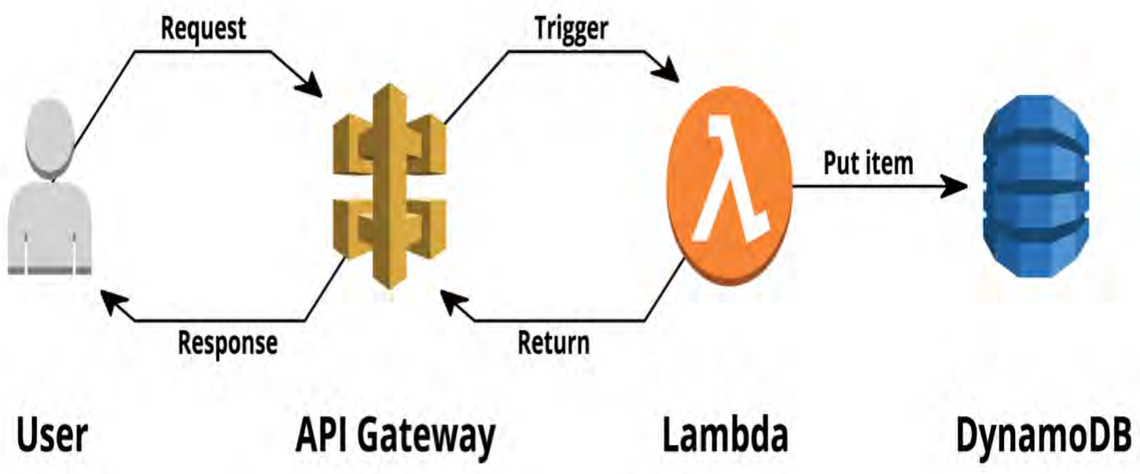

Figure 2.9: Function as a service model of AWS

Fig 2.9 provides a general architectural overview of FaaS applications. A user sends requests to the API gateway of the FaaS service provider. The API gateway triggers a lambda function based on the request parameters and configuration status. the lambda executes a request and completes a task; for example, put an item in DynamoDB as shown in fig. The lambda returns the output to the API gateway. The gateway passes the request to the user. The service provider manages resources, libraries, packages, and environments for the lambda that executes when a request arrives.

DBaaS

Database-as-a-Service is a service model where users can set up, operate and scale using a standard set of primitives in any database that the service provider manages. So, consumers do not need to worry about what kind of database he is using as the same set of interactions are used for SQL, NoSQL, and Object-Oriented Databases. Amazon first introduced it with its Amazon RDS service in 2009. The abstraction of the service provides more features like versioning, scalability, flexibility, migration, provisioning, and concurrency. DBaaS also ensure data security and reliability through its huge distributed cloud setup. Many applications can use the same data without any configuration changing though they can be for different platforms like Android, ios, or windows. DBaaS examples are IBM Cloud Databases, Oracle Autonomous Databases, Google BigQuery, etc.

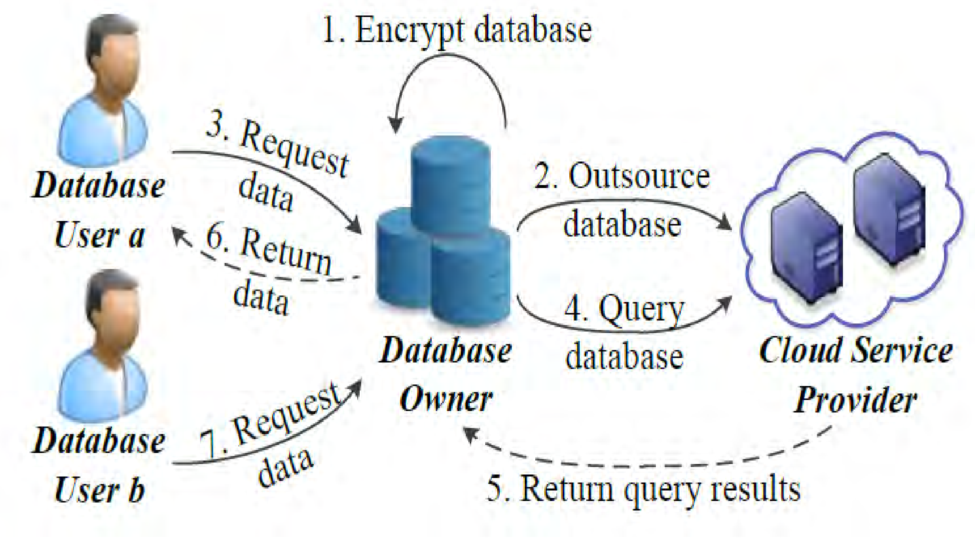

Figure 2.10: Database as a service model [126]

Here, we show an example of DBaaS in fig 2.10. We find that database users request data from the database owner. The database owner, however, does not host any data permanently; instead, he outsources data from a Cloud Service Provider (CSP). Based on the request received from the database users, the database owner sends a query to the CSP. The CSP fetches that data with a query result returned to the database owner. The database owner then returns the requested data to the database user. The database owner manages user privileges, user groups, and user policies. CSP ensures data integrity, confidentiality, and security.

NaaS

NaaS stands for Network-as-a-Service. It can be defined as a service delivery model where service providers supply a virtual network environment to the customer by consolidating network hardware and software resource. It eliminates the need to set up any network for testing and simulation tasks for the consumer. The network virtualization technique plays a key role in NaaS. Service providers utilize the same physical network to build multiple small or large virtual networks. These networks can communicate with each other for performance evaluation. It adds availability, reliability, and agility for contemporary static network environment. NaaS provider includes Senet-LoraWAN,

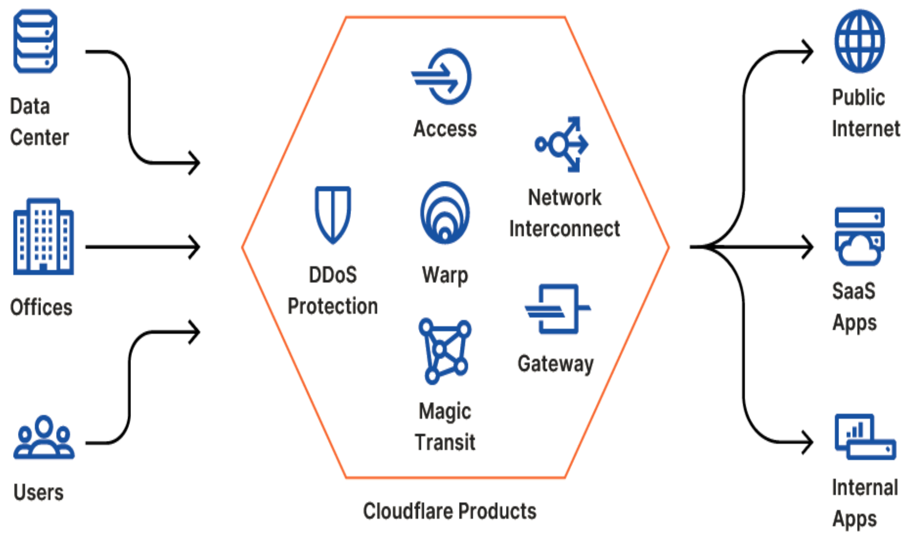

Figure 2.11: Network as a service model

An orthodox example of a NaaS service provider is Cloudflare, shown in fig 2.11. In the figure, we can see that NaaS provides a bridge between clients and applications. Clients from enterprises or general users use NaaS-provided networks to communicate with the application, websites in the cloud, and the public internet. Cloudflare's NaaS service includes DDoS protection, Warp, Internet Access, Network Interconnections, Transits, and Gateway services. As it runs on a subscription-based policy, users are charged based on the number of subscribed features. Enterprise users can utilize customized subscription plans based on their requirements.

CaaS

Container-as-a-Service is a cloud service model that lets developers deploy and manage applications through container-based abstraction using on-premises data centers or the cloud.

The service provider offers an orchestration framework in which the containers are maintained automatically. It encourages the current containerization trend for cloud-native apps and micro-services. Consumers only pay for the resources consumed by their containers for the running duration. CaaS stands in between IaaS and PaaS. Users can achieve portability and flexibility as developers can efficiently move them between cloud environments. Developers can scale it both vertically and horizontally. Because of the isolation and autonomy nature of CaaS, an application can effectively achieve increased security, efficiency, and reduced latency. The most famous example of CaaS is Kubernetes by Google and Redhat OpenShift.

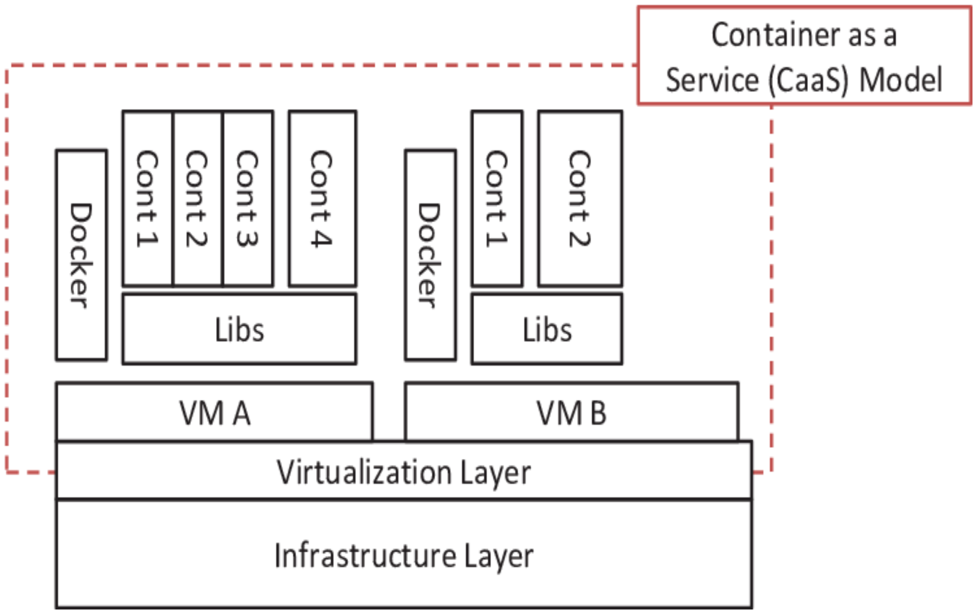

Figure 2.12: Container as a service model

In fig 2.12, we have shown a basic structure of CaaS services. CaaS service providers provide the underlying virtualization and virtual machine setup. Here, we can find that two virtual machines have been created: VM A and VM B. With the help of docker, configured and prepared by the vendor, developers deploy their containers into these virtual machines. As the containers are transferred into these virtual machines, appropriate libraries crate the compatible environment for the containers to run appropriately on these virtual machines. Thus developer remains free from the issues like memory management, CPU management, or disk management.

DRaaS

Data Recovery-as-a-Service offers replication and hosting-related services for physical and virtual servers to provide fail-over support in case of catastrophic events like a natural disaster, power outage, or any other kind of business disruption. The requirements and expectations are documented in the service level agreement, and the service provider offers DRaaS service on a pay-per-use basis. The off-site data recovery lets the service provider take a DR action plan in times of actual disaster since off-site locations are less likely to be affected than itself. It removes the cost of maintaining multiple secondary backup data centers for customers. DRaaS providers use automated, intelligent data replication mechanisms rather than like-to-like ones. An important issue is trust between enterprise and service provider, data security, and effective data replication. Flexential, Carbonite, and Infrascale are some DRaaS service providers.

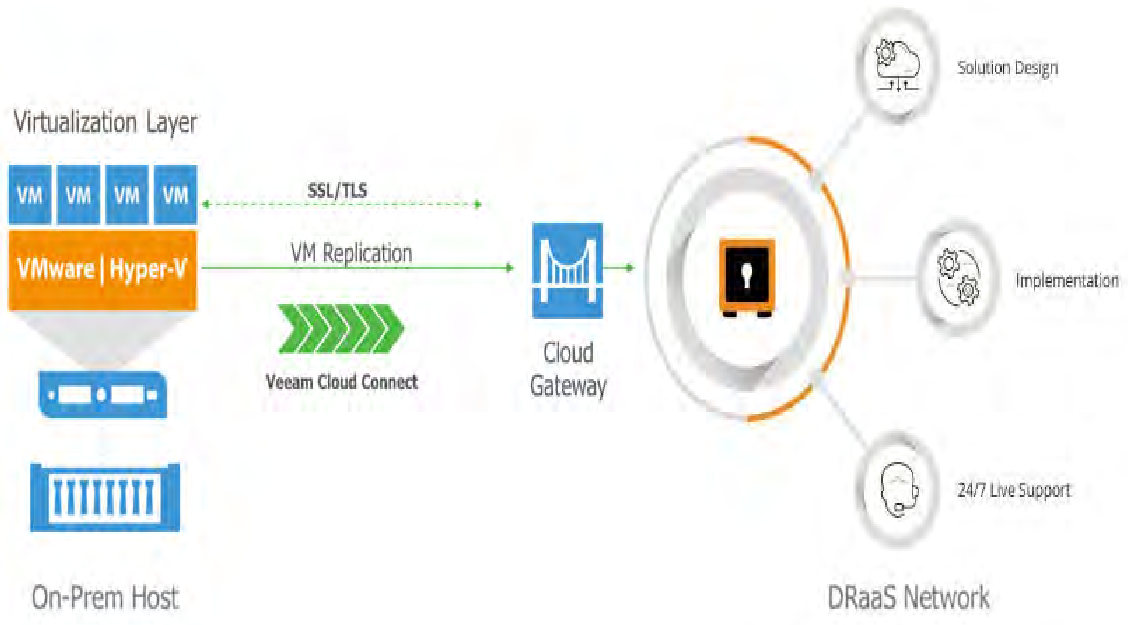

Figure 2.13: Disaster recovery as a service model

A popular DRaaS vendor in the international market is Veeam. Fig 2.13 shows a disaster recovery service model of Veeam. We can find that Veeam maintains a replication of each VM of the on-premise server VMs. Encryption-based SSL/TLS connection is used in the replication transfer process between the host and the cloud recovery site. The secured cloud gateway ensures that all replications are kept in the cloud in the most orderly and secure manner. Besides, Veeam provides solution design and implementation features for all replicated applications. 24/7 live support is also required to guarantee a secured, uninterrupted backup process behind the scene.

2.1.5 Virtualization

Virtualization is a term used heavily in a cloud environment despite the birth of the concept in the 1960s. Virtualization can be defined as a transparent emulation of the computer resource to facilitate higher access of the resources to the users, which is absent in physical form. The computer resources include storage, network, hardware, etc. Significant benefits of virtualization include memory expansion, resource optimization, fault tolerance, and higher availability. Developers use a tool called *hypervisor* to create a virtual machine over the host machine.

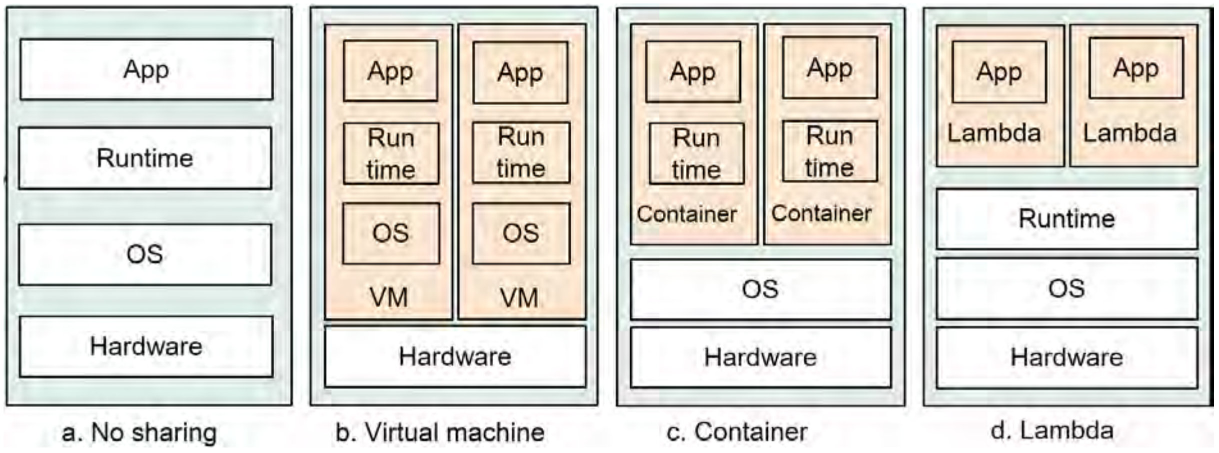

Figure 2.14: Virtualization trend

In earlier settings, applications needed their environment, including hardware and software. The introduction of virtualization has made it possible to share the same hardware for multiple operating systems (OS) known as Virtual Machine (VM) [17]. Later container-based virtualization allowed us to use the same OS and H/W for different applications. The repackaging of Unix-style processes combined with distribution tools has gained huge favor among developers, known as Docker [41]. However, the Lambda model is the latest virtualization model that allows sharing of runtime resources. It has revolutionized the concept of Serverless Computing.

OS-level Virtualization

OS-level virtualization is a special kind of operation system virtualization that allows running multiple isolated userspace instances over the same kernel [98]. The userspace instances are known as containers. These containers work like their own entities having all the features of a general OS. The containers allow the allocation of unique resources like CPU, RAM, storage, and network settings. They can be run separately or concurrently based on the demand. Different applications are assigned into individual containers to ensure isolated and dedicated resources for each application. The containers can communicate with each other for optimum resource usage. Features like high concurrency support, resource optimization, load balancing among the nodes, security, and hardware independence are implemented through OS-level virtualization. A drawback of OS-level virtualization is flexibility. It does not support guest instances that are different from host instances. For example, the Linux host instance functions well with other Linux distributions but does not support Windows.

Storage Virtualization

Storage virtualization is a pooling technique that consolidates storage from multiple physical storage devices and makes it available to applications that appear as one harmonious storage device. This virtual technique allows applications to spread their data over multiple storage devices removing the single-point failure problem of the data. The storage is virtually controlled from one central console. The technology requires software to identify available physical storage spaces and accumulate them to present them in a virtual environment. These virtual storage have a standard read-write operation mechanism for physical drives. One significant advantage of storage virtualization is data redundancy and replication over multiple disks. Two primary storage virtualization methods are block-based storage virtualization and file-based storage virtualization.

Network Virtualization

Network virtualization creates an abstraction of network resources by combining physical and software network resources to be delivered through a central software-based administrative entity. These abstractions are called virtual networks. This virtual network allows network administrators to move virtual machines over different domains without configuring the network. It also enables running multiple virtual network overlays over the same physical network. Thus it makes the network agile, dynamic, and efficient and reduces the provisioning for new application time from week to minute. This virtualization is popular with developers' software testing as developed can simulate the network environment and deploy the application to determine performance over any network. VLAN is the most common example of network virtualization by virtualizing large LANs into smaller network modules.

2.2 Serverless Computing

Serverless computing is just the latest edition of the "as a service" model of cloud computing. Different versions of the model, like Infrastructure as a service (IaaS), software as a service (SaaS), and Platform as a service (PaaS) have been provided by the cloud providers offering customers different levels of resources like software subscriptions to whole computing systems. Serverless is in the fourth category: Functions as a service (FaaS) [26], which is based on runtime-level virtualization. In the FaaS model, the developer does not need to think about the runtime that the code uses. He needs to write code based on the platform and use a way that the platform supports and uses the library provided by the platform.

With the Lambda model, applications are seen as a set of functions called lambda functions instead of a collection of servers and developers. Different lambda handlers are written to handle different kinds of requests. As we browse available lambda services, we find that the lack of complete control over the lambda model hinders the implementation of the lambda model in real life. All the core sections: request distribution, code files, and code executions- are hidden from the developer because of security, consistency, resource management, and performance issues [113]. The lack of control in code execution on specific containers results in increased start-up time, also known as the cold start problem. Specifically, the cold start mainly refers to the time delay from receiving a lambda request in the application to the start of executing code for that request. Alternatively, the time of preparing a container to serve any request is called cold start. As a result, the absence of any built-in mechanisms to counter cold start leads to the same problem for all lambda models in the industry. Thus, the only possible way out from the developers' end is to send continuous hello request actively to keep those lambda containers warm.

2.2.1 Emergence and Contextualization

Using the serverless platform can primarily be compared with a task like loading an image in cloud storage or adding an image thumbnail to a database table. Because similarly to the above, in serverless user writes a cloud function into his chosen language, uploads it into the platform, and picks an event that should trigger the running function. Every other thing is the concern of the service provider. Service providers' responsibilities include time-consuming jobs like instance selection, scaling, deployment, versioning, security, logging, monitoring, etc. Hence, allowing much freedom to the programmer and taking advantage of a high-level language. We will call the traditional approach of application development with consideration of server environment as serverful cloud computing. A responsibility comparison with serverful and serverless approaches is shown in table 2.1, indicating how serverless architecture provides programmers opportunities to focus on the core application.

The main difference between serverful and serverless platforms can be discussed in three main points.

- *Separation of computation and storage:* The storage space and computation are provided, provisioned, and scaled separately. In traditional serverful applications, the common practice is to have them on the same server, which becomes an impediment whenever only one needs to scale exceptionally high and independently. For serverless, storage can be provided by the same or separate provider and keep them on different physical entities than physical computation machines. The computation is also stateless.

- *No resource allocation:* The execution of the function code does not need to allocate and manage memories before its' execution. Most serverful applications need to explicitly manage their memories to ensure proper code execution in runtime because, otherwise, memory leaks are risky. But for serverless architecture, memories are fully managed by the service provider, and memories get allocated at runtime according to the need for resources for execution.

- *Payment based on the resource used, not resource allocated:* Previously, in serverful applications, users needed to pay the server provider according to the resource provisioned for that specific application. Developers tend to allocate additional resources than they need to support burst requests in peak hours. So, they had to pay for those additional resources for the whole time though they are only used in peak hours. But in a serverless, billing is done only for the resource used. Hence, eliminating the cost of standby resources. But users get additional resources in peak hours utilizing serverless platforms' dynamic resource allocation policy and are only billed for those peak hours.

The popularity of the serverless platform is influenced strikingly by cloud functions(FaaS). However, FaaS also owes its success to BaaS offerings as they existed in cloud platforms and services from the beginning. These services represent serverless computing in a more general form. An essential advantage of serverless over PaaS is allowing the developers to bring their libraries to the platform-providing their vast support for customizing applications according to their use cases. Kubernetes has become a famous "container orchestration" technology to deploy microservices. However, some confuse it with serverless architecture. A notable difference is Kubernetes is mainly a management model for serverful applications developed by Google. The primary use case for Kubernetes is to deploy an application on a cloud built for on-premise use only. Thus, bringing out the idea of a hybrid cloud. A hybrid cloud is when an application is partly run from local hardware, and other parts are run from the cloud. However, the adoption of primarily serverless diminishes the value of hybrid could. Another difference of Kubernetes is it is billed according to the resource allocated rather than the resource used. Edge computing can also be seen as a unique serverless platform with embedded serverless execution in edge devices.

2.2.2 Serverless Platforms

Various platforms have provided serverless service. In this section, we provide a brief description of major serverless service providers. We discuss their architecture, features, use cases, distinctive traits, and high-profile consumers. Though other emerging open-source platforms are expanding their businesses in serverless service, we only mention the key players that can provide enterprise-level solutions and support such large applications.

AWS Lambda

AWS Lambda was the first public cloud service provider to initiate serverless computing service for consumers in 2014. AWS utilizes its other services like AWS S3, AWS Edge, AWS Fargate, DynamoDB, AWS EFS, Amazon Athena, Amazon Kinesis, etc., to ensure isolated and independent service with AWS SAM. The event-driven computing service runs code responding to events and automatically manages computing resources. AWS starts a lambda container and provides the environment to execute codes for use cases like image uploads, object uploads, updates to DynamoDB, and sensor readings from IoT-connected devices. AWS containers are based on native Linux architecture. Provisioning based on custom HTTP requests to trigger backend services and scale back when resources are not required can be done using lambda. AWS Gateway API and AWS Cognito can handle the authentication and authorization tasks. AWS supports various development languages like Node.js, Python, Java, Ruby, Go, C#, and .NET. The maximum compressed size limit of the lambda package is 50 MB, and the maximum uncompressed size limit is 250 MB.

IBM OpenWisk

IBM announced its own version for function-as-a-service back in 2016 based on Apache OpenWisk. OpenWisk also works similarly to AWS Lambda with an event-driven architecture and lets developers trigger responses to the events. It enables the user to design microservices that execute code when an event like a mouse clock or sensor data retrieval from IoT end devices. It offers chaining to connect multiple applications developed separately. Integrated container support eliminated vendor lock-in by letting developers run custom codes on Docker containers. It supports languages like Node.js, Swift, PHP, Python, and Java. It is also compatible with other built-in cognitive services like Watson and Weather. A differentiating point of IBM OpenWisk from Google Cloud Functions and AWS Lambda is their emphasis on Docker container integration. Use cases of IBM Openwisks are the serverless web application, API development, and integration, mobile applications with serverless back ends, making searchable videos, microservices, etc. Notable customers of high ranks include *GreenQ*, *articoolo*, *SiteSpirit* etc.

Google Cloud Functions

Google cloud function is a serverless platform to develop and execute cloud functions and use such functions as building blocks for large applications. It was developed in 2016 to connect various google provided services and bring them in front of the developers so that they can be monitored with the help of tools like *Cloud Trace* or *Cloud Debugger*. These functions let developers trigger codes from other google services like Google Cloud, Firebase, and Google Assistant, or even via HTTP calls from any web or mobile application back ends. These cloud functions can be written using JavaScript, Python 3, Go, NodeJs, or Java runtimes.

Cloud functions also have access to the Google Service Account credential, allowing seamless authentication with other cloud services like Cloud Vision. Binding a cloud function with a trigger of an event allows users to catch the event and act on it accordingly. The fine-grained, on-demand nature of Cloud Function makes it a perfect candidate for lightweight AAPIs and Webhooks. Other use cases include sentiment analysis, video and image analysis, virtual assistants and conversational experiences, real-time stream processing, real-time file processing, serverless IoT backends, serverless mobile backends, integration with third-party APIs, etc. Major customers of Google Cloud Functions include *ebay*, *PayPal*. *HSBC*, *20 Century Fox*, *LG CNS* etc.

Azure Functions

An azure function is an event-driven serverless computing platform that works with triggers and binding of functions with those triggers to facilitate simplified and accelerated application development. The platform started its journey in limited release in March 2016 by Microsoft. The azure function was designed to extend existing Azure application services and empower them with capabilities like code implementation triggered by events occurring in the Azure cloud or other third-party services. Fully integrated with other Azure services and development tools, its end-to-end development experience allows developers to build and debug functions locally on any major platform like Windows, Linux, or macOS.

Automated scaling and provisioning of serverless nature provide simple orchestration for complex orchestration challenges. The pay-per-execution trait encourages deploying the same application in different hosting environments according to the application's needs. Azure function support various languages for its application developments like .NET, Python, NodeJs, Java, etc. The built-in security and monitoring tools like *Azure Application Insights* and *Azure Monitor* equip developers to spot bottlenecks and failure hotspots across the application architecture pipeline. The enterprise-grade FaaS platform also has PowerShell support and *Durable Function* feature, which provides a way for stateful applications to be defined in a programmatic serverless manner. Use cases of Azure function include time-based processing(Corn workloads), Software-as-a-Service (SaaS) event processing, mobile backends, real-time stream processing(IoT), or real-time bot messaging. High profile Azure functions include *Relativity*, *FujiFilm*, *direct.one*, *Hotailors* etc.

Oracle Functions

Oracle functions is a Function-as-a-Service provider that allows serverless benefits like fully managed, on-demand, high scalability, and multi-tenancy service. It utilizes docker containerization and enterprise-grade Oracle Cloud Infrastructure. It is built upon the open-source *Fn project*, a container native serverless platform. Oracle functions are also integrated with Oracle services like Oracle Cloud Infrastructure Identity and Access Management(IAM) to provide secured authentication and authorization service. Users can access via Console, CLI, or REST API. We can automate various tasks based on the state change of infrastructure resources of oracle cloud using the Oracle assigned identified called Oracle Cloud ID(OCID). However, the limited availability release of Oracle Functions has the constraint of a maximum of 10 applications and 20 functions in a tenant. The maximum data limit to send to a function or to receive as a response has a maximum limit of 6MB.

Oracle functions support various languages like Java, Python, NodeJs, Go, Ruby, etc. The administrator can also control the access of different resources by setting up **groups**, **compartments**, and **policies**. The rules defined for each entity are easy to set up and intuitive. It also provides predefined event types for subscribing to Oracle Cloud Infrastructure Native Service resource changes. Use cases of Oracle functions include back ends of event-driven applications for web and mobile, real-time file processing solutions, real-time stream processing, DevOps, and enterprise-level security solutions.

Cloudflare Workers

Cloudflare workers are the serverless platform by Cloudflare to allow developers to develop applications without the concern of infrastructure. Cloudflare workers' runtime uses the *V8 engine*, the same engine used by Chromium and nodeJs. These workers' functions run on the globally distributed network of thousands of machines called *Cloudflare's Edge Network*. Each machine hosts an instance and can run thousands of user-defined apps. Workers are built on lightweight contexts that group variables with the code allowed to mutate them, known as isolates. When other serverless service providers use containerized processes, Workers pay the overhead of JavaScript runtime. However, the isolate can run essentially limitless scripts and start almost a hundred times faster than a node process on a container. Isolates are also resilient and continuously available for the request duration, but isolates may get evicted in case of hitting the script limit, resource shortage, etc. Workers' instances may handle multiple concurrent requests in a single-threaded event loop. But there is no guarantee that two requests will land on the same instance. Computation per request is another essential trait of Workers.

Cloudflare Workers supports languages like JavaScript, C, C++, and Rust. It also supports 0ms cold starts with script isolates. It has built-in support for Edge storage: a low-latency key-value data store. It provides the ability to generate static assets like images, SVGs, and PDFs on the fly and to deliver them as static assets to users. It is also ten times less expensive than other serverless platforms with an exceptional free plan. Uses cases of Cloudflare Workers include customized affordable e-commerce experiences, enforcing secured custom authorization and authentication, robust, granular test deployment with Workers KV, and controlling cheaters, spammers, trolls, and other bad actors by storing IPs and user IDs. Major high-profile customers of Cloudflare Workers are *npm*, *Discord*, *Maxmind*. *Optimizely*, *Cordial* etc.

2.2.3 Challenges Ahead

In this section, we explore various limitations of contemporary serverless architecture. We mention various problems identified by the application developers and how they are affecting user experience. These limitations can also be viewed as challenges ahead for serverless platforms to be adopted as the first application development platform choices for developers. We also tell how these challenges could be tackled and various proposed methods by the developer to avoid various bottlenecks and overheads of traditional serverless architecture. We discuss issues like function chaining, heterogeneous hardware support, fault tolerance service, communication overheads, best storage solution, vendor dependency, start-up minimization, coordination mechanisms, etc.

Function Chaining

Diverse applications are required to fulfill the need of clients; sometimes, developers may need to design applications where multiple functions need to be executed to serve the request. This sequential execution of FaaS functions is an important requirement for applications with heavy workloads. However, this service is only available to a limited number of serverless service providers like IBM Action Sequences or AWS Step Functions [9]. The trigger process of these sequenced functions can be divided into two types. (1)*External event* triggers functions using the client request process. (2)*Internal event* triggers a new function as a part of the execution workflow of another function. Existing serverless architecture treats them as the same and often requires a full end-to-end function call path. Hence, it incurs undesired latency. But such latency can easily be avoided by sharing information about function states of member functions within the same function stack.

Heterogeneous Hardware Support

In this age of machine learning and artificial intelligence, it has become eminent that cloud services need to process large workloads that have never been seen before. As a result, the need to combine specialized hardware has become critical for service providers. The stagnant performance improvement of x86 microprocessors and DRAMs approaching the maximum capacity per chip are indications for the service providers to explore other approaches to meet future challenges. One such approach can be the widespread use of *Domain Specific Language*(DSL) [50]. DSL enables developers to write applications specifying cost-effective hardware requirements. Even developers can code targeting such hardware resources. It is possible to write code using a high-level language like Python or JavaScript that would create hardware-software co-design leading to a language-specific custom processor. These processors can be one to three magnitude faster than contemporary ones. We are already watching the evolution of such designs as Graphics Processing Units (GPU) to Tensor Processing Units (TPU). TPU outperforms GPU by 30x. Though this hardware-software co-design may clash with the "hardware independence" philosophy, we may think of it as soft binding to hardware rather than a hard one. So, dynamic physical decision-making to utilize available heterogeneous resources on the fly may change the course of serverless architecture.

Ensuring Secured Fault Tolerant Service

The ephemeral and distributed nature of serverless applications has made it challenging to design a fine-grained security model. We need access to private keys, storage objects, and temporary local resources to develop such a model. One such model can be a capability-based access control mechanism using cryptographically protected security contexts. Cloud applications' physical co-residency causes hardware-level side-channel attacks or Rowhammer [62] attacks. A randomized scheduling algorithm with physical isolation can prevent such attacks. Another vulnerability of cloud functions is that they can leak access patterns and timing information through communication. The widely distributed nature of cloud functions makes these leaked sensitive information an excellent tool to interrupt services. The practice of decomposing serverless applications into small functions also exacerbates the situation. The adoption of specific network pattern protection algorithms can be the solution for it. Users demand fine-grained system-level security isolation for each function. But providing such function-level sandboxing without caching execution environment to facilitate state sharing between repeated function invocations with short start-up time is a daunting task for security researchers. One solution can be taking a snapshot of the instances of each function to start a clean slate.

Reduction of Communication Overhead

In modern cloud architecture, the standard communication mechanisms among different cloud entities are broadcast, aggregation and shuffle. These operations are incredibly familiar among machine learning and big data applications as they need to share information and communicate with different cloud entities. From the previous communication pattern of VM-based instances, we now have a function-based communication pattern which has become too costly in some cases. In VM-based architecture, local instances within the same VM could share data with a single message for broadcast communication. But in function-based communication, the locality concept got excluded. They result in the requirement message propagation for each of the functions. It has increased by message operation by k times, where k is the number of function instances per container. For aggregation, it has also increased over time. Moreover, for shuffle communication, the number of message operations is swelled by $k\hat{2}$ times. It is challenging to reduce the communication overhead because contemporary algorithms like leader election or consensus do not apply to the transient nature of serverless architecture. A robust and effective algorithm must be developed to reduce communication overhead.

Optimized, Efficient and Affordable Storage Solution

The current architecture of serverless platforms shows us the need for a different type of storage to make considerable performance improvements. Researchers propose two different types of storage *Serverless Ephemeral Storage* and *Serverless Durable Storage* [58].

The transient nature of serverless architecture explains the need for special ephemeral storage. This storage is critical to maintaining the application's state during the application's lifetime. Once the application finishes, it can be discarded, aligning with the stateless nature of serverless applications. Researchers need to build a distributed in-memory service with an optimized network stack to provide such a store. The state of the applications stored in this temporary storage will allow applications to share and exchange states between functions during the application's lifetime. The key features of such a storage solution are automatic scaling, the transparent allocation of memory, and free after the termination of the application with access protection and performance isolation.

Serverless database application requires long-term data storage and the mutable-state semantics of a file system. These applications need longer retention and outstanding durability than serverless ephemeral storage. Researchers propose to leverage an SSD-based distributed store paired with a distributed in-memory cache. A key challenge would be managing low latency in distributed entities, cost-efficiency and high performance, and combining multiple existing cloud storage offerings. Ephemeral storage, transparent provisioning, memory isolation, security, and high performance should also be available here. One significant difference would be that resource reclamation was automatic in the previous one but would be only based on explicit command in the latter. Moreover, it must ensure extended durability, confirming fault tolerance to any acknowledged writes.

Inevitable Vendor Lock-in

Though code dependency over the cloud platform environment is minimal, applications developed over serverless platforms can become highly dependent on the service provider. As the platform provides all the basic features like authentication, monitoring, and configuration management, developers require keeping that in mind when designing the application. Serverless architecture also facilitates incentives to let client applications connect directly to other resources that are required and utilized by the application. These services generally come through the service providers' API. It makes the application highly reliant on the platform. Consequently, it reduces the flexibility of the application. Hence, if developers want to move the application from one platform to another platform would require a significant level of rewriting of existing code. So, over-dependency on vendors has become a severe limitation for serverless architecture.

Improved Co-ordination Mechanism

The state-sharing model of serverless applications can be compared with the contemporary producer-customer design pattern. Consumers require to know the availability of products as soon as possible from the producers. Likely, one function might need to signal other functions about the state or condition of a function when available and might also require coordination with more functions. Such a signaling mechanism will improve micro-second level latency, reliable delivery, and broadcast with group communication. However, implementing such a signaling algorithm would require extensive exploration as currently distributed algorithms would not suffice the requirements.

Start-up Minimization

The start-up time can be divided into three parts. (1) scheduling and starting resources to run the cloud function, (2) downloading the application software environment to run the function code, and (3) performing application-specific start-up tasks such as loading and initializing data structures and libraries. Researchers have proposed lightweight isolation-based models as resource scheduling, and initialization may incur significant delays and overheads.

One approach has been proposed to reduce application environment-related delay based on unikernels [58]. Unikernels are specifically designed to avoid these overheads in two ways. One, unlike traditional operating systems where hardware detection, application of user configurations, and data structure allocations are made dynamically, in unikernels, these costs are averted by statically allocating data structure and hardware detection mechanism. Two, unikernels only include drivers and libraries strictly required by the applications. So, unikernels have a lower footprint as they are tailored for specific applications. Another popular approach is to dynamically and incrementally load libraries as applications invoke them.

Reduction of application-specific initialization delays may require some out-of-the-box thinking. One recent proposal by researchers is to include a readiness signal in API to avoid sending work to function instances before they can start processing it [58]. Cloud providers can also perform start-up tasks ahead of time. This can include tasks like booting a VM with a popular operating system and loading a set of pre-configured libraries based on the type of application.

Cookies and Sessions

The architecture of serverless is based on the short-lived, stateless lambda functions. But in serverful applications, users typically expect web applications that consist of many different but related interactions among web applications themselves. In such applications, the traditional practice of the developers is to keep a shared view of cookies from the browser, which is interacted with by the applications whenever necessary. But the stateless nature of lambda functions does not support this concept. So, it is challenging for lambda providers to allow developers a shared view of cookie sate across application calls issued by the end-users.

In modern web applications, two-way data exchange is typical between servers and clients during a single session. This exchange is typically managed and facilitated by WebSockets or by long polls. These protocols are especially challenging for Lambdas because those protocols are based on long-lived TCP connections. Suppose the TCP connections are maintained within a Lambda.

Handler customers will be charged even when the handler is idle, which directly contradicts the pay-as-you-go principle of serverless platforms. Alternatively, to manage TCP connections outside of the handlers, providers need to manage connections for new Lambda invocations and past invocations.

Data Aggregators

As we live in an era propelled by information, applications must evolve to handle and manage large data sets. Hence, applications now require to make search queries to search, feed and analyze them for application purposes. So, parallelism over different data shards is critical to managing such applications. But building such an application over lambda is highly challenging because of the need for exceptional lambda support.

A solution for such a case could be a coordinated tree structure of Lambda functions. Leaf Lambdas will filter and process data locally, and a front-end Lambda will combine the results. It would be critical in performance prospects if Lambda leaves could be co-located with the data when they filter and transform large data sets. Custom data stores need to be built with lambda leaves to achieve that. Hence, it opens the possible adoption of diverse aggregator applications for pre-processing the data. Various APIs for coordination with a variety of backends will be required.

2.2.4 Serverless in Action

A recent study was conducted by serverless.com in their blog in 2020 to understand the adaption rate, problems, feelings of developers, and prospects of serverless architecture in [47]. So, they created a survey name the "State of Serverless Community" survey. They distributed the survey through their newsletter, blog page, Facebook, Twitter, and other social media to connect with the serverless application developers. They found a total of 137 responses from all around the globe including participants from North America, Europe, and the rest of the world. Four aspects were analyzed from the survey: (1) Use case of serverless. (2) Frameworks used, (3) Tools and language used, and (4) Future optimism ranking. But to better understand the result we need to realize about the participants first.

Participants Analysis

To better understand survey responses, the participants were asked questions. Fig 2.15 shows the role distribution of survey takers in the pie chart. We can see almost 60% of the participants count as a developer, which can be further divided into three main categories Front-end Developer(8.5%), Fullstack Developer(9.2%), and Backend Developer(44.6%). Almost 28% of the participants come from positions like Engineering Manager(15.4%) and Executives(12.3%).

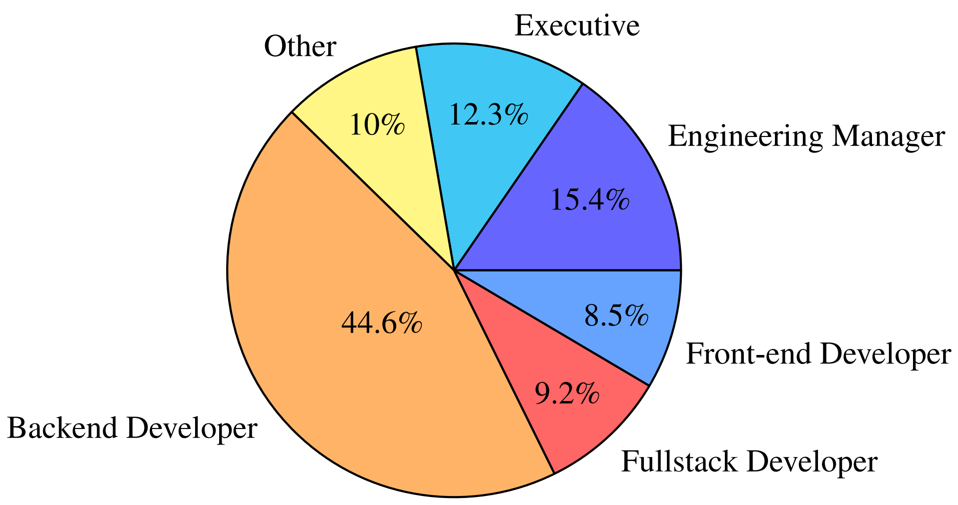

Figure 2.15: Role of serverless users

The rest, around 10%, come from various diverse roles like DevOps, Architect, Product Manager, etc. The popularity among businesses was also a significant concern. The survey shows start-ups are more likely to adopt serverless, counting almost half of the respondents. However, large companies also showed interest in serverless architecture, finding 39% of respondents are from SMBs and 15% are from enterprise-level solutions.

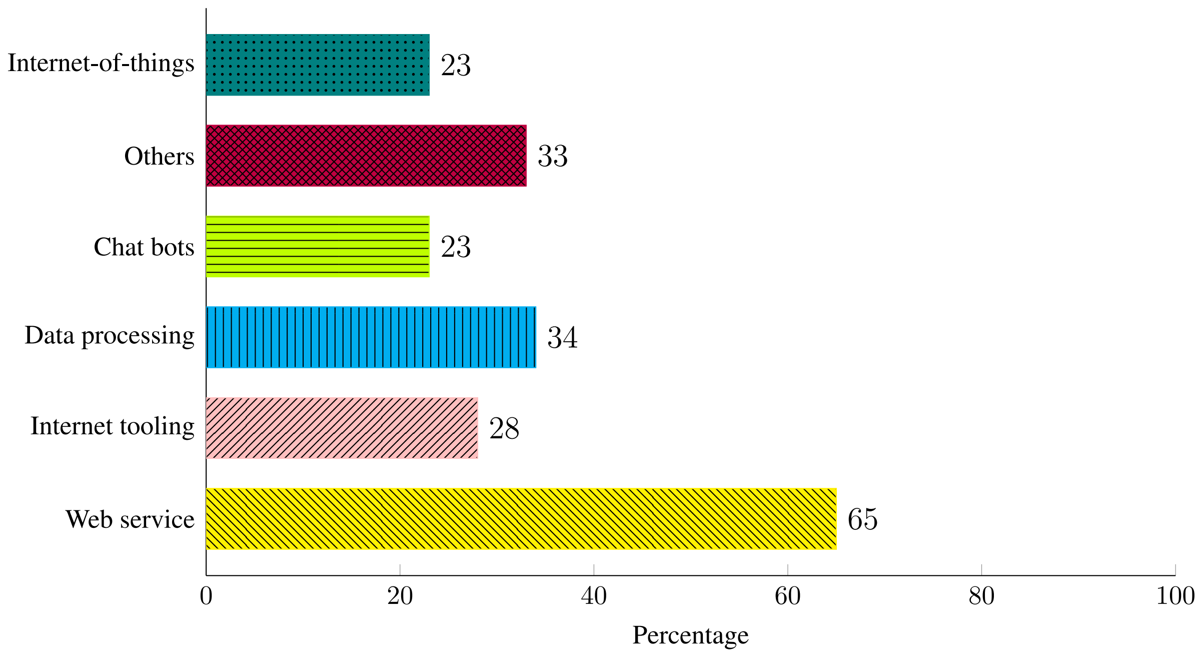

Figure 2.16: User cases of serverless architecture

Use Case of Serverless

Though serverless is still in its early days, it is utilized for mission-critical workloads by developers. From the survey, a horizontal bar graph is shown in fig. 2.16. We see about 50%

of the respondents use it for regular work purposes, which is enormous compared to its recent development. Then we find 21% of the survey takers use it for side projects, but 22% participants have experimented with serverless platforms but have yet to utilize it for projects. Among the use cases, we find that developers use it primarily for web service tasks like API development and management, consisting of 65% of the respondents. Chatbots, Internal tools, and Internet-of-things all count for over 20% individually. A significant 34% of the participants use it for data processing. However, 33% of the respondents utilize it for other purposes like mobile backends, research, and development tasks.

Providers and Frameworks Used

Here, we try to find the most accessible frameworks and providers of serverless architecture among participants. A breakdown of providers used by the respondent is shown in fig. 2.17. For providers, we find that AWS Lambda is the most popular for the choice of 96% participants. Then Azure came in second with 6%, and then behind that, Google Cloud Functions covered only 4%. Lastly, OpenWhisk and Webtask came last, with 2% voted by participants.

We show the findings by analyzing fig. 2.17. Surprisingly, 76% respondents prefer AWS serverless framework, also known as lambda framework. Apex was the second most popular, having only 10% responses. Others like ClaudiaJS, Chalice, and Sparta have 5% or fewer responses which means they are rarely popular among developers. However, 19% of survey takers do not use any framework, and only 6% use other frameworks. So, we can easily recognize the uncontested popularity of the lambda model among serverless architecture users.

Tools and Language Used

This survey indicates the most popular tools for serverless application developers to monitor the architecture's performance. In fig. 2.18, we can see CloudWatch of Amazon Web Service is the most popular monitoring tool among developers comprising 80% responses. New Relic came second for being used by 14% responders. In the third Data fog with 11% user response. All other monitoring tools like Splunk, IOpipe, Sumo Logic, and AppDynamics cover 5% or less individually. However, Developers use other tools for different purposes that only cover 7% responses.

A prime concern for serverless applications is language support. So, the survey has a question regarding the programming language used by the developers. We find that majority of the respondents that are 75%, use NodeJs as their development language. It is understandable because NodeJs was the first language supported by the serverless platform. In the second position, we find Python with 15% and then Java with 8%. This small pool of language support is one of the impediments to the extensive adaptation of serverless architecture. So, other language support would need to be available soon.

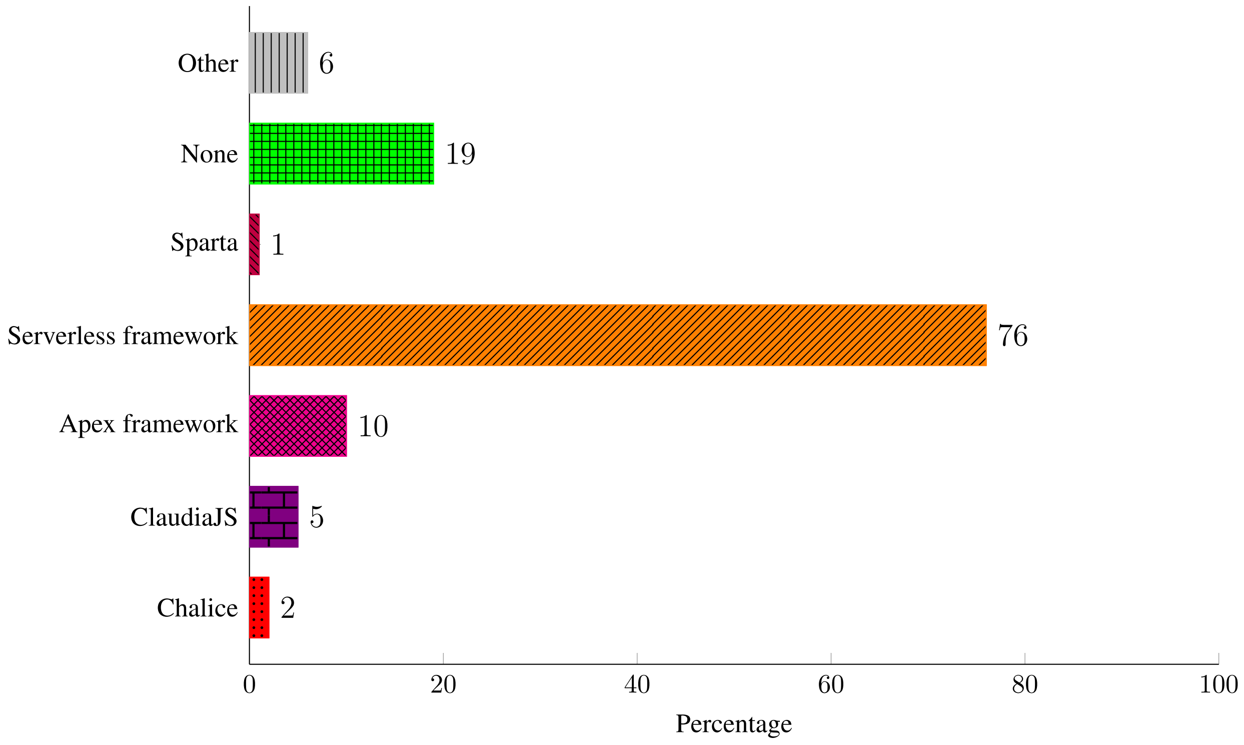

Figure 2.17: Frameworks for serverless architecture used by users

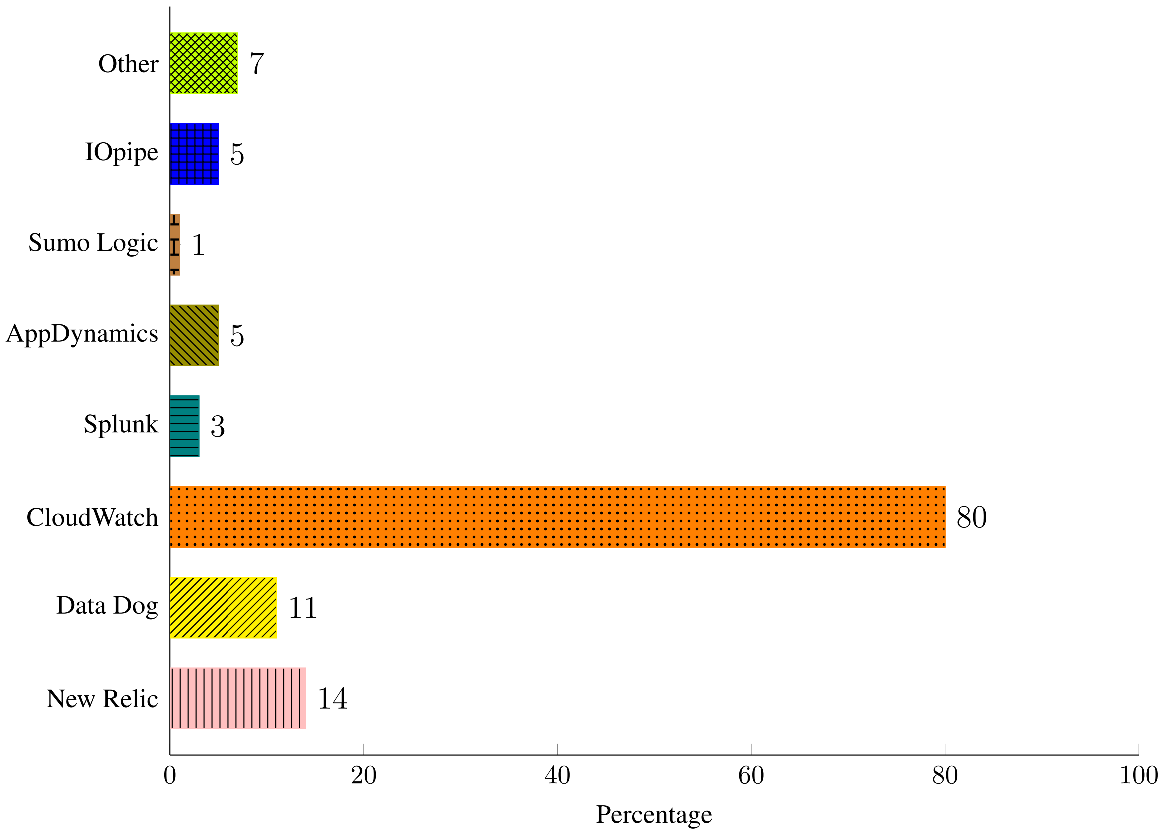

Figure 2.18: Tools for serverless architecture used by users

2.3 Architectural Perspective

Container-based virtualization has now become a cornerstone of cloud platforms and data centers. But the need for runtime-level virtualization to make easily deployable and elastic applications with lightweight bundling opens up a new door in the virtualization concept named lambda model. So, the recent development of OpenLambda has made it possible to take a closer look at many problems and research areas of runtime-level virtualization. The three critical components of the lambda model are 1) the application packaging system, which bundles server runtime with the required library (e.g., execution engine), 2) the memory and server time-sharing between applications, and 3) the registry store, which stores the codes for applications. These codes are called lambda functions. The packaged execution engine forms a sandbox known as workers/containers. Most other platforms keep the worker creation and code execution management out of developers' hands for security and maintenance reasons [124]. The platform providers manage the workers; hence, developers have no control over content management. As a result, the cold start problem remained an issue to be resolved.

2.4 Background Investigation

To improve the performance overheads of these secured serverless platforms, concepts like load balancing and packaging have been proposed [118]. Besides, the built-in load balancer also helps reduce the workers' response times. Pipsqueak [88], a shared packaging tool to reduce the start-up time of cloud functions, has also been introduced. It caches the required packages of functions at each worker in the sleeping state of the function. It has a wake-up and forking mechanism and a cache tree to support multiple dependencies. Nonetheless, its lack of elasticity and redundant cache loading on each worker seems like an overhead for large libraries like Pandas [84]. Function scheduling is also famous for load balancing to reduce response time. As no intelligent decision-making method is available to minimize the number of workers or to reuse the already packaged worker, they find minor usability in real-life applications.

2.5 Current Trends

The current trend is to keep workers warm to avoid the sizeable cold start delay. The AWS lambda service has this feature enabled, keeping some warm workers(around 10) per application in the memory to minimize the latency [124]. These workers are always kept in memory by AWS. These naive workers cause large memory consumption, less re-usability, and non-intelligent lambda function assigning. Even all serverless platforms have different kinds of features. We show the state-wise average timing for a worker of OpenLambda in Fig. 2.19 by making 1000 lambda requests with 20 workers. We find fresh-start and restart - 1000 times and 10000 times

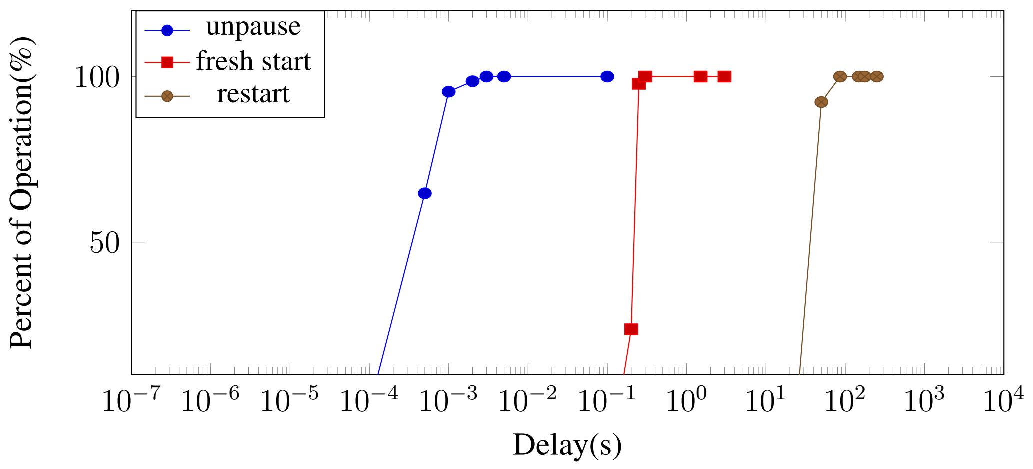

Figure 2.19: Readiness latency in unpause, fresh start, and restart case for OpenLambda

more delay than unpause state. Here, WLEC will try to ensure warm workers for every request, intelligent lambda function assigned to the workers, concurrent request handling, and re-usability of the workers with proper worker lifetime.

Table 2.1: Responsibility in Serverless and serverful platform comparison

Characteristics	**Serverless Platform**	**Serverful Platform**
When the program is run	On event selected by Cloud User	Continuously until explicitly stopped
Programming language	Javascript, Python, Java, C#, Go	Any
Program state	Stateless	Stateful
Maximum memory size	0.125-3 GiB	0.5-1952 GiB
Maximum local storage	0.5 GiB	0-3600 GiB
Maximum run time	900 seconds	None
Minimum accounting unit	0.1 seconds	60 seconds
Price per accounting unit	$0.0000002	$0.0000867 - $.4080000
Operating system and libraries	Cloud provider select	Cloud user select
Server instance	Cloud provider select	Cloud user select
Scaling	Cloud provider responsible	Cloud user responsible
Deployment	Cloud provider responsible	Cloud user responsible
Fault tolerance	Cloud provider responsible	Cloud user responsible
Monitoring	Cloud provider responsible	Cloud user responsible
Logging	Cloud provider responsible	Cloud user responsible

Chapter 3

Related Works

Various efforts to reduce the start-up cost of containers have been proposed recently. As serverless computing is the newest concept, new research has just started. We have divided the related works into six main categories and discussed them in this chapter.

3.1 Platform Based Approaches

In this section, we discuss research works related to performance enhancement and performance analysis. We also mention their impact on other serverless platforms. McGrath *et al.* [83] made an analysis where they showed that different platforms suffer from cold start time problems. They showed Big platforms like Azure [5], OpenWhisks [1], Google Cloud [3], and AWS [2] face cold start around 15 min mark of the container's last invocation. They designed a high-performance focused serverless platform and implemented it in .NET, where they used Windows containers as a function execution environment. However, their main focus was single-function execution. Baldini *et al.* showed the challenges of the serverless platform and defined cold start as a problem that occurs from the scaling to zero feature [20]. Their article surveyed all existing serverless platforms from academia, industry, and open-source projects. Then they identified their key characteristics. Besides, the flexibility of serverless architecture has inspired developers to build their applications on it. From simple image resizing applications in [76] by Liston *et al.* to video encoding applications using thousands of small workers, [44] by Fouladi *et al.* are available in AWS Lambda for benchmarking the platform performance.

Furthermore, AWS has presented a new feature called SnapStart [100] for lambda functions to reduce cold start latency for Java 11 functions. Before starting the invocation, it takes a snapshot of the function and stores it in a multi-tiered cache. The memory and disk state of the function after initialization is stored in the snapshot. Subsequently, the stored snapshot is used to facilitate fast lambda container initialization for following requests. Admittedly, using a previously stored snapshot eliminates *INIT* phase and moves to the next phase *INVOKE*. Thus, resulting in 10x faster startup performance than before for Java environments. In addition, a major advantage of SnapStart compared to provisioned concurrency is that it is free. Nonetheless, SnapStart is only applicable for fully qualified lambda ARNs. Another pitfall of SnapStart is that function initialized with a unique value (*e.g.* unique ID, unique secrets), network connections, and temporary data require total overhauling to be used through SnapStart. Further, SnapStart does not come with any fault tolerance scheme as well. Again, the degree of concurrency supported by SnapStart remains a contentious matter and it is exclusively designed for AWS lambda architecture.

Here we mention some of the most recent related works and endeavors designed to provide a clear view to delineate the impact of colds start in the performance of serverless platforms. Researchers illustrate the root cause and the effect of cold start in various platforms in studies such as [20], [83]. Authors described a container management scheme named Pagurus that re-purposes a warm container in [73]. Zijun *et al.* proposed a lightweight, secure container runtime called RunD [72] to support high-density deployment and high concurrency startup. Sashko *et al.* introduced a spawn start mechanism to analyze performance issues across three FaaS providers [97]. Ping *et al.* suggested leveraging a pool of function instances to mitigate cold start latency [75]. Paulo *et al.* presented a strategy named Checkpoint/Restore In Userspace (CRIU) that reduces function startup time utilizing function instance cloning [104]. Prediction based strategies like Adaptive Warm-Up Strategy (AWUS) [132], time series forecasting [56], retention-aware container caching [91] and scheduling based strategies such as OWL [116], FaaSRank [134] have showed substantial potential. Baird *et al.* [19] showed the performance difference between the AWS platform's warm and cold containers. The article also provided a detailed overview of lambda handlers and event-based lambda invocations in the AWS platform.

Malishev [80] compared different languages like Golang, Python, Java, .Net, and NodeJs in cold start timing. He also tested with different memory sizes like 128MB, 1024MB, and 3008MB of a container. McAnlis [82] tried to reduce cold start by trimming the dependencies. Yet it proved to be very language-dependent. Another approach was dependency caching, which also suffers from the same problem and non-scalability. He also described a lazy loading concept that can be implemented in a lambda function to reduce the cold start time. Cordova [32] introduced a warming-up function and delay mechanism. But warming up all lambda functions costs a large memory and is non-scalable. He also proposed allocating more memory for the lambda containers, but memory allocation is costly in the serverless platform, making the idea impractical for large applications.

3.2 Architecture Based Approaches

Oakes *et al.* proposed SOCK [89], an optimized lambda container system with modified Zygote [61] provisioning along with a three-tier caching mechanism to counter kernel overhead.

Authors use Linux *cgroups* primitives to avoid container bottlenecks, with results in 18X speedup over Docker. SOCK uses lean containers to avoid the expensive operations required to create general containers, which leverages a three-step lambda container creation model. The three-tier caching system comprises of (1)*handler cache*, which maintains idle handler containers in the form of pause containers without consuming any CPU resources, (2)*install cache* includes a pre-installed large set of static packages on disk to isolate them from handlers in case of malicious content within the packages, and (3)*import cache* manages Zygote containers and selectively activate them based on the import cache policy to reduce CPU memory consumption. But the implementation was bounded by only the python package caching, specifically PiPy repository [31]. Another concern is that Zygote provisioning often runs into compatibility issues in the case of concurrent cases.
Chen *et al.* proposed BIG-C [29], a container-based preemptive task scheduling for clusters with heterogeneous workloads. Immediate and Graceful, two types of preemption, were proposed to handle task scheduling for short and long jobs in a shared cluster. Authors implement their design with the task preemption technique of Hadoop YARN and its resources. The architecture of BIG-C consists of a resource monitor (RMon), a preemptive fair scheduler at the resource manager of each cluster, and a container allocator and a container monitor at each node manager of the machine. However, the reclamation of memory of preempted tasks and delay caused by memory restoration becomes a major overhead for BIG-C. The authors argue that such overhead is avoided most of the tune due to graceful preemption. But considerable performance degradation is inevitable if graceful preemption fails to satisfy short job demands.
For VM-level cold boot, we find LightVM [81], a lightweight virtualization technique with a lower boot time than a Docker-based container and constant boot time no matter how many VMs are already running. It was achieved using unikernels for specialized applications and with Tinyx, a tool that enables the creation of tailor-made, trimmed-down Linux virtual machines. Besides, LightVM is based on optimized Xen to offer fast boot times. A complete redesign of Xen's control plane, transforming the centralized operation to a distributed one so that minimal interactions with the hypervisor make LightVM a completely new option for server administrators. However, issues like portability, complexity, generality, and the need for significant engineering efforts have made LightVM less inspiring to replace ubiquitous containers.
Another recent approach was proposed in [85], where authors try to reduce the cold start time utilizing the pause containers. Steps include pre-creating networks and connecting them to function containers so that the delay involving network creation and establishment of connection can be removed from the critical step of the startup. The network pre-creation tasks follow Kubernetes and pause containers concept. These pre-created network containers are called pause containers. They are then kept in a pool of similar containers and maintained by *pause container pool manager* (PCPM). The pool management is done by following three key steps. (1) *Build phase* is responsible for initialization and setup tasks of the FaaS framework. Several pause

containers are launched, and a free pool is created by PCPM with associated identifiers of the pause containers. (2) In *Execution phase*, the invoker queries the pool manager and obtains the identifier, which is then used to attach the new container to the associated pause container. Later, the pause container gets removed from the free pool. (3) Finally, during *Completion phase*, the function execution finishes, and the corresponding container is recycled or terminated. The invoker informs the pool manager and reclaims the pause container resource. Nonetheless, a critical drawback of PCPM architecture is that it only considers network-related delays for cold start problems. Additionally, Wang *et al.* designed a middleware system named FAASNET [123] for FaaS workloads to alleviate container provisioning utilizing an adaptive function tree structure. Besides, optimizing resource allocation and configuration are also explored in [128] and [43], respectively.

3.3 Application Based Approaches

Cui [34] proved that the cold start improvement techniques of lambda functions are more effective in the case of python and node js because of the pip and npm package management tools, respectively. He showed that java and C# suffer 100 times more start time than python. He also mentioned that the code and memory size linearly increase cold start time. Besides author says idle timeout for lambda is not constant. For AWS, it depends on a specific region's demand and supply of resources. He added higher memory(RAM) tends to take more time to stay warm, and deployment package size does not affect cold start time.
Approaches like function scheduling [7], package caching, and load balancing were also explored to improve response time for new concurrent requests. [35] showed how concurrency can affect the cold start and showed the huge impact on the response time for lambda functions. Daly *et al.* created Lambda Warmer [36], a lightweight module that uses the "ping" method of CloudWatch Event to keep specific lambda functions warm.
Lakhsman [42] showed a way where he warms up all the functions every 20 min to make them warm and calls it a hack to reduce the start-up latency. He argues that using a single wake-up function is better than using a different wake-up lambda for each function. He shows why warm-up after every 20 minutes is a better choice for lambda users, and a cost overview is discussed that will still be within the budget. Finally, he emphasizes that AWS Lambda SDK is the best choice for war-up tasks as it reduces invocation time and keeps uniform lambda with different event sources.
Serhat [28] described the pros and cons of cold start and elaborated on the reason behind them. He emphasizes that it is specifically essential when there are sync applications, low request volume, use of statically typed language, or deployment of a new version of the application. He mentions how it can cause frustration among users because of the lack of controls on the user end for managing those containers. He elaborated that a cold start may cause timeouts to the

calling functions as a chain reaction. He mentions major factors that increase cold starts, like code size, virtual private cloud, HTTP calls, the requirement of the classpath, memory size, and language choice. He recommends ways out, like increasing memory, warming up, etc.
Kotlin *et al.* introduced a parallel execution mechanism for FaaS workloads utilizing Intel Memory Protection Keys to reduce function interaction latency [64]. Ashraf *et al.* introduced SONIC, a data passing management tool that optimized data passing [77]. Jie *et al.* presented a serverless platform named TETRIS [70] to ensure lower memory consumption through combined batching and concurrent execution for deep learning services with tensor redundancy. Ashraf *et al.* designed ORION-an optimization technique for serverless DAGs to reduce End-to-End (E2E) latency [78].
However, none of the previous works provides a platform and language-independent serverless architecture to simultaneously ensure reduced cold start delay and high concurrency support.

3.4 From Stateless to Stateful Serverless

Vikram *et al.* developed a stateful FaaS platform named Cloudburst that uses a data cache to improve storage access latency [109]. Furthermore, it provides a centralized key-value store for functions to share states and data to facilitate interdependent function execution. Zhipeng *et al.* proposed Boki that leveraged a shared log API to provide durability, consistency, and fault tolerance for serverless function [57]. Martijn *et al.* , using Apache Flink StateFun, builds a novel programming model which enables serverless functions to perform stateful transaction operation [37]. Minchen *et al.* argued that serverless functions should follow a data-centric approach to couple function flow and data flow tightly. Further, the authors proposed a novel serverless platform named Pheromone [135] that ensures high usability, broad applicability, and latency reduction between function invocation and data exchange. Peter *et al.* developed a Database Management System (DBMS) backed transactional FaaS framework named Apiary [65] with scheduling and tracing layers to ensure efficient clustering and truncating communication overhead. Jonas *et al.* developed a programming model named PORTALS [107] for stateful serverless systems to ensure atomic streams and atomic processing contracts. Daniel *et al.* proposed CRUCIAL-a system to facilitate high parallel execution for stateful serverless applications. Simon *et al.* described a performance-enhancing runtime name FAASM [103] for stateful serverless applications to ensure resource fairness, memory safety, and in-memory state sharing. Anurag *et al.* suggested a far-memory system named Jiffy that stores ephemeral states to match resource demand for stateful serverless applications [60]. Adil *et al.* proposed a high-level programming model to ease developers' burden for creating stateful serverless functions and deploying them in the cloud [8]. Martijn *et al.* discussed transaction orchestration using a proposed programming model and implementation to guarantee exactly-once processing [39]. Claudio *et al.* provided three solutions for reducing performance bottlenecks during data transfer for edge networks [30]. We also find recent works for reducing performing bottlenecks and performance enhancement strategies such as [135], [22], [103], [60] and [74].

3.5 Rank Correlation Based Works

In [133], Emine *et al.* proposed a new average precision-based rank correlation coefficient called AP correlation with a probabilistic interpretation. It acts similarly to Kendall's correlation if the errors are distributed uniformly. However, it provides a lower value than Kendall's if the top/bottom of the list has more errors. Comparison of the effectiveness among various ranking correlation coefficients was shown by Trastion [112]. Sathya *et al.* proposed a Spearman's rank correlation coefficient based on practical web content mining with minimum irrelevant and redundant data, also known as outliers [21]. Anca *et al.* proposed a new ranking correlation coefficient name ClasSi that deals with class label rankings. It also introduces a new distance function to calculate the similarity among classes [54]. Hauke *et al.* provided a comparison study between Pearson's and Spearman's correlation coefficient on the same set of data [49]. We can find recent applications of Spearman's correlation in correlation studies with COVID-19 and weather [117], dementia [18], meteorological parameters [66].

3.6 Fault Tolerant System Focused Works

In [94], Brian *et al.* discussed various aspects that a system needs to have to be considered as a fault-tolerant system. Improvement on the traditional lineage-based failure recovery model was presented by Stephanie *et al.* by introducing the Lineage Stash model [125]. Diyu *et al.* introduced HyCoR, a fault-tolerant mechanism for containers based on container replication utilizing a hybrid of traditional checkpointing and replay methods [138]. Vikram *et al.* presented a fault-tolerant system called Atomic Fault Tolerant (AFT) shim that can guarantee atomic read and write operation [108]. Pascal *et al.* proposed a benchmark tool for serverless platforms to identify their performance in terms of fault tolerance and other aspects [79]. Heus *et al.* introduced a new programming model to handle stateful distributed transactions with serverless architecture and showed that it performed better than traditional two-phase commit transaction architecture [38]. A hybrid fault tolerance technique named Threshold-Based Adaptive Fault Tolerance(TBAFT) was introduced by Ajay *et al.* [96]. TBAFT is developed by combining proactive and reactive strategies to ensure high throughput, decreased migration, and execution time overhead. Yilei *et al.* suggested a byzantine fault tolerance framework called Byzantine Fault Tolerant Cloud (BFT Cloud) [137] that guarantees robustness in case of crashes and arbitrary behavior faults. Zhang *et al.* proposed a library named Beldi to write fault-tolerant transactional stateful serverless functions. Authors assert its effectiveness and affordability by implementing it with three different applications [136]. Wubin *et al.* provided a high availability-based fault tolerance study by comparing availability between a container and virtual machines [71]. Yasmin *et al.* showed a comparison of the performance of various FaaS platforms like Kubeless, Fission, and OpenFaaS concerning fault tolerance, energy consumption, and extensibility [24]. Siyuan *et al.* pointed out a communication framework named Hoplite [140] to ensure fault tolerance for task-based distributed systems based on efficient dynamic data transfer scheduling. We can find a comprehensive overview of fault-tolerant issues regarding cloud computing in [94], [10], [67], [71]. Other noteworthy fault-tolerant tools consist of a scalable data store called UNISTORE [25], FaaSdom [79], and Olive [101].

Chapter 4

Preliminaries

In this chapter, we would like to discuss some of the concepts that would be important in understanding the thesis. Definitions are also provided for terminologies that we will use later in this work. Definitions not mentioned in this chapter have been discussed when they are mentioned in other chapters. Here at first, in section 4.1, we discuss Pannier, which significantly influences our work. Then in section 4.2, we elaborate on the OpenLambda platform and its major components. Next, we mention the Thundra platform and its utilization in our work in section 4.5. After that, we define cold start in section 4.6. In our last section 4.7, we explain various contemporary cache replacement policies used in-memory cache management in ubiquitous computing systems.

4.1 Pannier

Pannier is a caching system proposed by Cheng *et al.* in 2015 [69]. It is for flash devices and is designed after container-based systems. The unique feature that it provides besides caching is the identification of divergent containers. Divergent containers have blocks with varying access patterns. A survival queue based on the survival time is used to rank the containers, a priority queue. The survival time denotes the period that the container remains alive. A container is alive if it has live data or data currently in use and validated by the system. Pannier uses one of the most common cache policies, the Least Recently Used (LRU) policy. Yet to make it effective, Pannier takes it one step further. Instead of basic LRU, they use two segments LRU called S2LRU. Not only that, it further modifies it to become container aware, transforming it to S2LRU+. So it can handle the containers fluently with the same queue model. S2LRU introduces hot and cold data from the two segments to segregate them. Another essential feature is the TIRE system, a multi-step feedback controller to throttle flash writes. It also extends the lifespan of the flash device's read writes by a significant margin. Fig. 4.1 describes the architecture of Pannier. The three main techniques of Pannier are:

- Leveraging block access counts to manage cache containers.
- Incorporating block aliveness as a property to improve flash cache space efficiency.
- Designing a multi-step feedback controller to ensure the flash cache does not wear out in its lifespan while maintaining performance.

Pannier also maintains a global clock named "Wall Clock" to keep track of the insertion operations in the cache read/write, which may cause invalidation. The clock value later assigns the decay point and survival time. An invalidation bitmap is used, which keeps a bit for each block in the cache and can keep track of the valid or invalid caches. So, a simple mechanism is if any cache block gets invalidated, the corresponding access bit is cleared in the bitmap. Pannier also uses a concept called "ghost cache", which stores the metadata, manages the counter for any block, especially for the evicted ones. As it only stores metadata, it can even store more than the physical cache size, which reduces the tracking overheads. Though block-based caching provides superior per-block tracking to container-based caching, Pannier proposes new mechanics to reduce the tracking overhead by combining and relocating the blocks.

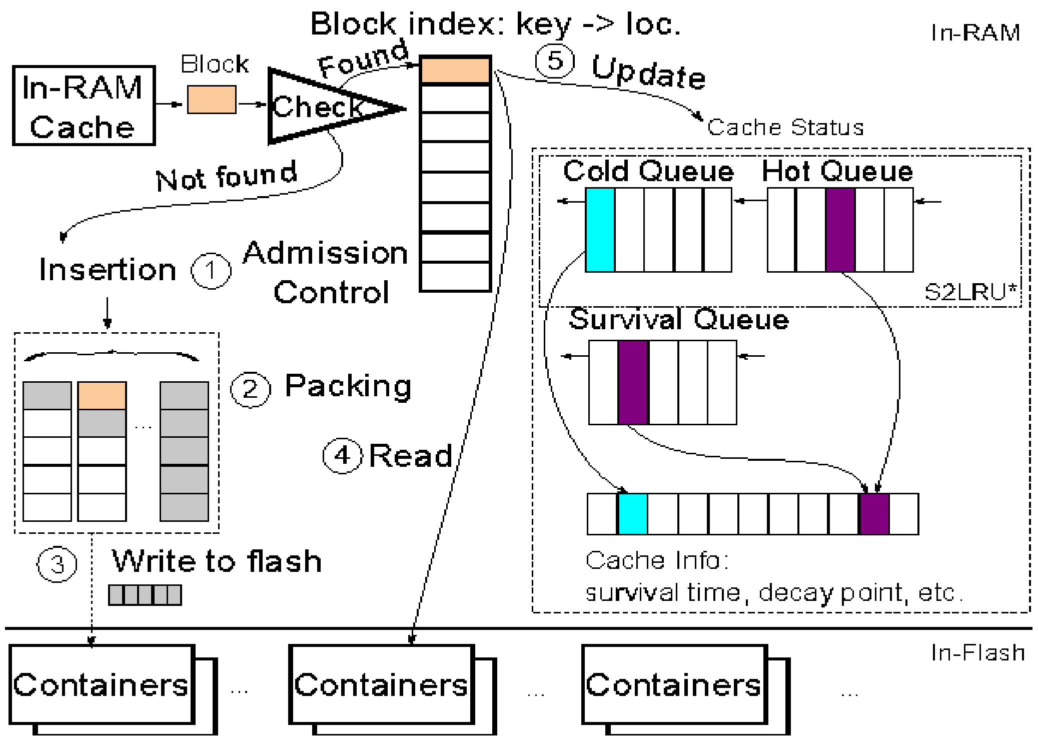

Figure 4.1: Container based flash caching [69]

4.1.1 S2LRU*

Pannier has a unique cache mechanism system that takes advantage of the contemporary two-segmented cache management model. This traditional model is called S2LRU. A significant advantage of this data segregation is to protect the second-level cache locality. The fundamental basis of Pannier is S2LRU. However, it needs to be container-aware to make it adaptive and intuitive for Pannier. The authors proposed that S2LRU can be container-aware by managing the container granularity instead of the block granularity named S2LRU+. To do that, S2LRU+ inserts a new container into the MRU position in the cold queue. Whenever a hit occurs on a container, the container is promoted to the hot queue. The migration from the hot and cold queues is also at container granularity. Container information structures are maintained through the reference of S2LRU+ entries. However, S2LRU+ is further modified to suit Pannier by implementing modified insertion, promotion, and reinsertion operations to support divergent containers and segregation of hot and cold blocks. A structure of S2LRU* is shown in fig. 4.2.

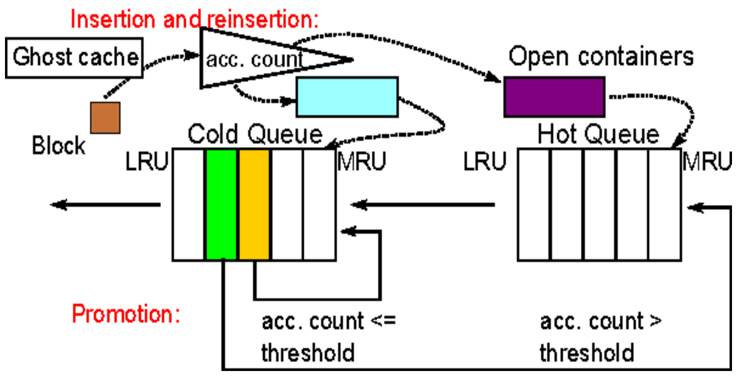

Figure 4.2: S2LRU* in Pannier [69]

4.1.2 Survival Queue

One of Pannier's most complex challenges was managing the divergent containers in the cache. Without proper identification, these containers could survive for a long time because of the small number of repeatedly accessed blocks. Pannier introduces a survival queue with its S2LRU* structure to manage such divergent containers. As they get identified by the system, they get copied to form a new container to segregate them from hot and cold data. They resulted in the free of the original container. The authors mentioned two new parameters named *decay point* and *survival time* for each container to describe when to inspect a container. These parameters also define how long a container can stay in the cache. The survival queue is ranked using survival time. It also points to container information structures shared with S2LRU* structures. The wall clock was used to determine each container's decay point and survival time.

4.1.3 Invalidation and Access Bitmaps

To equip Pannier with better block-based validation, Pannier uses an invalidation bitmap. This bitmap tracks the blocks of each container. A bit is reserved for tracking whether that block is validated or invalidated.

Again, an access bitmap is introduced to keep track of the access footprint. Similarly, a bit is reserved for defining if that corresponding block has been accessed since it was written to flash. If a block gets invalidated, the corresponding access bit is cleared from the bitmap.

4.1.4 Ghost Cache and Access Counter

An 8-bit counter was maintained to keep track of the block access. This counter is also called the access counter. A counter for blocks of Pannier in the cache and those recently evicted from the cache was used using a particular type called ghost cache. These caches were configured to track more than the actual physical cache size to keep Pannier more effective and clean. It was made possible by the intelligent choice of keeping only the key and access counter in the ghost cache. This key and access counter is referred to as metadata for the cache. Researchers also tweaked it with different techniques to reduce tracking overheads. The ghost cache was set to hold two times the actual cache size. It was later used for insertion and reinsertion purposes of Pannier.

4.1.5 TIRE System

TIRE stands for Throttle Insertions and Reinsertions for Erasures. TIRE manages lifespan-related tasks and plays a vital role in implementing Pannier. Flash devices have a respective usable lifespan of 2-10K erase cycles, and the admission control component ensures that the flash cache does not wear out. Maintaining the flash erasure quota is the principle of the TIRE admission policy. TIRE uses a credit-based approach to manage the quota. The execution time was split into quanta as an accounting period, and an eraser quota was set for a specific period.

4.2 OpenLambda

As the lambda model provides far more elasticity and scalability for container-based applications, it has become essential to have an open-source lambda service to facilitate low-cost research in this field [86]. OpenLambda is an Apache-licensed serverless computing project written in Go and based on Linux containers. This new paradigm opens interesting challenges for execution engines, databases, schedulers, and other systems. Besides, the LambdaBench comes as a bonus tool with the OpenLambda package. LambdaBench is a new benchmark suite for applications in the lambda programming model. The project provides a complete set of tools to make it easier for later research attempts in this platform to evaluate new designs, models, and implementations in subsystem levels within the platform. OpenLambda consists of 4 subsystems which coordinate with each other to run the lambda handler. The components are as follows:

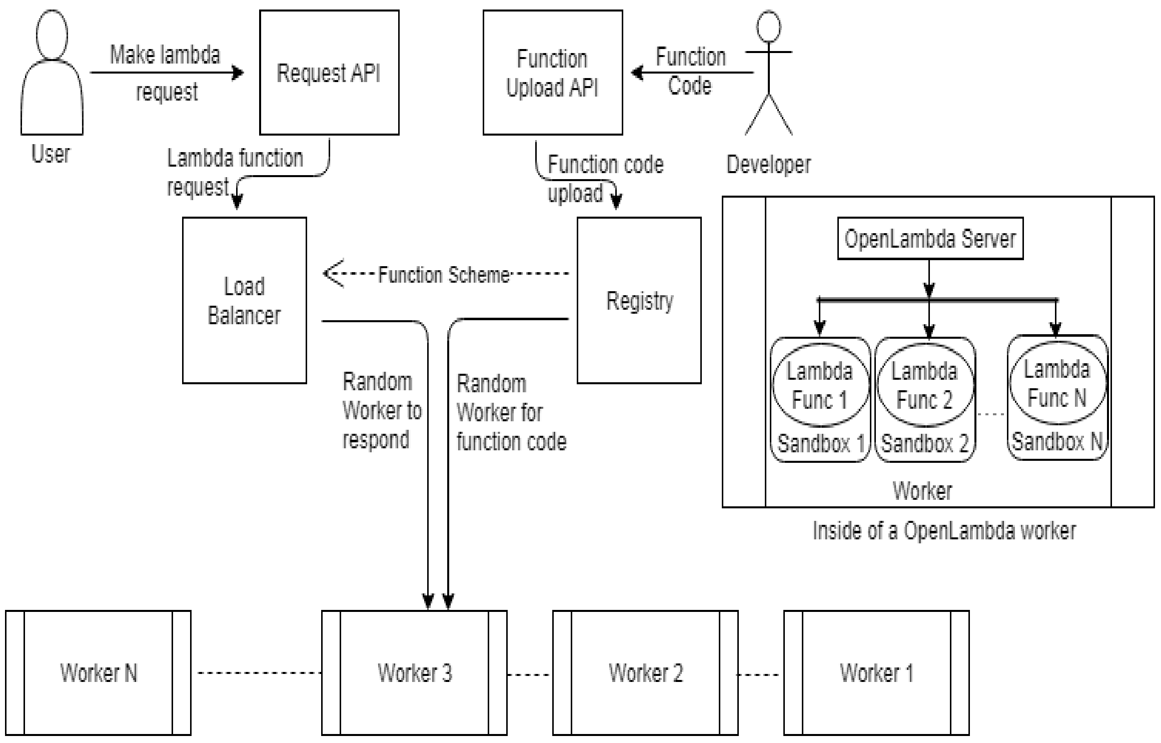

Figure 4.3: OpenLambda architecture

4.2.1 Lambda Store

In a serverless container-based environment, someone must host and distribute handler code across different lambda containers according to their demand. To respond to lambda requests, it has to be robust and fast. The Lambda store provides this facility to OpenLambda. It makes the handler codes more effective and easy for developers. As most lambdas are written in interpreted languages, the store has to provide the corresponding compilers for languages like Java and JavaScript. Just-in-time (JIT) compilers are suitable for python to provide dynamic profiling for performance improvement. Handling all these things in each lambda function may become costly. So the store should provide central support for any serverless platform. The store should support most of the packages so developers can use them in the lambda invocations. If the package is unavailable at the store, it has to collect via the internet and then combined with running the invocations that need those packages. But a package-aware platform would be able to handle these packaging issues behind the scenes, affecting very little performance. Some popular packages may include npm, pip, etc. As these packages may become very large, maintaining them for lambda would be a bottleneck. Managing and resolving the package may reduce the start-up latency for other platforms.

4.2.2 Local Execution Engine

A sandbox for executing handlers is the heart of the Lambda architecture. AWS Lambda uses containers to sandbox handlers [93] but avoids the overheads of Elastic BS [13] and other container-based services by sharing servers and runtime between different instances. The cold start depends on the architecture of the execution engine heavily. AWS reuses the containers to minimize start-up time and ensure the efficient use of resources. In the case of OpenLambda, it uses the same reuse mechanics [121] along with multiple handlers for the same container. Unfortunately, even with this optimization, Lambdas are significantly slower than containers at low request volumes.

4.2.3 Load Balancer

OpenLambda intends to minimize the load balancing time, ensuring the localities of different aspects of the system. Previous cluster schedulers like Sparrow [90] took 100ms for balancing, whereas OpenLambda took only 10ms [119]. The load balancing mechanism highly relies on the locality management of the system. The localities considered in this case were session locality, code locality, and data locality.

4.2.4 Lambdaware Database

The support of user-defined functions [130] in the databases creates a great way for lambda functions to be used on cloud-based databases. For architectural purposes, databases need to be distributed. As S3 [12] supports DynamoDB [40], OpenLambda supports databases like RethinkDB [122], CouchDB [14] and DynamoDB. Relaxed consistency models were also evaluated in the context of RPC handlers for OpenLambda. An application may require that all RPC calls from the same client have a read-after-write guarantee, but weaker guarantees may be acceptable between different clients, even when those clients read from the same entity group. So, the creation of better compute models for lambda functions depends on which actor accesses that function.

4.3 Rank Correlation and Rank Correlation Coefficient

Rank correlation is a common way nowadays that is utilized to perform information retrieval and estimation tasks. Various ranking correlation measurement methodology has been proposed by researchers to quantitatively or qualitatively measure the association among ordinal variables. Hence, various ranking correlation coefficients are developed to handle different and exceptional cases with non-linear, non-deterministic, non-parametric statistical data.

4.3.1 Rank Correlation

A rank correlation is an ordinal measurement of association among variables and indicates the relationship among the ranking of such ordinal variables. Research in the field of value estimation and information retrieval often uses a ranking list of variables to predict unknown values. We can divide the correlation methods into three main categories.

- **Unweighted Rank Correlation** This is the most basic form of correlation among variables. All the variables are considered equal in weight. Examples of unweighted rank correlation models are Spearman rank correlation, Kendall rank correlation etc.
- **Correlation of Scores** In this method, a scoring system is used to make the correlation among ordered categorical data. This method assigns the first arbitrary equal-interval scores to the ordinal level. Then classical statistical methods are applied based on the scores.
- **Weighted Rank Correlation** The assumption of linear or non-linear relationships among variables can sometimes be impossible to understand truly. In such cases, weight is introduced among the variables. Even most of the correlation measurement incorporates an implicit weighting scheme. Similarly, the robustness and flexibility of the weight rank correlation method make it more suitable than any other method. Example includes Weighted Spearman rank correlation, Weighted Kendall rank correlation etc.

4.3.2 Rank Correlation Coefficient

A rank correlation coefficient calculates the degree of existing similarity between ranked variables and assesses the significance of the relationship among the variables. These coefficients are calculated differently by different rank correlation methods. Ex: Spearman's Rank Correlation Coefficient(ρ), Kendall rank correlation coefficient(τ). Each method has its rank correlation coefficient and is used based on its use cases. But all the coefficient determined how significant the relation among variables are. For compatibility and better understanding, the coefficients are usually constructed to vary between -1 to +1. The higher magnitude (+1/-1) value shows that the variables have a positive relationship and maintain a strong association. But a negative value means the variables maintain strong association but in the opposite order. A value of 0 indicates that variables do not have any relation, but that does not necessarily indicate that variables are independent.

4.3.3 Spearman's Rank Correlation Coefficient

The Spearman's rank correlation coefficient method identifies the strength and direction (positive or negative) of the correlation between two variables [106]. In statistics, Spearman's correlation coefficient is denoted by the Greek letter ρ. It is a non-parametric measure of rank correlation and statistical dependence between two variables. It assesses the association between two variables that can be described with a monotonic function. It is appropriate for both continuous and discrete ordinal variables.

Spearman's correlation is similar to the Pearson correlation, which also involves the rank values between two variables. The notable difference, however, is while Pearson's correlation assesses the linear relationship, Spearman's correlation assesses the monotonic relationship. If there are no repeated values, a perfect Spearman's correlation will have a value between -1 to +1 if each variable is a perfect monotone function of the other. It is observed that the correlation between two variables will be high when variables have a similar rank and low when variables have dissimilar values [27].

4.3.4 Mathematical Formulation of ρ

Let x_1, x_2,...,x_n and y_1, y_2,...,y_n be two samples of size n. R_{x_i} denotes the rank of x_1 compare to the other values of the x sample, for $i = 1, 2, ..., n$. $R_{x_i} = 1$ if x_i is the largest value of x, $R_{x_i} = 2$ if x_i is the second largest value of x etc., until $R_{x_i} = n$ if x_i is the smallest value of x. In the same way, R_{y_i} denotes the rank of y_i. for $i = 1, 2, ..., n$. The Spearman's rank correlation denoted as ρ , is defined by:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)} \tag{4.1}$$

where $d_i = R_{x_i} - R_{y_i}$.

If several observations have the same value, those observations will be given an average rank. However, if there are many average ranks, it is best to correct to calculate the coefficient with the following equation:

$$\rho = \frac{S_x + S_y - \sum_{i=1}^{n} d_i^2}{2\sqrt{S_x . S_y}} \tag{4.2}$$

where

$$S_x = \frac{n(n^2-1) - \sum_{i=1}^{g}(t_i^3 - t_i)}{12} \tag{4.3}$$

with g the number of groups with average ranks and t_i the size of group i for the x sample, and

$$S_y = \frac{n(n^2-1) - \sum_{j=1}^{h}(t_j^3 - t_j)}{12} \tag{4.4}$$

with h the number of groups with average ranks and t_j the size of group j for the y sample.

However, if there are no average ranks, the observations are seen as groups of size 1, meaning that $g = h = n$ and $t_i = t_j$, for $i, j = 1, 2, ..., n.$ and

$$S_x = S_y = \frac{n(n^2 - 1)}{12} \tag{4.5}$$

4.4 Fault-Tolerant System

The advent of new hacking methodologies and an increased number of security-unaware naive users have made system exploitation a common problem for application designers. At the same time, the fourth industrial revolution (4IR) inspired the adoption of inter-connected systems with the help of the internet. Thus, hackers and exploiters are finding new ways to compromise systems and achieve personal gain through numerous attack strategies. Despite the advent of new technologies to withstand such attacks, the systems still seem vulnerable to attacks like service interruption and Denial of Service (DoS) attacks. Consequently, the need for automatic fault-tolerant and mitigation systems is increasing daily. Various architectures and schemes are proposed to take on the challenge of unique scenarios.

4.4.1 Goal of Fault-Tolerant System

A fault-tolerant system needs to achieve dependability to be considered a fault-tolerant system. Yet, the "Dependable system" must have certain criteria; otherwise, the term seems ambiguous. So, Abdeldjalil *et al.* defines four definite aspects that should be fulfilled by any fault-tolerant systems [68]. The aspects are shown in Fig. 4.4 as follows:

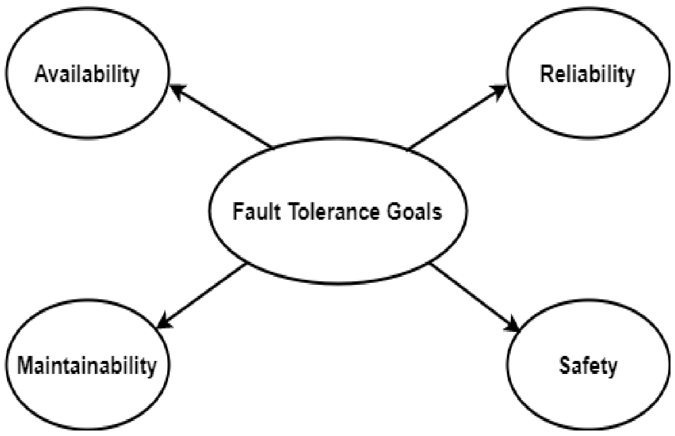

Figure 4.4: Fault tolerance goals

Availability

This indicates that an ideal fault-tolerant system would be available 100% of the time. Thus, the system is never down and always ready to be used.

Reliability

Reliability presents that Users can trust its outcomes all the time and without any doubt. Hence, this system can be relied upon in every critical situation and never fails.

Maintainability

System is efficiently and promptly maintainable. It shows that the system can mitigate any failure at a moment's notice and without service disruption. As a result, it ensures seamless service continuation for the users.

Safety

Users can use the system without the risk of personal data leaks, data loss, or privacy breaches. It guarantees that users' data is protected and safe even in catastrophic situations.

4.4.2 Reason Behind Faults

Though the reason for a system failure can not be easily categorized, thousands of issues can lead to system failure. It can even be accidental or intentional. However, system-level faults can be mainly divided into two main components: hardware failure and system Bugs.

Hardware Failure

Hardware-level failure contributes to most reported failure incidents. Connection disruption, expired device components, power outages, natural disasters, server failure, temperature, and environmental hazards can lead to hardware failure. Sometimes in consideration of cost and maintenance, these cases are unavoidable. However, redundancy is the most practical solution to hardware failure.

Software Bugs

Software bugs refer to errors in the design or development face of software that lead to abrupt system failure. About 60-90% of system failures are caused by buggy software. Lack of testing in the development phase, proper deployment and configuration, lack of specification, and shortage of full-proof tools lead to software bugs. However, most of these faults are transitory and can be easily mitigated by good version controlling and other fault-tolerant techniques. According to [10], software bugs can be divided into two main categories:

- **Request Stage Failures:** This occurs when a request fails without acquiring application resources. Requests do not access the resources and do not hold any resources. So, these failures are mostly seen in primary and new systems. They are less costly and easy to fix. For instance, overflow and timeout are the most common request stage failures.
- **Execution Stage Failures:** Execution failures point to failure during the request processing stage. Unlike request stage failures, execution failures occupy resources. Hence, many failures may become costly and detrimental to the system. These are complex, and failure identification and remediation may become unfeasible and unavoidable. Sometimes, remediation requires a complete system overhaul. Resource missing, database failure, hardware failure, and resource shortage are typical examples of execution stage failure.

4.4.3 Components of Fault Tolerance

In computer systems, three components play a vital role in creating a fault-tolerant system. These components and their implications are considered in terms of the maintenance and operating cost, quality of service, and fault detection capabilities. The components are as follows:

Redundancy

Redundancy refers to having multiple components rather than only a single component to do a specific job. It removes the single point of failure. If the main component faces any issue, the redundant component will take the job and resume the operation. It provides time for the fixation of principal components and eliminates any chance of business interruption.

Diversity

Diversity means having different types of components responsible for the same job. Diversity protects from the built-in faults within a system. Diversity ensures fault tolerance by introducing components from various sources and software with different designs.

Replication

Replication indicates having a backup copy of a system. It mainly utilizes the backup copy to restore any system in times of failure. Replication is a more complex process than any other two. It involves multiple copies of systems and subsystems to provide the same results. We can also use replication to identify the faulty component.

4.4.4 Common Techniques of Fault Tolerance

All systems generally introduce some fault tolerance mechanisms in case of execution failures. Similarly, cloud systems are not an exception. Based on the criticality of the system, different strategies are employed. Though various techniques for fault tolerance in cloud computing (FTCC) have been proposed since the emergence of cloud computing, these can be categorized into two main groups. They are:

Reactive Fault Tolerance

Reactive fault tolerance (RFT) techniques are the most popular among developers. RFT techniques focus on the recovery process and sustain the failure. They are generally coupled with robust data loss prevention and recovery methods to avoid any data loss. They do not take any preventive measures but focus on the fast recovery of the system. So, RFT techniques are passive. Some most widely utilized RFT strategies are as follows:

- **Retry:** Retry is the most common technique that can be found in all systems to some degree. It indicates repeating the same operation in case of failure. However, a predetermined number of retries is set before reporting a timeout. This technique is incredibly convenient for transient faults like network connectivity failure, resource shortage, temporary unavailability of service, and components. A significant advantage of the Retry technique is that it does not necessarily cost additional resources compared to others. Consequently, making it particularly appealing for naive and low-budget systems. Nevertheless, it may incur undesirable situations, for instance, ambiguity, pointless redundancy, and race conditions.

- **Record & Replay:** It refers to the mechanism for maintaining a secondary copy for the primary entity. The primary entity sends instructions and data over a dedicated link to the secondary copy. Then the secondary copy replicates all the execution steps by replaying the primary entity. However, for complex function chaining and multi-threading, the non-determinant nature of this strategy often suffers from the lack of synchronization of the replica. However, it is considered the easiest and most reliable way for small and simple applications. In times of failure, the secondary entity replaces the primary entity to provide seamless service continuation. For instance, James *et al.* proposed HQ replication [33] that achieves fault tolerance against byzantine faults.

- **Checkpointing:** It refers to the ability to store different stages of a computation in a log and replay the computation later using the log to provide the same output without changing the behaviour of the computation. The logged stages are called checkpoints, and replaying the computation is called a restart. It is one of the later high-availability strategies suitable for applications with a large computation cycle and various complex stages. It is less costly and requires fewer resources as all the data are not copied or replicated continuously like *Record and Replay* strategy. However, cases like data races and incomplete checkpointing can lead to the failure of the restart process. The Non-blocking Coordinated Checkpointing Algorithms [111] developed by Surender *et al.* can be brought up as an example.

- **Migration:** Migration indicates replacing one resource with another resource. It can replace physical resources like servers or logical resources like containers and VMs. Migration provides physical and logical separation of failed requests and a new environment for request execution. Various efficient and intuitive migration techniques have been proposed previously. However, it suffers from migration latency, cost overhead and security risks. Yet it facilitates easy request management and load balancing. For example, a power and cost-aware migration technique pMapper [120] was proposed by Akshat *et al.*

- **User Defined:** User-defined fault tolerance can also be known as exception handling-based fault tolerance. Developers write specific code to avoid a system failure in case of any unexpected condition reached during code execution. Exception handling is a naive approach to achieving fault tolerance since it is impossible to anticipate all such unexpected occasions. However, developers usually handle fundamental cases like network errors, resource shortages etc. It is especially effective to avoid system crashes and maintain parallel request processing.

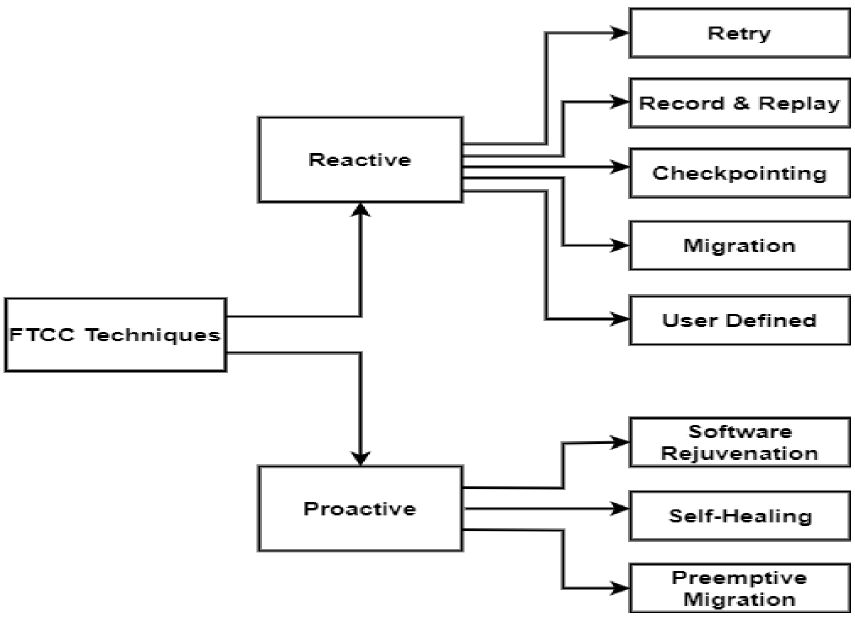

Figure 4.5: Fault tolerance techniques

Proactive Fault Tolerance

Proactive fault tolerance (PFT) techniques have defined preemptive measures to avoid system failure. Researchers are trying to develop various intelligent PFT strategies. Unlike RFT, PFT minimizes the risk of system failure by anticipation and proper system assessment. They are active and can be effective for critical systems. Some promising PFT strategies are as follows:

- **Software Rejuvenation:** It refers to the planned reboot of the system to change a particular state of the application. It is utilized based on request patterns and system behaviour. If a system reacts slowly for a prolonged period, the reboot of the system may evade the system crash. Besides, clearing cache memory, buffer overflow, and memory blocking are also efficacious. Prediction-based fine-tuned software rejuvenation strategies are introduced by Jean *et al.* [16].
- **Self-Healing:** Self-healing indicates the automatic detection of faulty modules of a system and the replacement of such modules to reduce the risk of system failure. Systems are generally equipped with multiple instances of the same module. Thus, requests can be easily rerouted to the alternative module in case of failure. It ensures a seamless service to the user. It is effective in case of load balancing and burst request management. An example of such a self-healing-based framework is presented by Tanim *et al.* for distributed software systems [48].
- **Preemptive Migration:** It indicates active monitoring and analysis of system modules. Intelligent preemptive migration can eliminate the risk of system failure. It may require colossal computation resources and integration of various tool feeds to generate an effective migration strategy to replace low-performing faulty modules.

4.5 Thundra

Thundra [115] is an open-source tool which provides a deep insight into a serverless environment. Thundra collects and correlates all the metrics, logs, and traces to quickly identify problematic invocations and analyzes external services associated with any lambda function. With zero overhead and automated instrumentation capabilities, Thundra frees developers to write code without worrying about bulking up their Lambdas or wasting time chasing black-box problems. As we search for good warm-up techniques for existing lambda functions in OpenLambda, we find that Thundra provides this service more efficiently, such as saving the work to create a warm-up service centrally for all the workers in the cluster. So, we use Thundra to warm up the selective lambda functions located in the cold queue to keep them warm.

4.6 Cold Start

As we discussed earlier, to execute a function for the first time or after having the function's code or resource configuration updated, the system will spin up a container to execute this function. All the code and libraries will be loaded into the container. The code will then run, starting with the initialization code. The initialization code is the code written outside the handler. This code is only run when the container is created for the first time. Finally, the Lambda handler is executed. This set-up process is considered a cold start [63]. Initially, the cold start time may seem like a redundant problem, but in the case of applications with a large number of requests, cold start becomes noticeable as it may take around 10min to respond.

4.7 Cache Replacement Policy in Container Management

The cache is the most expensive memory of a computer, but it provides the least access time and fastest response time. If we keep data in RAM or Hard Disk, it takes a long time to serve to handle simple requests. When frequent requests come for the same data, it is wise to keep that data close to avoid searching the long memory devices [129]. So, the cache holds the most recently used data in the processor. But the problem is the size of the cache. It is costly for computers to use large caches. So, it becomes a challenge to use this low memory optimally. Data are frequently written and erased according to the need. As the cache becomes full, replacing the old data with the new one is essential. As the cache doesn't allow random placement, the data replacement mechanism is organized in a very structured pattern, also known as "Cache Replacement Policies". It plays a pivotal role in different memory hierarchy systems. Different cache replacement policies have been developed to reduce the cache miss and latency, like LRU, MSU, LSU, SRLU, LIFO, TLRU, PLRU, RR, LFU, and Pannier in [59, 92]. The main objective is to utilize the cache and other resources more effectively. Among all the cache replacement policies, LRU is the most common. It exhibits thrashing behaviour for memory-intensive workloads that do not fit in the available cache.

In our problem, we have containers that, when serving any request, will be warm, and, later, it can go cold because of no further call in a threshold time. However, if we can keep all the containers warm, it will remove the container restart time. As the containers are warm, we can say they are ready to serve any request, and all necessary packages are installed already. The serverless platform will serve any lambda request instantly. But keeping all the containers warm will consume a large number of resources. As we can compare it with cache memory, like all data in the cache, all container in a warm state is impossible. So we need to do it selectively. As cache provides different policies to do the job, we also need a container replacement policy to switch containers between warm and cold queues. So we propose a modified S2LRU+ policy called S2LRU++ along with a third queue called "Template Container List". As the replacement is about containers instead of memory blocks, it brings more challenges in the architecture, especially for lambda containers.

Chapter 5

WLEC

In this chapter, we discuss the solution to minimize cold start time by defining WLEC architecture for serverless platforms. First, we provide an overview of the general serverless concept then we describe various components of WLEC architecture. We briefly mention the necessity of each component's structure and how they work together to provide a reduced cold start time. We briefly discuss S2LRU++, Template Container List, Warm Time, Container Header, Container Management Service, Container Management Service Function, and Optimization as the components of WLEC architecture.

5.1 Overview of WLEC

Fig. 5.1 shows the architecture of OpenLambda with WLEC. It consists of 3 main components. a)Load balancer based on Nginx [87], which receives the client request and assigns this request to a worker. The load balancing mechanism highly relies on the locality management of the system. The localities considered in this case were session locality, code locality, and data locality. b)Lambda store is also known as a registry that holds all the codes to handle different kinds of user requests. c)Execution engine, which consists of many workers. Each worker works as a sandbox based on a Linux container to execute code files, the function code associated with the lambda request issued by the client. We can see that WLEC works as a middleware for the load balancer and registry to provide the best worker available at any time for function execution. The internal architecture of WLEC is shown in Fig. 5.2, including all the components. The components are described in section 5.2. One of the most challenging parts of the design of WLEC is concurrent request and late request(after 15min) handling. The most challenging perspectives in this research are to address both design challenges with a single system and to manage the system accordingly. We assume that we can control the invocation of containers in lambda functions.

5.2 WLEC Components

In this section, we describe the main components of WLEC. We define the terms and components used in architecture. Then we narrate CMS and its sub-functions and their algorithms. Next, we delineate the workflow of WLEC.

5.2.1 S2LRU++

S2LRU is a partition technique used to make two segments, one for the probationary and the other for the protected segments. In the standard S2LRU algorithm, new data are inserted into the most recently used (MRU) position in the probationary segment. On a hit, data are promoted from the probationary segment to the MRU position in the protected segment. If data in the protected segments get hit, they are promoted to the MRU position of the same segment. When the protected segment exceeds its predefined size (e.g. half of the cache size), the LRU data are migrated to the MRU position in the probationary segment. For clarity, we rename the protected and probationary segments as the warm queue and cold queue [139] respectively shown in Fig. 5.2. Here we use containers instead of data. For S2LRU++, we maintain another queue named template container list, a copy of warm containers for each type of application request to serve any concurrent request. We will place the lambda containers with respective functions in the hot and cold queues instead of the data. The term “Data hit” is replaced with “Container invocation”. So, container placement is managed using the number of invocations of a container.

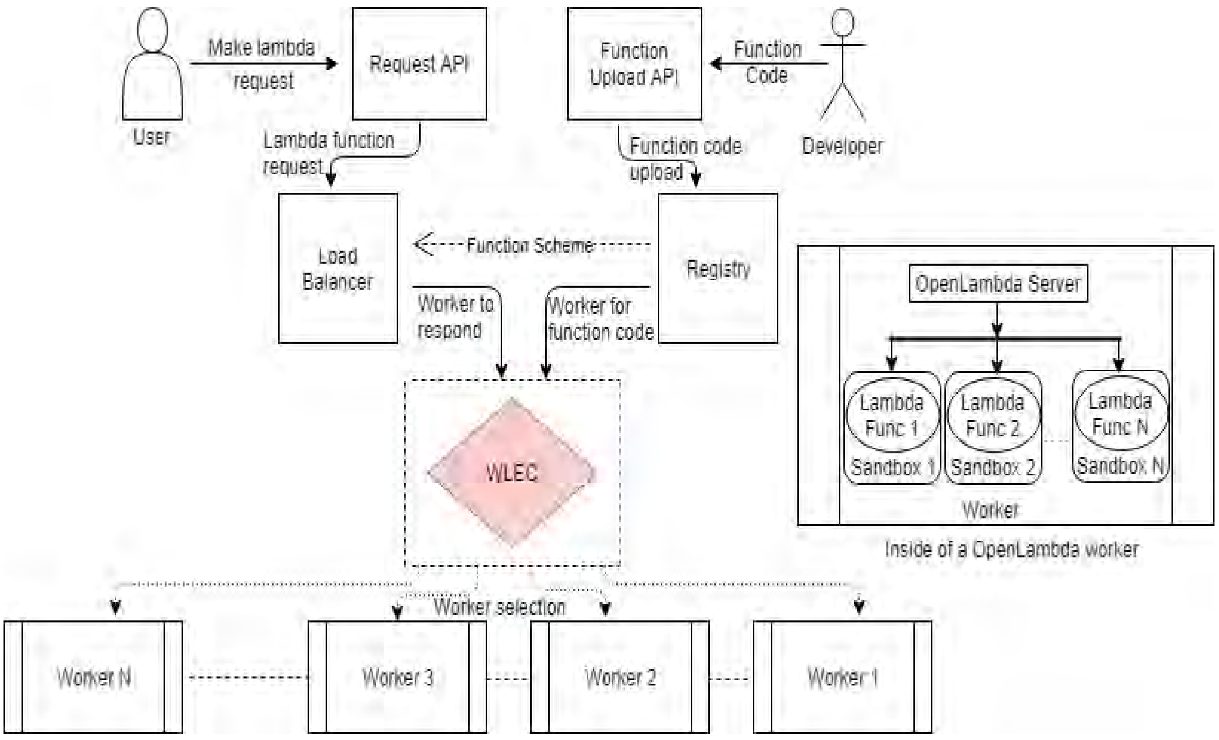

Figure 5.1: OpenLambda with WLEC

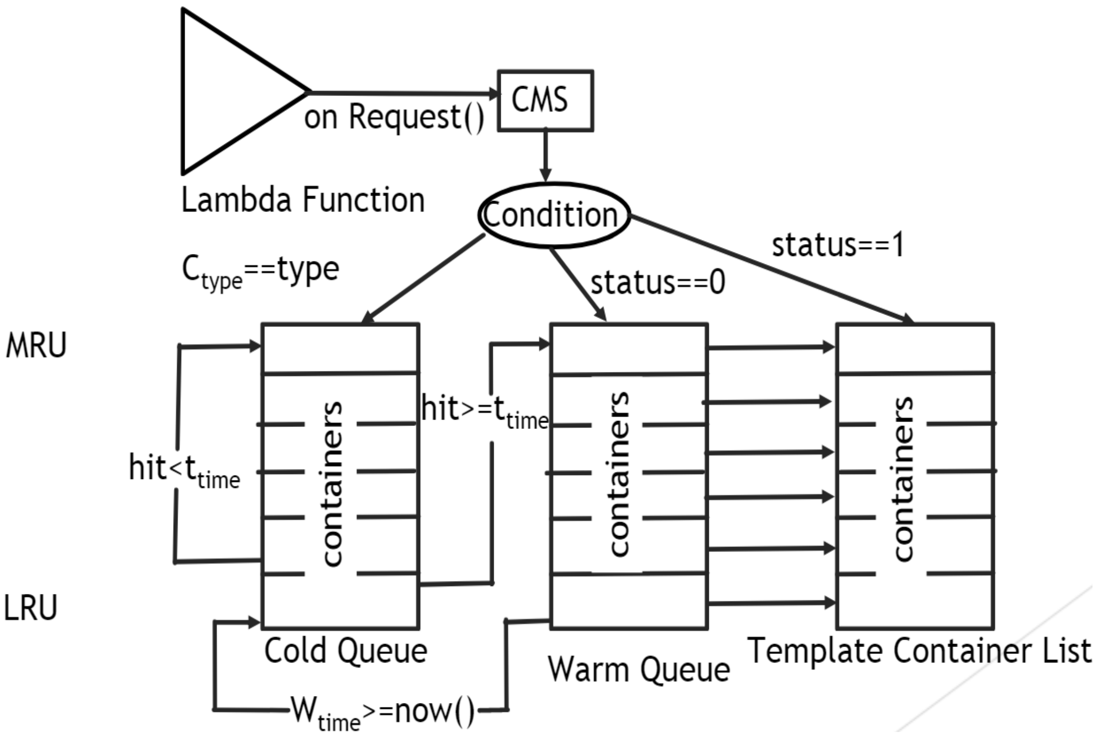

Figure 5.2: WLEC architecture.

5.2.2 Template Container List

A list of container which holds the duplicate container of all the containers in the current warm-up list. Here, the term duplicate container means another container with the same library packaging and environment configurations of a container that it is cloned from to serve any similar incoming lambda requests. If any concurrent request is issued for any lambda functions currently serving, another request will be passed to the copy container of this list. It reduces the cold start as the container is already packaged with the necessary libraries. As this list of containers is processed in the system's background, no cold start occurs for the request. After completing the response, the container is not pushed to any other queue as it is created to handle the concurrent requests only. If it is not busy, this copy container is destroyed, and the respective warm-up container moves to the cold queue simultaneously. As we try to minimize the cold start time, concurrency handling is a significant factor. We use this template list specifically to maximize the concurrency factor. However, the size of this template container list depends on the application. The total number of supported lambda request types defines its size. We also introduce a variable named the concurrency factor. It defines the number of copies a worker will have in the template container list. For example, the video transcoding application in [46] supports two different lambda requests from clients for HLS and DASH video coding. So, there will be two duplicate containers in the template container list with concurrency factor 1. Hence, the template container list size will be only 2 in this case.

Table 5.1: Table of header components

f_{type}	meta data about lambda function
q_{type}	the location queue of the container
$status$	execution status of lambda function
w_{time}	warm time of the container
hit	hit count of the container

5.2.3 Warm Time

For every container in the warm queue, there should be a warm time in that container header. It indicates the time stamp when a container should leave the warm-up queue. Warm time is maintained in the container header using Hr:Min:Sec format. The warm time of a lambda container will change based on its last invocation time. So, the warm time needs to be updated according to its last invocation time. For a container, its warm-up time is calculated as follows:

$$warmTime = invocationTime + a_{time}.$$

In the case of the first invocation, the invocation time will be the initialization time of the container. The a_{time} is the default time a container can live in the warm queue. Invocation time is configurable and varies from platform to platform from 15min to 45min. We assume a container can stay active for 15 minutes from its last invocation because 15 minutes is considered standard alive time in other serverless platforms like AWS, Azure, etc.

5.2.4 Container Header

We define some flags and variables in the header section to maintain, monitor and keep track of the containers. We propose five headers for every container in OpenLambda. The header parameters can be divided into two parts: Function parameters and Container parameters. Function parameters are maintained for each lambda, but container parameters are maintained for each. When a new lambda function is invoked in any container, the container parameters get updated if it is already initialized. Because lambda parameters get updated every time upon invocation. The f_{type} and $status$ variables are maintained for the lambda functions. The other three q_{type}, w_{time} and hit are used for the container, which holds a lambda function. In the initialization period, these flags are assigned and initialized by the Container Management Service and are also maintained and updated by the CMS in runtime. Every container has a header with the fields shown in table 5.1.

5.3 Container Management Service

CMS is the system that manages all the containers and their placement in queues. Whenever a lambda container is created, the container's management is done using CMS. CMS is a unique service that also incorporates and manages all the requests in WLEC. From request handling issued by a client to serving the request-all, the functionalities are maintained and served by CMS. As a container is created, the related flags are created by CMS, and through these flags and related algorithms defined in the CMS function section, the container is transferred to its rightful position. The architecture of CMS consists of two queues and one container list named warm, cold, and template container list, respectively. The containers kept in these structures are the principal concern of our architecture. If any container becomes invalid, it is tracked by CMS and evicted from these container structures. The symbols used in CMS algorithms are listed in table 5.2.

5.3.1 CMS Function

This section defines fundamental CMS functions as three functions and their algorithms. The three functions are Initialization, QueuePlacing, and OnRequest. The functions, their functionalities, and descriptions are given below:

Initialization()

When a new container is invoked, the headers are set accordingly. The hit count is incremented and is placed in the Q_c. Thus the q_{type} is set to 0. The f_{type} is set from the metadata of the lambda function provided by the function writer. Status is set to 1 when it is in the execution phase. Back to 0 when it has finished the execution. w_{time} is updated according to the algorithm 1. The initialization function is invoked every time a new lambda function is created. The lambda function needs to be packaged in a container to serve any request from the client application.

Table 5.2: Table of symbols of CMS

Q_c	cold queue
Q_w	warm queue
L_t	Template container list
c	a container
LRU	least recently used
MRU	most recently used
λ_{type}	lambda function type
t_{value}	threshold number to change queue
a_{time}	alive time(15 min)

The new container selection is determined in the OnRequest function. WLEC must create any container not found in the queues through the initialization method. The algorithm for the initialization of containers is given in 1.

QueuePlacing()

The QueuePlacing function works as a caretaker of these lambda containers initialized earlier. When a container has hit value 0, it is placed in Q_c in the LRU position. If it gets another hit, container c is then moved to the MRU position of Q_c. If it gets more requests, then the hit count gets incremented every time. If the hit count crosses t_{value}, it is then moved to the MRU position of Q_w. Each time a_{time} is updated on a hit. When the $now()$ crosses the a_{time}, then c is pushed to the Q_c. The algorithm is given in 3.

OnRequest()

This function is called whenever a request is issued from the client side. After the validation check of the request, a lambda container is picked from the queues using this function for executing the requested lambda function.

The logic is to first search in the cold queue. If found, it is returned instantly as it was kept warm by warm-up calls. If the cold queue does not have the container, it is searched in the warm queue. If the same type of container is found in the warm queue, then the state of the container is evaluated to determine if it is busy. If the status flag shows that it is not busy, then it means no request is in the processing state currently for that container then that container is returned. If it is busy, then this request is a concurrent request. So, the corresponding template container list is searched. When the corresponding container is found, the corresponding template container is returned to serve the request. A new container is needed if no suitable container in all the queues is found. Thus the initialization function is invoked to provide the new container to serve the request. The algorithm is given in 2.

5.3.2 Work Flow

When a request is issued from the client side, the request is sent to the OnRequest method of CMS. This client request can get, post, or put requests. The OnRequest method searches the cold and warm queues for the container that can serve the client request optimally. If any container is found, then that container is returned to serve the request. If the container is warm but busy serving any concurrent request, then the template container list provides the temporary container restored in the list. Then a new template container of the same type is created in the template list, and new pointer points from the same type of busy containers are in the warm queue to handle the next concurrent request for the same container. If the previously assigned container serves

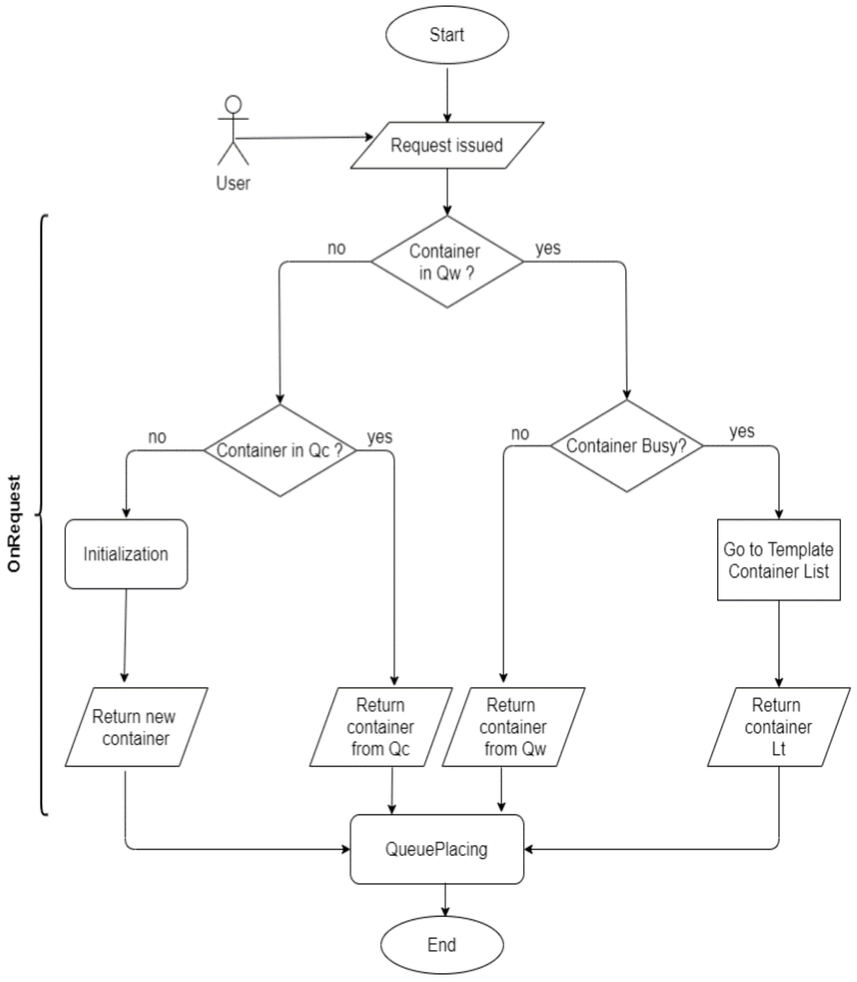

Figure 5.3: WLEC workflow

the request, that container gets deleted from the template list.

If no container is found in the lists, a new container must be issued. So, the initialization method is called, and a new container is returned to serve the request. All the corresponding flags are updated according to the algorithm of the initialization phase.
After serving a request, all the flags like hit, w_{time} are updated by CMS. Then the QueuePlacing method is called to place the lambda container in its respective position. The method checks the flags and variable values to determine which queue should hold the container. If it belongs to the warm queue, another template of the container is created in the template container list to serve any concurrent request. All containers in these queues are maintained, updated, and served by CMS using only CMS functions.

5.4 Optimization

We evaluated our architecture, and WLEC initially performed well. However, after 3hours of running time, the system got slower in our setup. After some investigation, we found that some optimizations were needed to make the architecture more viable and optimal than before. They are listed below:

5.4.1 Recycling of Workers

We found significant performance improvement after enabling scheduled recycling of the containers. We removed containers from the cold queue according to the sorted alive time. The queue became less crowded and easier to search. So, the excellent and qualified container's search time was lower than before.

5.4.2 Load Balancing

We also enabled load balancing to make proper distribution of lambda functions along with the warm container. It should be noted that load balancing is an optional feature of OpenLambda. We used round-robin load balancing, which also contributed to reduced response time. This uniform distribution of requests to the containers made the template containers more likely to be used and ensured the reusability of the warm containers.

5.4.3 Package Caching

For library and code packaging with Python 2.7.14, we used the Pipsqueak package caching tool to manage the packaging at each worker. It caches and maintains the Python interpreters in a sleeping state. This pre-installed package also reduced response time latency and eliminated downloading, installing, and importing packages in the workers.

The above optimizations alleviated the request handling procedure for OpenLambda, and our proposed architecture showed no slowdown in the runtime of our evaluation period.

Chapter 6

Advance WLEC

In this chapter, first, we describe the WaLCoR fault tolerance model and its workflow. Then we describe the determination of the Replica Factor (R_{f_i}) and Computed Replica Factor (R'_{f_i}) using Spearman's rank correlation coefficient to achieve dynamic replication of containers in the Template Container List. We present a high-level overview of WaLCoR implementation in Fig. 6.1 as a part of AdWaLEC in the OpenLambda architecture.

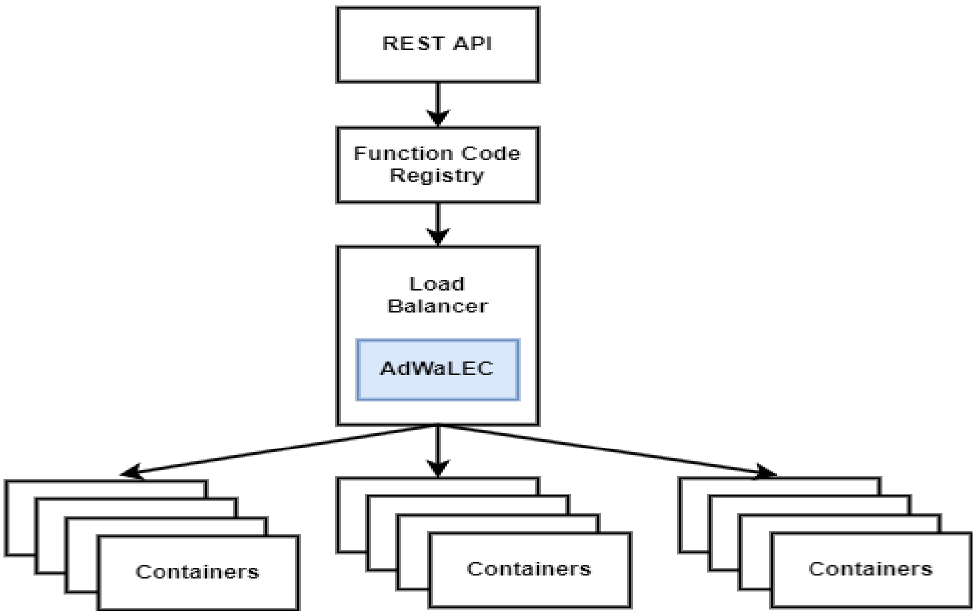

Figure 6.1: AdWaLEC in OpenLambda

6.1 WaLCoR

High availability is one of the critical factors for fault-tolerant applications nowadays. It refers to the ability of a system to be continuously functional and available to be used 99.999% of the time. However, keeping any system highly available requires special care for the architectural design and a handy fault-tolerant mechanism in place. As digitization is taking place in all sectors of our world, critical systems like air traffic control, oxygen supply management, and emergency response require 100% availability. Two strategies are commonly practiced in applications to achieve high availability, such as *Record & Replay* and *Checkpointing*.

In our case. to enhance the high availability feature in AdWaLEC, we introduce Warm Lambda Container Replication (WaLCoR). WaLCoR utilizes the initial recording mechanism for the input from Record and Replay but uses the checkpoint feature only to replay the execution stages in times of failure. Thus, it makes WaLCoR more efficient for serverless computation than the individual *Record and Replay* strategy and *Checkpointing*. The stateless nature of lambda functions makes it difficult to track the execution stages. To overcome this challenge, we divide the checkpointing into three main stages. 1)Input loaded, 2)Execution finished, and 3)Response sent.

In our case. to enhance the high availability feature in AdWaLEC, we introduce Warm Lambda Container Replication (WaLCoR). WaLCoR utilizes the initial recording mechanism for the input from Record and Replay but uses the checkpoint feature only to replay the execution stages in times of failure. Thus, it makes WaLCoR more efficient for serverless computation than the individual *Record and Replay* strategy and *Checkpointing*. The stateless nature of lambda functions makes it difficult to track the execution stages. To overcome this challenge, we divide the checkpointing into three main stages. 1)Input loaded, 2)Execution finished, and 3)Response sent.

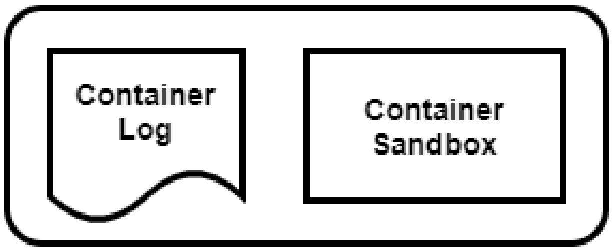

Figure 6.2: Container structure of template container list with log file

AdWaLEC includes a log file in each of the secondary replicas in the Template container list that is shown in 6.2. The primary container in the Hot queue sends over its input to the replica at the start of request processing. We know that template containers are already packaged with required libraries and dependencies. Upon receiving the input from the primary container, the log is updated with the “Input loaded” flag. The input gets loaded to the secondary replica. An acknowledgment message is sent to the primary replica after the completion of input loading. The primary lambda container continues its function execution. After finishing the execution, the message is sent to the secondary replica. The log of the secondary replica is updated with the “Execution finished” flag. After sending the response to the request, the primary container sends another message with “Response sent”. It is also noted in the log file. The sequence diagram is shown in Fig. 6.3.

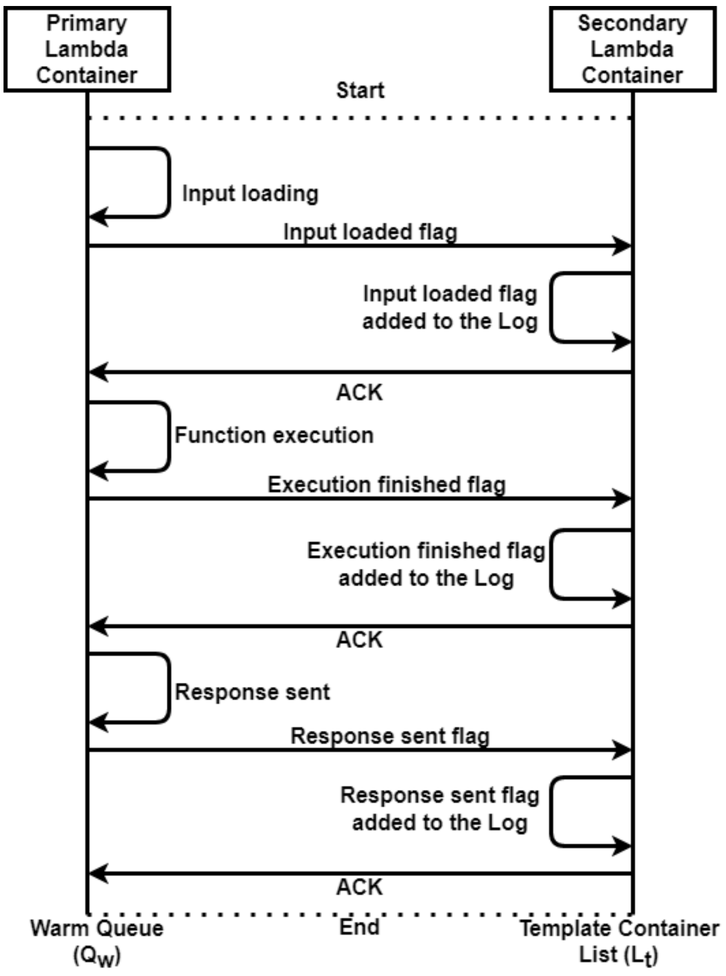
Primary Lambda Container
Secondary Lambda Container
Start
Input loading
Input loaded flag
Input loaded flag added to the Log
ACK
Function execution
Execution finished flag
Execution finished flag added to the Log
ACK
Response sent
Response sent flag
Response sent flag added to the Log
ACK
Warm Queue (Q_W)
End
Template Container List (L_t)

Figure 6.3: WaLCoR sequence

In times of failure at the container level, including cases like container corruption, data corruption, or computation failure, the fault manager sends the instruction to one of the replicas of the template container to start the execution of the function following the log stored in it, and the result is sent as a response. Hence, it can ensure high availability for warm containers using the template container list of AdWaLEC.

6.2 Fault Manager

Fault management and tolerance require efficient and effective continuous monitoring and fault detection mechanisms of any system. Besides, the WaLCoR management and fail-over process handling inspire us to incorporate a new entity into the AdWaLEC architecture. We name it Fault Manager. It handles container monitoring, fault detection, WaLCoR management, and fail-over process management. The processes are run based on their needs and provide fault tolerance service to the AdWaLEC. It ensures a separate fault tolerance management entity without delaying the function execution process. A basic structure of fault manager is shown in

Fig 6.4. The task of each process are described below:

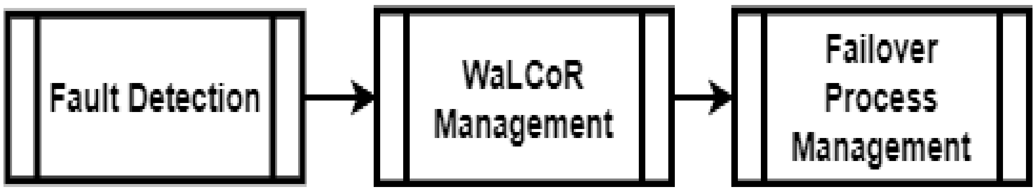

Figure 6.4: Tasks of fault manager

6.2.1 Fault Detection

The first task of the fault manager is to monitor each of the containers in hot, cold, and template container queues. The container monitoring process is running continuously to monitor each container periodically. If any of the queues exhibit suspicious behaviour, the fault manager can stop, destroy and relocate the container accordingly. For convenience, we consider that failure after 05 successive execution of a function as a fault incident. Unresponsive and idle containers are treated by relocating them to a cold queue. Through the container monitoring process, if any containers are identified as unresponsive and idle, it is considered faulty. For each detection, the primary container is shut down, and the secondary container is updated and checked with the last message in the log. Then the WaLCoR management process is called.

6.2.2 WaLCoR Management

The whole WaLCoR process is also monitored by the WaLCoR Management process of the Fault Manager. It ensures that every primary container communicates with its secondary pair as intended in the system design. If any discrepancies are found, the log messages are sent again. If that does not resolve the issue, the primary container is restarted. Besides, if any container requires to execute up to its log flag, the Fault Manager initiates such a process.

6.2.3 Failover Process Management

In case of failure, the fail-over process management process takes control. The last flag is checked with the secondary lambda container log. If it matches, It discards the faulty primary container using a global garbage collector and frees the space. The secondary container starts executing the lambda function following the log entry. After reaching the last checkpoint, the secondary container is moved to the place of the primary container. It resumes function execution, and a second replica is created by replicating the secondary one.

6.3 Determination of ρ, R$'_{f_i}$ and R$_{f_i}$

In this section, we calculate the Spearman's rank correlation coefficient (ρ) from the two variables container count (C_{c_i}) and priority factor P_{f_i}. Then we utilize ρ to determine Computed Replica Factor R$'_{f_i}$ and finally we compute Replica Factor R$_{f_i}$ from R$'_{f_i}$.

6.3.1 Priority Factor

The priority factor parameter indicates the importance level of certain types of requests. It works as same as weights for functions. Developers set the priority factor value based on their business importance with clients' suggestions. The priority factor can only be a positive integer number, which takes a value between 1 and 10. Hence, the value 10 priority factor means that function has the highest priority, and value 1 indicates the low priority. So, the (P$_{fi}$) is an ordinal value and largely depends on the client's choice. However, identifying the priority of function provides us with the knowledge to predict request traffic. It also helps us to organize the template container list with the right redundant containers. Based on the assigned value, this parameter is passed with the containers and sent to the Advanced WLEC. It is later used along with C$_{c_i}$ to make the rank correlation using Spearman's rank correlation coefficient (ρ). Finally, the ρ is used to determine R$_{f_i}$ and R$'_{f_i}$.

6.3.2 Container Count

The container count parameter records the total number of containers currently active in the hot queue at a single time of a certain type. This value is updated whenever a new container joins the hot queue. Advance WLEC maintains the value for each container in the n array. So, the C$_{c_i}$ value will always be a positive integer. The value can start from 0 to +∞. Here, 0 indicates that a specific type of request has yet to be issued so far. Hence, no lambda container in the hot queue has served it in recent times. This is also a significant parameter to calculate the R$_{f_i}$ and R$'_{f_i}$ later using ρ.

6.3.3 Steps to Determine Spearman's Rank Correlation Coefficient

We assume that they are independent and ordinal. From our collection of values, we find that both of them follow a linear pattern. So, Spearman's rank correlation coefficient would be the best correlation coefficient for our case. We emulate the following steps to determine ρ:

Step 1 We assume that two variable C_{c_i} and P_{f_i} has no association. This is called **Null Hypothesis**.

Step 2 Then, we rank each variable individually. We rank them from lowest to highest. The highest value of each variable is ranked as 1, and from there on, we rank the rest of the data. Hence, we create two ranked list for each variable C_{c_i} and P_{f_i} named $R_{C_{c_i}}$ and $R_{P_{f_i}}$. When the variables' values are the same, we take the average of their rank.

Step 3 For each ranked list we compute the rank difference d_i using the following equation.

$$d_i = R_{C_{c_i}} - R_{P_{f_i}} \tag{6.1}$$

Step 4 We determine the square of ranked difference called ${d_i}^2$.

Step 5 We find the value of ρ with the equation 4.1 for C_{c_i} and P_{f_i}. However, if we find that more than 50% of the ranked list values of either of the variable (C_{c_i} and P_{f_i}) has an average value/fraction number rather than a clear ranked integer ranked number, we modify the equation 4.2 as following:

$$\rho = \frac{S_{C_{c_i}} + S_{P_{f_i}} - \sum_{i=1}^{n} d_i^2}{2\sqrt{S_{C_{c_i}}.S_{P_{f_i}}}} \tag{6.2}$$

where

$$S_{C_{c_i}} = \frac{n(n^2-1) - \sum_{i=1}^{g}(t_i^3 - t_i)}{12} \tag{6.3}$$

with g is the number of groups with average ranks and t_i the size of group i for the x sample, and

$$S_{P_{f_i}} = \frac{n(n^2-1) - \sum_{j=1}^{h}(t_j^3 - t_j)}{12} \tag{6.4}$$

with h the number of groups with average ranks and t_j the size of group j for the y sample.

We use the value of ρ to find later the *Computed Replica* factor R'_{f_i} and *Replica* factor R_{f_i} for each container type.

6.3.4 Computed Replica Factor

The replica factor is a special variable introduced specifically in Advanced WLEC. The dynamic value assigned to the containers based on their request frequencies and priority can be considered a better approach to handling unpredictable burst request patterns. Therefore, the size of the template container list will also be dynamic. The R$'_{f_i}$ value is directly calculated from Spearman's rank correlation coefficient and C$_{c_i}$. One important thing to remember is that ρ can be positive or negative. But P$_{f_i}$ will always be positive and can be zero. So, to avoid negative R$'_{f_i}$, we take the absolute value of ρ. Accordingly, to avoid fractional value for R$'_{f_i}$, we take the ceiling value of the multiplication of C$_{c_i}$ and ρ. Compared with the WLEC, Advanced WLEC containers do not have an individual replica for each. Rather multiple same types of containers can have one template container. The R'_{f_i} is calculated with the following equation:

$$R'_{f_i} = \lceil C_{c_i} * |\rho| \rceil \tag{6.5}$$

6.3.5 Replica Factor

The R'_{f_i} value is passed to the Advanced WLEC manager to determine if that is valid. Hence, we determine the replica factor, R_{f_i}. The R'_{f_i} is compared to the total number of existing active containers (C_{c_i}) of that type. Here, the number of active containers is the sum of the hot containers from the Hot Queue and Template Container List of any specific lambda container type. This is done to ensure that we do not destroy any active functional containers currently processing a lambda request. The R_{f_i} factor for Template Queue formation is determined with the following equation.

$$R_{f_i} = \begin{cases} R'_{f_i} & \text{if } R'_{f_i} \geq C_{c_i} \\ C_{c_i} & \text{otherwise} \end{cases}$$

The R_{f_i} is passed to the *Initialization* function to create the template lambda container in the template container list.

6.4 Achieving Fault Tolerance

In the design phase of our Advanced WLEC architecture, we deliberately took those decisions that will provide fault tolerance features compared to WLEC. So, in this section, we describe our proposed methods, changes, and modifications of the previous WLEC. We also describe the logical reason for taking that decision route and how our architecture achieves fault tolerance.

6.4.1 Mitigating Single Point of Failure

OpenLambda, as well as all other lambda service providers, use ubiquitous random container selection methodology. A criticism of WLEC architecture was the central management of containers. It can easily lead to the *single point of failure* issue of the architecture. In terms of fault tolerance, the Single Point of Failure refers to the case where the failure of a specific component of a design can fail the whole system. The over-dependency on the S2LRU++ model of WLEC design is responsible for it. In Advance WLEC, we keep the baseline request serving process intact as a backup solution to our proposed architecture. So, in case of failure in our architecture, requests are served automatically following the baseline method ensuring service continuation.

6.4.2 Adaptive Container Selection

For Advanced WLEC, we adjust the WLEC architecture with a redundant container management system to mitigate the single-point failure fault tolerance. We use the traditional OpenLambda container selection process. It uses a random selection of containers rather than a selection based on previous request processing records. Initially, only the traditional random container selection is active. Based on the request pattern, the path controller component of the OpenLambda system controller calculates the rate of cold start cases. If the path controller reports that the cold start is transpired for more than 15% of the requests, then the controller switches the container selection process from random to an S2LRU++ based WLEC model. From our 20 simulations, we decided that 15% was the threshold value that provides satisfactory performance for OpenLambda. After that, the WLEC kicks in and maintains the latency within the state-of-the-art cold start latency.

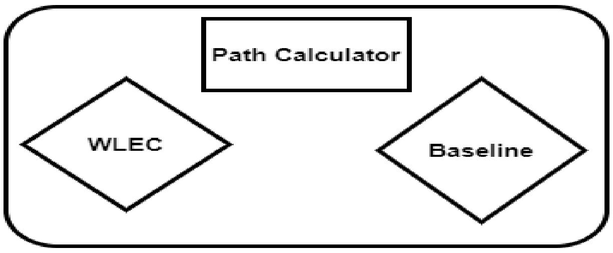

Figure 6.5: OpenLambda system controller

6.4.3 Data Consistency Assurance

For function chaining in serverless platforms, it is evident that data atomicity and data consistency requires close attention. The stateless nature of serverless architecture has affliction towards transaction processing. We find that stateful approaches are proposed to handle transactional workloads utilizing serverless architecture. But for starters, we introduce buffers to handle cases like a single lambda request trying to update the same variable multiple times. Instead of writing the value each time, we keep them in the buffer. This buffer is present in both the primary and secondary lambda container. The log of the secondary container also tracks each concurrent value change. We commit the value only after the completion of the entire request. Hence, we avoid latency regarding updating the same variable multiple times. Thus, it also ensures data atomicity and data consistency.

6.4.4 Data Corruption Mitigation

The effectiveness of any fault tolerance system depends on the data corruption mitigation success of that system. Advance WLEC shows promise as it has a redundant copy of processing data always present in the secondary lambda container of the template container list. Suppose any bad request or intentional actions create a situation where data gets corrupted in the warm container. In that case, the application can recover that data from the secondary container as data is loaded there too. Besides, the secondary container's log also helps track each step of the primary container's request processing phase. It provides the opportunity to do forensics to identify why data corruption occurred.

Chapter 7

Evaluation & Results

In this chapter, we evaluate our proposed methodologies-WLEC and AdWaLEC, after implementing them with various metrics. We assess their merit in different circumstances. We divide our evaluation process into two sections. First, we analyze the performance of WLEC in OpenLambda and compare different metrics from our settings to realize its effectiveness. Then, similarly, we evaluate AdWaLEC in OpenLambda and make a comparison with other approaches.

7.1 Evaluation of WLEC

The different numbers of workers are used to evaluate the performance of the WLEC approach. All the metrics are plotted to compare the performance with traditional approaches like the All warm or No warm approach. Here, All warm approach means a customized system of warming every worker initialized in OpenLambda for request handling. Details are described by Cordova [32]. Periodic ping requests are issued to each worker to keep them alive for a specific amount of time which is mentioned in the warming system. On the other hand, No warm approach excludes any customized warming mechanism. This approach uses only the typical OpenLambda setup.

7.1.1 Experiment Testbed

We did all the experimentation in the OpenLambda environment with two different virtual machine setups. One of them was on a local laptop computer, and another setup was on Amazon Web Server with an EC2 instance. In both cases, we used Ubuntu 14.01 as the Operating System. For simplicity, we used the default setup of OpenLambda from the Github repository [15]. We wrote the lambda functions in Python [15]. So, we also took the warm-up functions in the same

Table 7.1: Table of parameters and their values, default values are in bold

Parameter	**Value**
f_{type}	**get**, post, apiCall, dbOp
q_{type}	**0**,1,2
$status$	**0**,1
w_{time}	**15**,20,25
hit	**0**, 1, 2...
t_{value}	**25**,50,100
a_{time}	**15**, 16, 17 ..
$concurrencyFactor$	0, 1, **2**, 3 ..

language—the local machine VM was assigned 40GB hard disk space with 2GB RAM with four vCPU. The AWS VM was a t2.micro instance with one vCPU, 1GB memory, and EBS-type storage with 8GB storage space. We used AWS's US East region for our computation.

7.1.2 Methodology

For every experiment, we varied the total number of workers to 500, 1000, 1500, 2000, 2500, and 3000. We used ten different lambda functions from [23] as our workload and invoked them randomly. We kept the request number as same as the worker number. For warming, we used the Thundra [115] default warm-up mechanism using a lambda function call without any parameter. The metrics were calculated according to the AWS metrics definition provided in their documentation [99]. We took ten simulations with each parameter set and then calculated the mean. We kept ten requests per min as a standard request policy for both Local and AWS simulations. We set the cold and warm queue size maximum of 1/3 of the total available workers for each. Besides, client requests to these lambda functions were random to ensure a bias-free result. The value of the concurrency factor was two throughout the whole experiment. The parameters and respective values used for our experimentation are shown in Table. 7.1.

7.1.3 Results

Now, we illustrate the results of our evaluation regarding WLEC with six different metrics and then show a performance comparison. We define each metric and then compute them with our testbed setup in both platforms with OpenLambda. Next, we provide a quantitative improvement report against each stated metric with variances and standard deviations in the case of both Local and AWS VM in Table 7.2. Finally, an analogy of WLEC over a real-time image-resizing serverless application is incorporated.

Number of Cold Start Invocation

The number of cold start invocation metric shows the count of workers that faced cold start during invocation initiated by the lambda functions. In our experiment, we count the number of cold starts that occurred with the change in the number of workers. Fig. 7.1 shows the number of workers that suffer cold start problems. We find that the number grows with the available worker number proportionally. As more requests are sent, more containers are initialized, increasing invocations that suffer cold start in both Local and AWS VM cases. But with our WLEC, we find that the cold start occurrence is decreased by about 27% and 31%. Besides, we also find that AWS VM shows slightly better results. The better package management of AWS VM causes it. Though initially, we see a decline in the number of cold start invocations, as the number of workers grows, the superiority of WLEC gets more apparent, and we find 4x fewer containers that suffered a cold start at 3000 available workers.

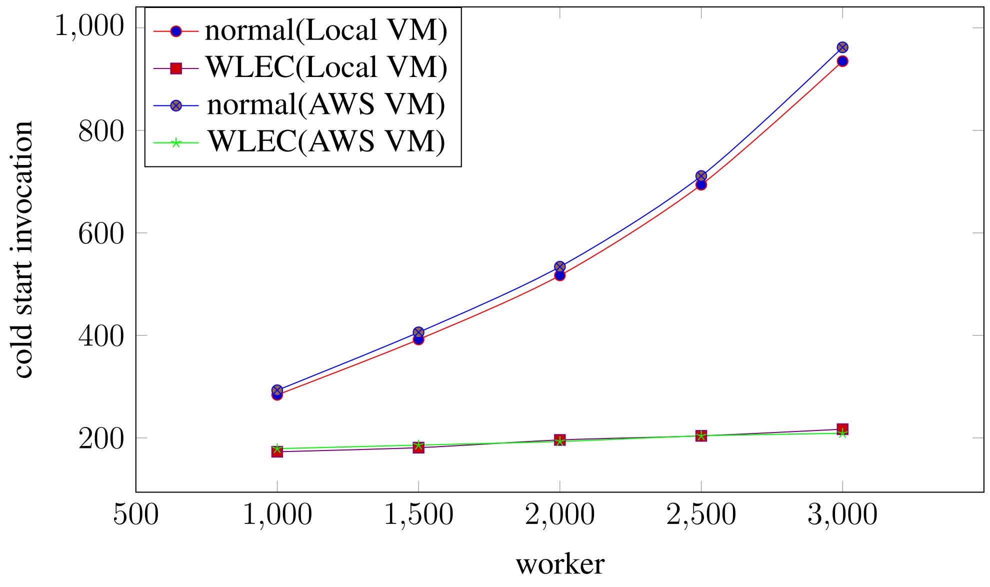

Figure 7.1: Cold start invocation vs worker

Average Cold Start Duration

This metric shows the average time the cold-started worker takes to respond to lambda requests. The ratio of total cold start duration to the number of cold starts for all workers determines it. Fig. 7.2 compares it for AWS and local VM, where we find that WLEC reduces the number of cold start occurrences and the duration of cold start simultaneously. As the cold start is found, the latency is shown on Y-axis. We find that the cold start duration decreases slightly with the number of available workers for all cases. Initially, the cold start duration is longer with less number of workers. We assume that the cold start duration shortens because of the caching and reuse of workers by WLEC. We can see a 21% improvement for Local VM whereas 23.5% improvement for AWS VM with WLEC in addition to around 4.5% standard deviation.

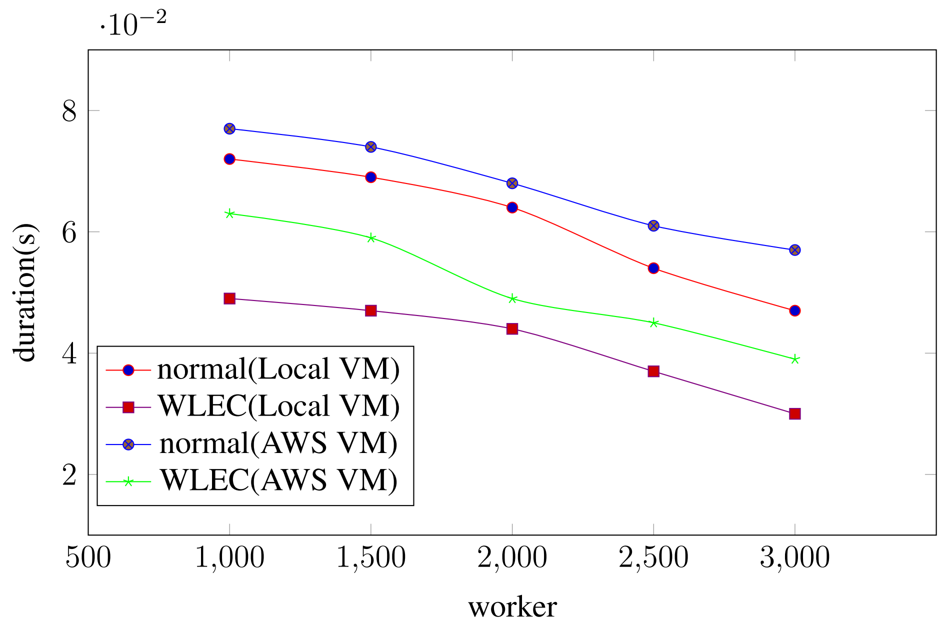

Figure 7.2: Duration vs worker

Number of Container Invoked

This metric counts the total number of containers invoked by the requests from any event or call from the pool of available workers. Like lambda function invocation count, it is also a metric in the serverless platform but only limited to the containers themselves. For the OpenLambda environment, The number indicates how many workers are invoked to the corresponding requests, which correspond to lambda events [11]. We change the number of available workers and then make the same number of requests to the lambda functions. We do not select any specific lambda

metrics	*environment*	*mean*	*variance*	*S.D.*
Total number of Container Invoked	Local VM	21.534	23.544	4.852
	AWS VM	23.558	18.771	4.332
Average number of Invocation Per Invoked Container	Local VM	27.964	71.324	8.445
	AWS VM	31.256	61.336	7.832
Number of Cold Start Invocation	Local VM	60.53	174.026	13.191
	AWS VM	60.53	174.027	13.192
Average duration of Cold Start Invocation	Local VM	32.544	3.352	1.830
	AWS VM	25.44	33.970	5.828
Maximum StartUp Latency	Local VM	70.218	644.589	25.389
	AWS VM	77.63	374.009	19.339
Occupied Memory$*$	Local VM(WLEC to No Warm)	39.288	40.272	6.346
	Local VM(WLEC to All Warm)	50.36	7.942	2.818

Table 7.2: Mean, variance, and standard deviation of percentage improvement between no warm and proposed strategy across different metrics on both experimental environments. $*$ was taken only on Local VM and the improvement was shown between WLEC to No warm and WLEC to All warm strategy

function. We instead keep it random. We do this simulation for both the Local VM and AWS VM. In Fig. 7.3, we took the different numbers of available workers and made the same number of requests to the lambda functions. The functions then invoked the workers. From the plot, we can see as the number of workers increases, the number of invocations also increases. The reason here is as the number of workers increased, more fresh workers were available for a lambda function. The invocation increase also indicates that the requests invoke more workers, and less concurrency occurs in the workers. But as the workers' number grows larger than 2000, we see a slight sluggishness at the invoked container number. It must result from the saturation of the required number of free workers. Yet, the WLEC architecture implementation always invokes less number of containers to respond to the requests because of its reuse principle of workers. Between Local and AWS VM, we find AWS invoked slightly more container invocation, which may be caused by network delay.

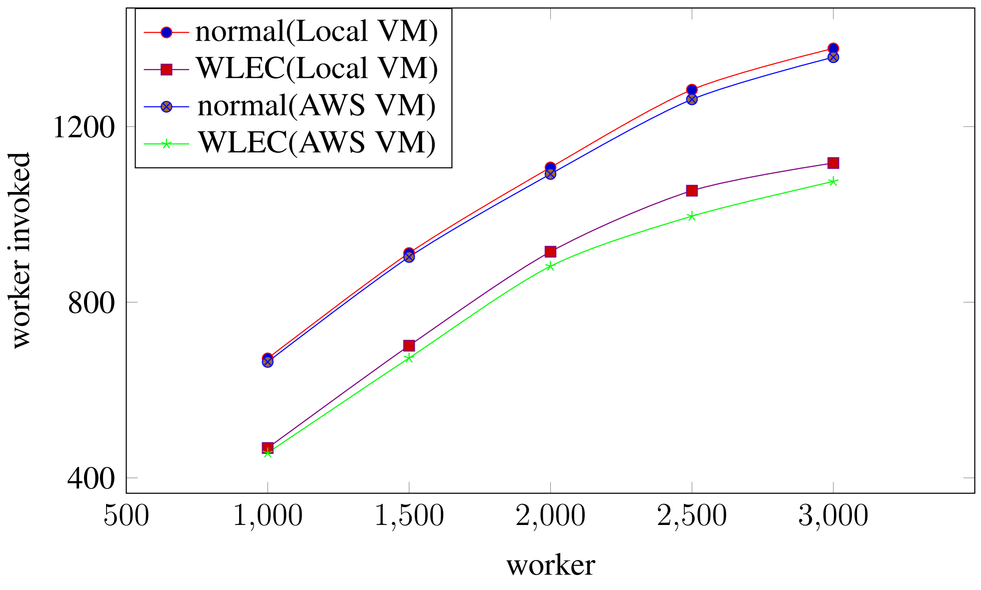

Figure 7.3: Worker invoked vs worker

Average Invocation per Invoked Container

The ratio of total lambda requests to the number of invoked containers calculates the Average Invocation per Invoked Container metric. This metric approximates the number of times the requests invoke a worker. Dealing with burst requests requires the repeated invocation of available workers. High invocation per invoked container indicates a reduction of fresh starts of workers resulting in less cold start occurrence. From this metric, we can predict how many workers may need to be warmed to deal with the burst if any burst has come. We show the mean number of invocations for each invoked container in Fig. 7.4. We can also denote it as a mean of the number of requests handled by each worker. We find, in both AWS and Local VM, the mean of invocation per invoked container is low with the small number of available workers. As it gets more available workers, the mean gets lower. However, WLEC uses the same workers as much as possible, providing higher invocation per worker. The mean increases with the request number because more workers can be alive. Even so, this leads to more invocations in the same workers. From our calculation, we see 27.96% and 31.25% mean invocation increase per container with 8.4% and 7.8% S.D., respectively, for Local and AWS instances.

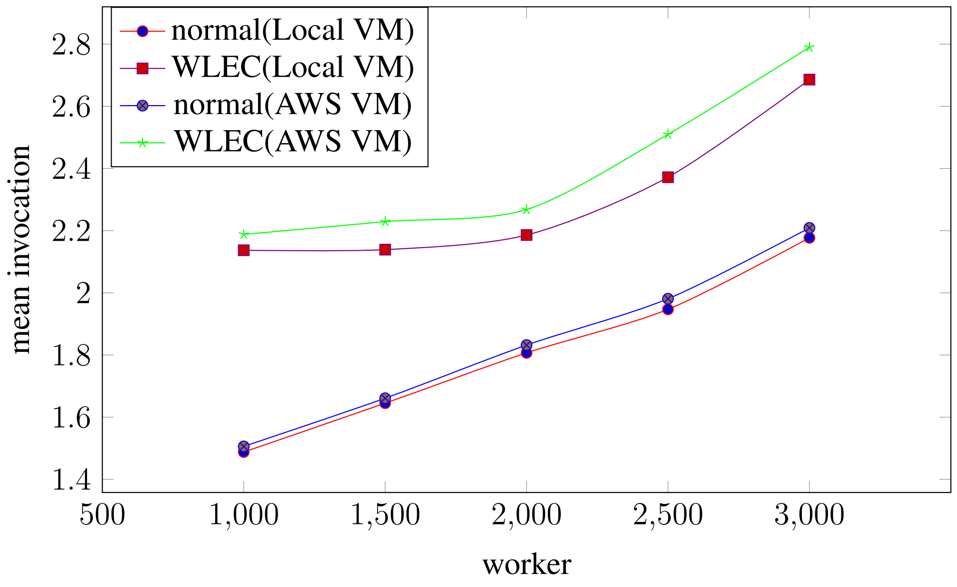

Figure 7.4: Mean invocation vs worker

Maximum Start-up Latency

Start-up latency [99] means the time workers take to take the first fresh start. Maximum start-up latency was calculated from the maximum time taken by any of the given workers. This metric is directly linked to our cold start reduction as the start-up time contributes most of the cold time for the workers. Low latency indicates that the system takes less time to prepare for request handling.Fig. 7.5 shows how the start-up latency is affected by the number of available workers in cold and warm conditions. As we can see, initially, the start-up latency is low in the cold state, but as the number of workers increases, the latency also increases proportionally, as expected. But for a higher number of workers, especially from 2500 workers, the latency graph jumped and continued upward afterward. The latency becomes a severe issue as it starts to take around 8 min for 3500 workers in our setup. So, if we use WLEC, we observe that the curve does not jump up like before for any number of workers. Even for 3000 workers, it only takes around 15sec in both AWS and VM instead of the 372sec maximum delay shown earlier-resulting 77.6% and 70.2% start-up time reduction.

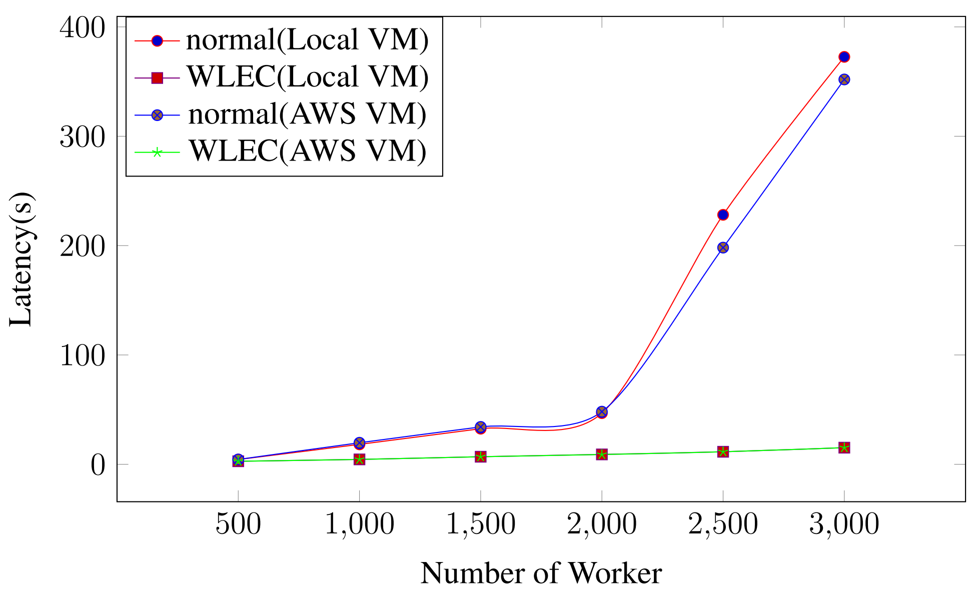

Figure 7.5: Maximum start-up latency vs worker

Occupied Memory

We use another metric named Occupied Memory to describe memory consumption for the different numbers of workers. It indicates the total sum of the memory occupied by all warm containers. It shows the lightness of the architecture, which leads to higher memory availability for concurrent executions. Besides, a better architecture will always cause less memory occupation despite serving the same number of requests. Fig. 7.12 shows how the OpenLambda platform behaves in the case of the different number of available workers in the Local VM environment. No warm setup requires less memory, but the performance degrades rapidly because of no caching. For all warm architecture, the memory requirement is unrealistic for large applications as it keeps all the containers warm. Instead of WLEC, we can see that the memory requirement is halved from all warm approach though the performance remains acceptable. We keep the container size at 10MB. We also find around 39.28% more memory required for WLEC over no warm approach. On the other hand, we reduce total occupied memory by 50.36% compared to all warm strategy.

7.2 Evaluation of AdWaLEC

Now, we assess the performance improvements of AdWaLEC. This section explains our implementation of AdWaLEC and describes our evaluation process in detail. We discuss our experiment testbed and methodology for AdWaLEC. Then, we compare AdWaLEC with approaches such as baseline, all warm, SOCK, and WLEC. Likewise, we show the worthiness and effectiveness of AdWaLEC using the same OpenLambda platform. In addition, we implement AdWaLEC and WLEC with a real-time image resizing application to determine and compare

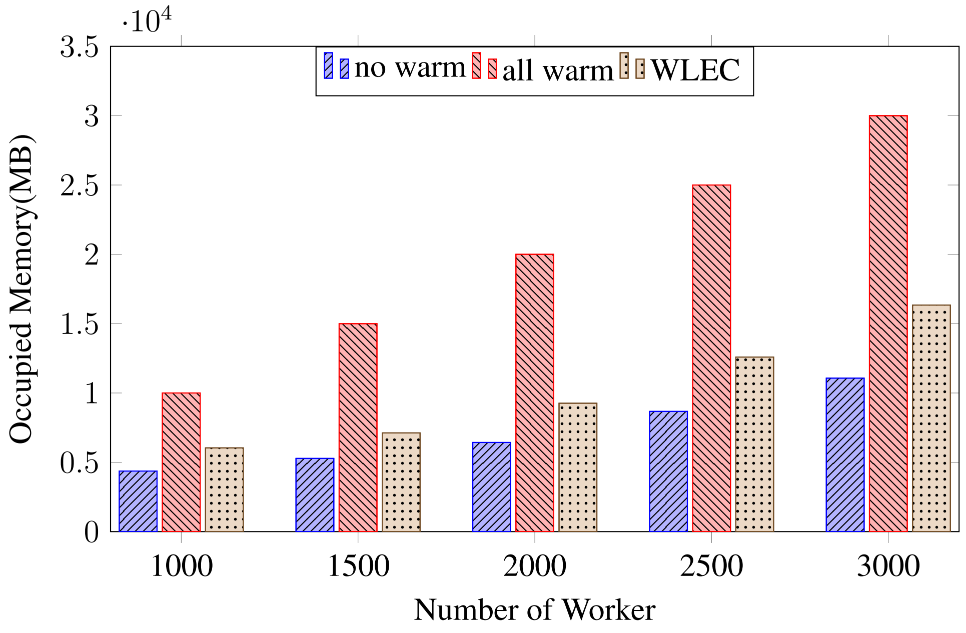

Figure 7.6: Occupied memory vs worker

their acceptability and practicality in a realistic layout.

7.2.1 Experiment Testbed

We evaluate the performance of AdWaLEC for serverless architecture utilizing the OpenLambda platform. We use Amazon Web Server with an EC2 instance with Ubuntu 14.01 as the Operating System. We run our experiment on an m5.8xlarge instance with 32 vCPU, 128 GB RAM, and 200 GB EBS storage in AWS's Asia Pacific (Singapore) region for our simulation. We utilize the Azure Function trace from [102] to simulate different function invocation scenarios varying in execution time, memory usage, and cold start latency. However, [102] contains a collection of over 50,000 functions in addition to their execution times, memory sizes, and invocation timestamps. For better-tailored performance, we filter the data set and make a representative collection of 1000 functions based on their invocation frequency. We select 50 random functions from the representative collection for our experiment. For simplicity, we used the default setup of OpenLambda from the Github repository [15] written in Python. We put OpenLambda container sizes fixed at 400 MB as [102] suggests 90% applications consume less than 400 MB memory.

7.2.2 Experiment Methodology

For every experiment, we vary the memory size from 50%(64 GB) available memory to up to 100%(128 GB). Then we use 50 random functions selected from the representative collection trace. We take about ten simulations with each parameter set and then calculate the average for the analogy. To provide better insight regarding AdWaLEC, we compare performance with the state-of-the-art technique-SOCK approach from [89], baseline OpenLambda, WLEC and all warm approaches. Here, baseline means using the default keep-alive policy (10 min) of OpenLambda. On the contrary, the all warm approach indicates all the containers are warmed utilizing [36]. A random priority factor value was set before the simulation, where each function was assigned to a value starting from 1 to 10 based on their invocation frequency. The container count was calculated on the runtime to determine the Replica factor. Other parameters and their respective values are taken from [105].

7.2.3 Results

Now, we illustrate the results of our evaluation regarding AdWaLEC with seven different metrics and provide a performance comparison. First, We define each metric and then compute them with our testbed setup with OpenLambda. Then, We calculate the outcome for AdWaLEC, WLEC, SOCK, baseline, and all warm approaches and plot them for better understanding. Finally, we implement AdWaLEC on a real-time image resizing application to discern the impacts for a practical case.

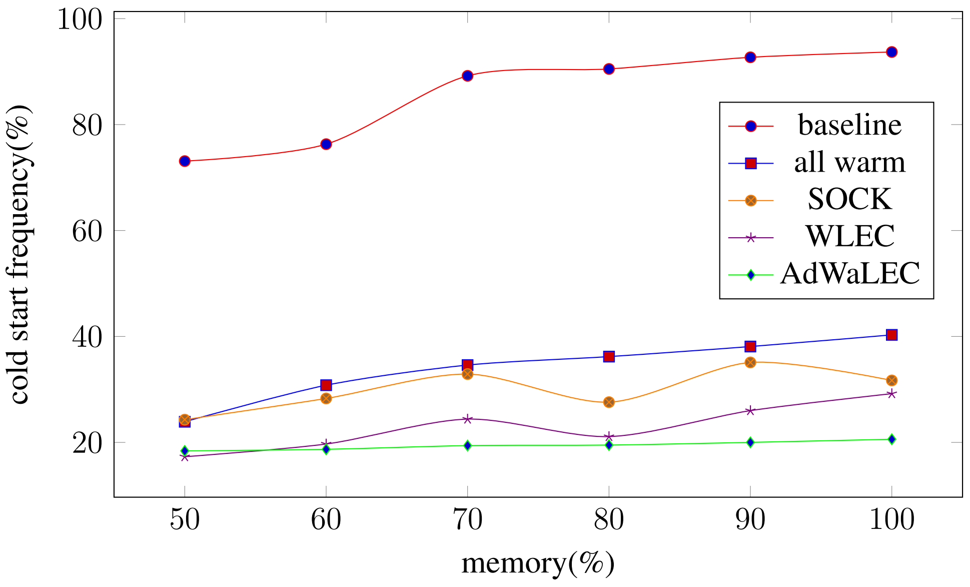

Figure 7.7: Cold start frequency vs memory

Cold Start Frequency

The cold start frequency shows the percentage of containers to the total requests that faced cold start at the time of invocation, initiated by the lambda functions. Fig. 7.7 shows the percentage of containers that suffer cold start problems. We find that the number grows with the available memory proportionally for all cases. Nevertheless, with AdWaLEC, we find that the cold start occurrence is decreased by about 77.23% and 42.21% to baseline and all warm approaches, respectively. However, an important takeaway is that AdWaLEC does not linearly increase cold start occurrence and almost remains the same, outperforming SOCK by 33.84% on average. The dynamically created replica containers are always present to serve any incoming request in our AdWaLEC scheme. Hence, the cold start occurrence remains constant for every available memory case.

Cold Start Latency

This metric shows the average time the cold-started container takes to respond to lambda requests. Fig. 7.8 compares all five approaches. The cold start latency decreases slightly with the available memory space for all cases. For AdWaLEC, we find that the average cold start duration is slightly higher than WLEC. It might occur because of the constant number of container replication strategies of WLEC. In this case, WLEC outperforms AdWaLEC by 17.62%. However, we find that AdWaLEC performs better for higher available memory space than SOCK. The large zygote forking overhead of SOCK may cause it. We also find 27.36% and 18.72% cold start latency reduction than baseline and all warm approaches, respectively.

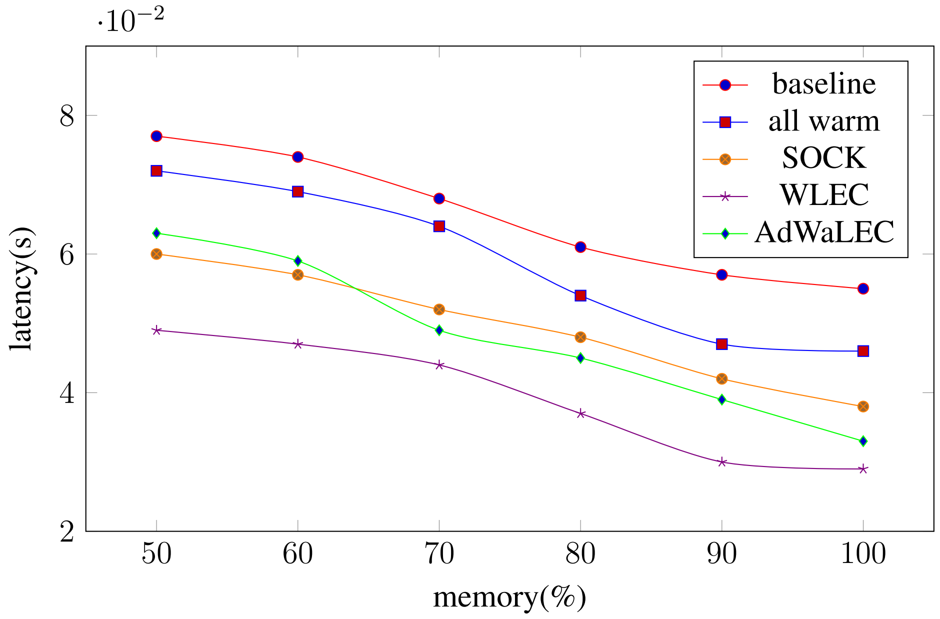

Figure 7.8: Latency vs memory

Container Invoked

This metric counts the percentage of containers invoked by the requests from any event or call from the pool of available containers. From Fig. 7.9, we can see that as the container's percentage decreases, the percentage of memory increases. The reusing nature of the already invoked container of AdWaLEC ensures the reuse of idle containers rather than creating a new one. So, as the memory increases, the invoked container percentage decreases. AdWaLEC decreases new container invocation by 23.5% and 22.6% to baseline and all warm approaches. Besides, we find AdWaLEC improves container reusability by 13.84% and 3.8% against SOCK and WLEC, respectively. AdWaLEC achieves this by avoiding new container creation and adopting replica containers.

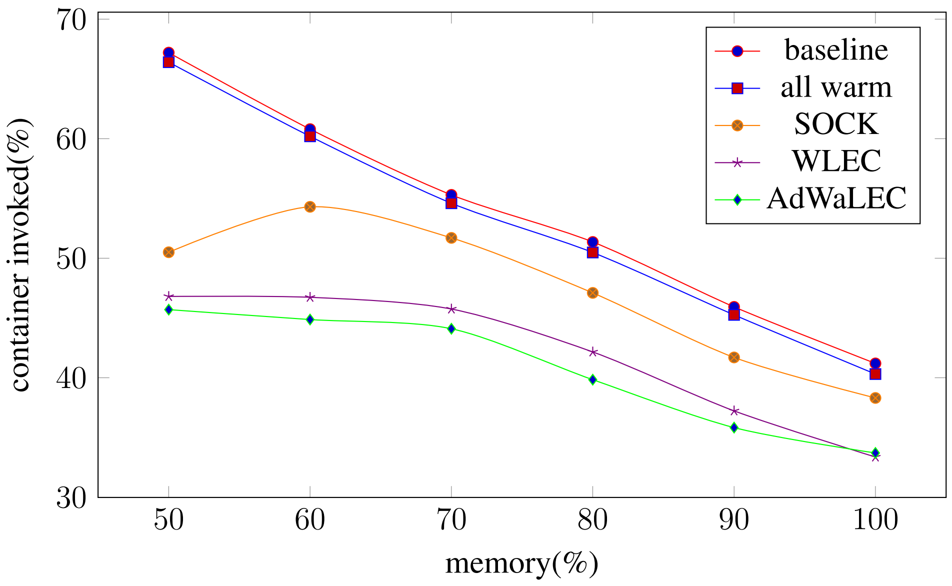

Figure 7.9: Container invoked vs memory

Invocation per Container

The ratio of total lambda requests to the number of invoked containers calculates the Invocation per Container metric. Fig. 7.10. We find that mean invocation increases as memory increases. However, AdWaLEC uses the same containers as much as possible, which provides a higher invocation per container. From our calculation, we find that AdWaLEC has 29.95% and 30.39% better invocation per container than baseline and all warm approaches. As SOCK creates a new container efficiently with the help of zygote provisioning instead of ensuring reusability, AdWaLEC also exceeds SOCK by 10.88%.

Maximum StartUp Latency

Startup latency means the time workers take to make the first fresh start. Maximum Startup latency was calculated from any container's maximum time taken to serve a request. Fig. 7.11 shows how the startup latency is affected by the number of available containers in the case of cold and warm conditions. For AdWaLEC, it remains constant like WLEC also. When a large number of requests create template-ready containers, they incur latency. However, AdWaLEC creates replica containers behind the scene and always ensures ready containers available for

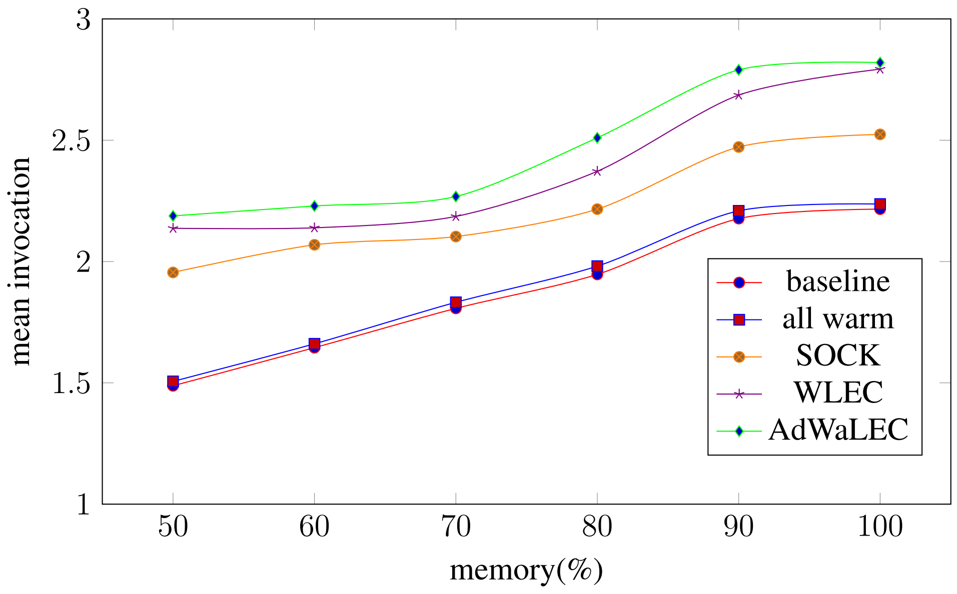

Figure 7.10: Mean invocation vs memory

function invocation. This AdWaLEC provides the lowest latency of all other approaches in this study. We find that AdWaLEC outperforms baseline and SOCK by 77.28% and 67.68%. WLEC performs better for lower memory availability, but AdWaLEC exceeds WLEC by 28.43% on average for higher memory availability.

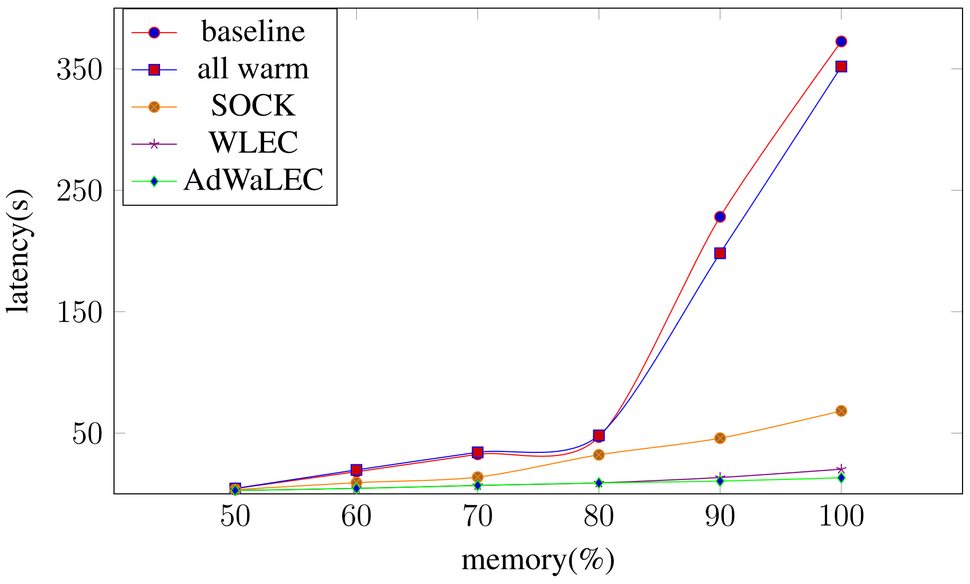

Figure 7.11: Maximum start-up latency vs memory

Occupied Memory

Occupied memory indicates the total percentage of memory occupied by all containers. It shows the lightness of the architecture, which leads to higher memory availability for concurrent executions. Fig. 7.12 shows a comparison of occupied memory amount in the case of the different amounts of available memory. For all warm architecture, the memory requirement is unrealistic for large applications as it has massive memory consumption. As all containers are kept warm, it almost consumes the whole available memory space. In the case of AdWaLEC, memory consumption is relatively lower (about 26.49%) than all warm approach. The replica containers of AdWaLEC contribute to a higher memory consumption than SOCK. We think this is a reasonable trade-off between fault tolerance and memory consumption for AdWaLEC. AdWaLEC also outperforms the baseline by 6.69%. WLEC occupies more memory than AdWaLEC initially but later, it shows improvement in higher memory availability cases and reduces consumption by 26.45% because of its static container replication scheme.

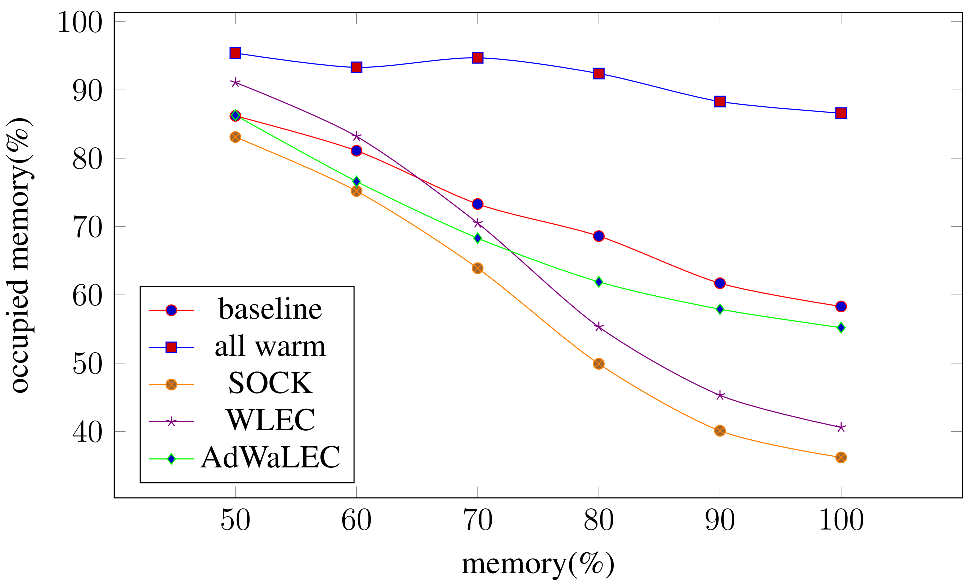

Figure 7.12: Occupied memory vs memory

Recovery Latency

We measure the capabilities of AdWaLEC at the time of execution failure using the recovery latency metric. Here, we start our experiment with a similar amount of available containers. After 10 min of execution, we reduce the number by half. We terminate those containers randomly. Hence, it is most likely, that many of those containers will be middle of execution in times of termination. This scenario will force the corresponding template containers to run a fault tolerance scheme and provide the response by following the process mentioned in the WaLCoR sequence. Recovery latency reduces with memory as higher memory ensures a warm container for function execution shown in Fig. 7.13. Only all warm and AdWaLEC can provide fault tolerance capability for low memory availability. However, keeping all the warm containers causes a high recovery time because of the high searching and container creation latency. We find that recovery is possible only for SOCK, baseline, and WLEC for 70% or higher available memory. The WaLCoR mechanism of AdWaLEC contributes to 82.6% and 85.43% recovery latency reduction compared to SOCK and all warm approaches. For critical applications where requests are highly valued and do not necessarily deal with continuous burst flow, our recovery latency would not impede AdWaLEC adoption.

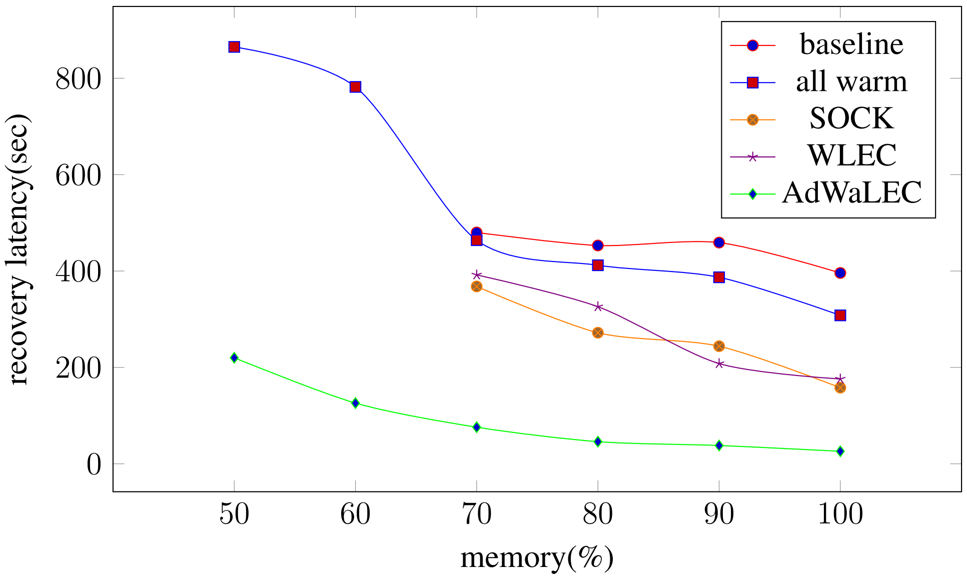

Figure 7.13: Recovery latency vs memory

7.2.4 Case Study: Image Resizing

To realize the impact of AdWaLEC, we evaluate AdWaLEC by implementing it in a real-time serverless application. For that, we consider an application from the AWS lambda application pool. Then we modify the application based on OpenLambda architecture rather than its typical AWS build. Next, we implement it in the OpenLambda platform and calculate both platforms and compute latency in baseline, with WLEC and AdWaLEC, to present a proper comparison. We use an on-demand image resizing application from [76] to evaluate AdWaLEC with the real-time application. In Fig. 7.14, We determine the platform and compute instance latency of the image resizing application with an average of 20 runs. The result shows that the compute and platform latency is reduced to 224sec and 576sec for AdWaLEC. So, it is improved by 37.95% and 65.67% compared to the baseline OpenLambda setup for compute and platform instances respectively. Nonetheless, WLEC shows 55.90% and 88.76% less latency for the platform and compute instance respectively compared to AdWaLEC. The latency overhead can be attributed to the initial computation of the correlation coefficient.

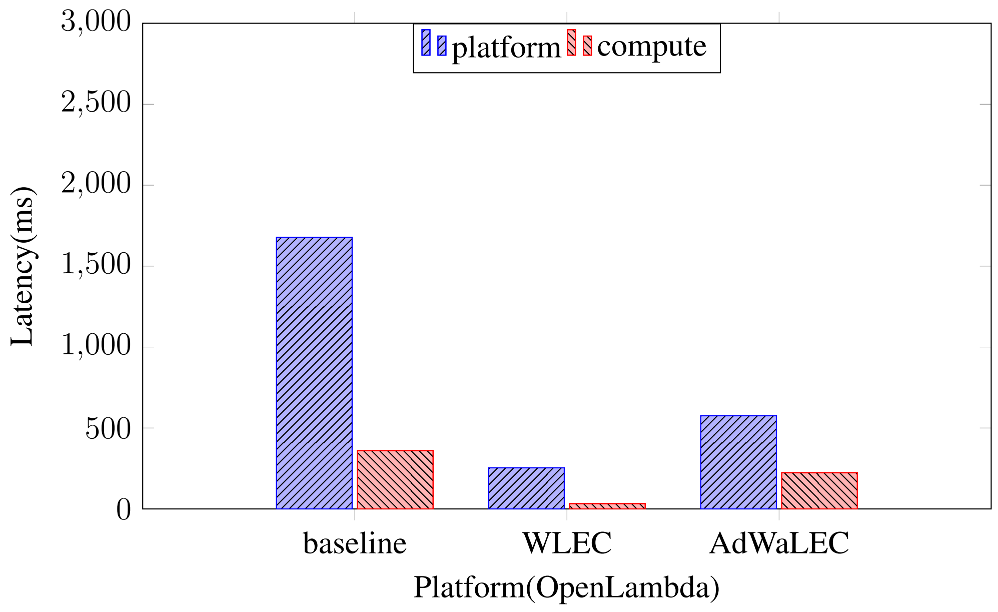

Figure 7.14: Latency vs platform

Chapter 8

Future Work

Though we try to make WLEC as advanced as possible, more research is required to eliminate the cold start problem. Elimination may require proper and effective redesigning of serverless platforms like AWS, OpenWhisks, and Google Cloud. But redesigning will always cost more effort, resources, and time. Large containers and divergent containers prove a big challenge for our architecture. Enabling WLEC to be compatible with other lambda platforms can also take time and effort. It is also an open problem for Warm Up architecture to solve the random container picking issue in the serverless platform. Instead of a warm queue structure, a warm stack or consistent hashing [110] can also be another topic of interest.

8.1 Enhancing Versatility

The next step is implementing AdWaLEC in other open-source serverless platforms like OpenWhisk, OpenFaaS [6], and Knative [4]. Furthermore, we intend to evaluate AdWaLEC performance with divergent workloads like AWS Lambda Function trace and Google Cloud Function trace. In addition, we plan to assess the merit of using other rank correlation coefficients such as Pearson's r and Kendall's τ. Besides, Fine-tuning the user-given priority values and including weighted schemes can be an exciting topic for researchers to investigate.

8.2 Security

In this work, we mainly focus on the design, implementation, and effectiveness of WLEC architecture. We plan to consider the security concern of WLEC for our next research step. We plan to improve WLEC's features like intrusion detection, misuse detection, and security automation before code execution. Our plan also extends to make WLEC compatible with other security services for serverless architecture like virus scanning, compliance checking, and incident response. Various attacks like side-channel attacks, man-in-the-middle attacks, Eavesdropping, Malware installation, etc., still pose a prime threat to serverless architecture and WLEC. We intend to design a similar threat management system like the Serverless Threat-Intelligence Platform proposed by Sanghyun *et al.* [53]. It will ensure secure function code execution, seamless integration with other security tools like AWS X-Ray, and automatic vulnerability scanning with the recommendation to thwart such vulnerability.

8.3 Queue vs Stack

One potential research idea regarding WLEC is introducing a stack rather than a queue. We want to experiment by implementing WLEC with stack and find out how it will behave. It is possible to get better performance in case of some special conditions. It is another research topic to design using stack and queue. These approaches may lead to the flexible and versatile design of container caching depending on application and platform constraints. Further extensive study is required to find the viability of such architecture.

8.4 Windows Container Support

Another new research area can be the windows container support for AdWaLEC architecture. As OpenLambda workers are purely Linux-based, windows-based workers remain another prospect to be explored. It will make AdWaLEC a versatile solution and ensure future integration with Azure services. It will open a new dimension for AdWaLEC regarding architectural modification and performance-oriented designing. Automated support for old and new windows OS versions will also be a challenge to overcome.

Chapter 9

Conclusion

In this chapter, we review the ideas discussed in this thesis and how these ideas helped this thesis to achieve its goal.

In this thesis, we have addressed the cold start problem of serverless architecture, also known as "Faas". We have introduced a container caching mechanism named WLEC to minimize the cold start time. We discussed our design choices and implementation process of WLEC and then showed our experiment result.

At first, we started with the contemporary cloud architecture and its implementation variations like public, private and hybrid cloud. Then we mentioned the evolution of cloud computing to an as-a-service cloud model. We have delineated eight as-a-service models and provided a primary overview for all of them. Later, we explored various virtualization techniques used for the previously mentioned service model and showed their significance for the "Function-as-a-Service" model. Next, we studied contemporary serverless architecture and reviewed its advantages over traditional architecture. Then, we have considered major serverless service providers of the ubiquitous market. After that, we have provided a detailed discussion of the future challenges for serverless architecture. Next, we have mentioned a survey result to understand the current demand for serverless models among professionals. Then we mentioned the scope of this thesis, main contributions, and thesis organization.

In Chapter 2, we have mentioned the background and motivation behind this thesis. We first discussed the architectural perspective to present the need for an architecture-based solution. Then, we presented our investigation regarding cold start time in the current OpenLambda architecture. We have also shown the latency for different container states. Later, we explored current practices and trends for minimizing cold start time in various serverless service providers.

In Chapter 3. we have narrated a few related works associated with our thesis. Here, we have discussed the works in six main categories. We first mentioned some performance analysis studies. We have divided those studies into platform-based approaches and cold start time-based approaches. Next, we have mentioned some recent research studies regarding cold start minimization. We have also divided these studies into architecture and application-level approaches. Then for AdWaLEC, we mention research studies regarding the formulation of rank correlation and its diverse applications. Next, we state various fault-tolerant-focused works to discuss the importance and implication of fault-tolerant systems. We have mentioned the most recent and relevant approaches to our knowledge in this chapter.

In Chapter 4, we have defined some terms used throughout this thesis. In the meantime, we also described some related architectures that have inspired our WLEC design choice. For example, we described Pannier, OpenLambda, and S2LRU* architecture details. In the last part of this section, we mentioned some standard cache replacement policies to understand our S2LRU++ approach in WLEC better.

In Chapter 5, we have provided an overview of WLEC in OpenLambda to show the placement of WLEC. Then we discussed details of the components of WLEC like S2LRU++, Template Container List, Warm Time, and Container Header. Later, we introduced the Container Management Services and its three main functions- Initialization, QueuePlacing, and OnRequest. Next, we have described the workflow and optimization of WLEC.

In Chapter 6, we have discussed Advance WLEC and its components and working methodology. We have mentioned WaLCoR and Fault Manager in detail. Then we pointed out the steps to determine Spearman's coefficient and computed the replica factor and replica factor. Later we have narrated how WaLCoR helps serverless architecture to achieve fault tolerance.

In Chapter 7, we evaluated WLEC and AdWaLEC in two different setups and discussed the results. We described the evaluation process that we have considered. Then we mentioned the experimental testbed and methodology. Next, we have discussed all metrics and their determining process that we have used for our thesis. We have compared the result of each metric to understand the behavior of WLEC and AdWaLEC in all those cases. We have added a recovery latency metric for AdWaLEC. Finally, we have examined AdWaLEC in an open-source application regarding image resizing to realize their practical effectiveness.

In Chapter 8, We have explained some open problems and compelling ideas regarding WLEC and AdWaLEC. We have discussed potential security issues and their probable solutions. Then we proposed an idea regarding stacks to replace queues of WLEC. Later, we also mentioned windows container support for WLEC to make it compatible with windows based serverless services like Azure.

References

[1] Apache openwhisk. https://openwhisk.apache.org. Date last accessed on 13-11-2022.

[2] AWS Lambda. https://aws.amazon.com/lambda/. Date last accessed on 10-11-2022.

[3] Google cloud functions. https://cloud.google.com/functions/. Date last accessed on 01-11-2022.

[4] Knative. https://knative.dev/docs/. Date last accessed on 03-11-2022.

[5] Microsoft azure functions. https://azure.microsoft.com/en-us/services/functions/. Date last accessed on 01-11-2022.

[6] OpenFaaS. https://github.com/openfaas/faas. Date last accessed on 18-11-2022.

[7] ABAD, C. L., BOZA, E. F., AND VAN EYK, E. Package-aware scheduling of FaaS functions. In *International Conference on Performance Engineering* (2018), pp. 101–106.

[8] AKHTER, A., FRAGKOULIS, M., AND KATSIFODIMOS, A. Stateful functions as a service in action. *Proceedings of the VLDB Endowment (PVLDB) 12* (August 2019), 1890–1893.

[9] AKKUS, I. E., CHEN, R., RIMAC, I., SATZKE, M. S. K., BECK, A., ADITYA, P., AND HILT, V. SAND: Towards High-Performance Serverless Computing. *The Proceedings of USENIX Annual Technical Conference (USENIX ATC '18).* (July 2018), 923–935.

[10] ALSHAYEJI, M. H., AL-ROUSAN, M., YOSSEF, E., AND ELLETHY, H. A study on fault tolerance mechanisms in cloud computing. *International Journal of Computer and Electrical Engineering 10*, 1 (March 2018), 62–71.

[11] AMAZON WEB SERVICE. Invoking Lambda Functions, 2014. https://docs.aws.amazon.com/lambda/latest/dg/invoking_LambdaF_unctions.html, Date last accessed on 29-10-2019.

[12] AMAZON WEB SERVICE. S3 events. `https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html`, 2014. Date last accessed on 01-08-2019.

[13] AMAZON WEB SERVICE. Deploying Elastic Beanstalk Applications from Docker Containers. `http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/createdeploydocker.html`, 2018. Date last accessed on 20-08-2019.

[14] ANDERSON, J. C., LEHNARDT, J., AND SLATER, N. CouchDB: The Definitive Guide. `http://guide.couchdb.org/draft/notifications.html/`, 2009. Date last accessed on 04-08-2019.

[15] APACHE. OpenLambda. `https://github.com/open-lambda/open-lambda`, 2016. Date last accessed on 09-11-2019.

[16] ARAUJO, J., MATOS, R., MACIEL, P., VIEIRA, F., MATIAS, R., AND TRIVEDI, K. S. Software rejuvenation in eucalyptus cloud computing infrastructure: A method based on time series forecasting and multiple thresholds. In *2011 IEEE Third International Workshop on Software Aging and Rejuvenation* (Hiroshima, Japan, November 2011), IEEE, pp. 38—-43.

[17] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, 2015.

[18] AZARPAZHOOH, M. R., AMIRI, A., MOROVATDAR, N., STEINWENDER, S., ARDANI, A. R., YASSI, N., BILLER, J., STRANGES, S., BELASI, M. T., NEYA, S. K., KHORRAM, B., ANDALIBI, M. S. S., ARSANG-JANG, S., MOKHBER, N., AND NAPOLI, M. D. Correlations between COVID-19 and burden of dementia: An ecological study and review of literature. *Journal of the Neurological Sciences 416* (September 2020).

[19] BAIRD, A., HUANG, G., MUNNS, C., AND WEINSTEIN, O. Serverless Architectures with AWS Lambda. In *Serverless Architectures with AWS Lambda* (2017), AWS.

[20] BALDINI, I., CASTRO, P., CHANG, K., CHENG, P., FINK, S., ISHAKIAN, V., MITCHELL, N., MUTHUSAMY, V., RABBAH, R., SLOMINSKI, A., AND SUTER, P. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing* (2017), pp. 1–20.

[21] BAMA, S. S., AHMED, M. S. I., AND SARAVANANA, A. Mathematical Approach for Mining Web Content Outliers using Term Frequency Ranking. *Indian Journal of Science and Technology 8* (July 2015), 1–5.

[22] BARCELONA-PONS, D., SUTRA, P., SÁNCHEZ-ARTIGAS, M., PARÍS, G., AND GARCÍA-LÓPEZ, P. Stateful serverless computing with CRUCIAL. *ACM Transaction on Software Engineering Methodology 31*, 3 (March 2022), 1–38.

[23] BLOCK, J. AWS-Lambda-List. `https://github.com/unixorn/aws-lambda-list`, 2018. Date last accessed on 22-10-2019.

[24] BOUIZEM, Y. An Approach for Energy-Efficient, Fault Tolerant FaaS Platforms. In *19th ACM/IFIP International Middleware Conference* (Rennes, France, December 2018), pp. 1–2.

[25] BRAVO, M., GOTSMAN, A., DE RÉGIL, B., AND WEI, H. UniStore: A fault-tolerant marriage of causal and strong consistency. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC' 21)* (July 2021), USENIX Association, pp. 923–937.

[26] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Lessons learned from three container management systems over a decade. In *Communications of ACM* (2016).

[27] CARUSO, J. C., AND CLIFF, N. Empirical size, coverage, and power of confidence intervals for spearman's rho. *Educational and Psychological Measurement 57* (1997), 637–654.

[28] CHAN, S. Everything you need to know about cold starts in AWS Lambda, 2018. `https://hackernoon.com/cold-starts-in-aws-lambda-f9e3432adbf0`, Date last accessed on 05-11-2019.

[29] CHEN, W., RAO, J., AND ZHOU, X. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. *The Proceedings of USENIX Annual Technical Conference (USENIX ATC '17).* (July 2017), 251–263.

[30] CICCONETTI, C., CONTI, M., MINGOZZI, E., AND PASSARELLA, A. Stateful Function as a Service at the Edge. *Computer 55*, 9 (2022), 54–64.

[31] COMMUNITY, P. Python Package Index, 2019. `https://pypi.org/`, Date last accessed on 16-02-2020.

[32] CORDOVA, A. Cold starting AWS Lambda functions. `https://read.acloud.guru/cold-starting-lambdas-2c663055589e`, 2018. Date last accessed on 24-08-2019.

[33] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (CA, USA, November 2006), USENIX Association, pp. 177—-190.

[34] CUI, Y. Does coding language, memory or package size affect cold starts of AWS Lambda?, 2017. `https://read.acloud.guru/does-coding-language-`

`memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76`, Date last accessed on 20-09-2019.

[35] CUI, Y. Wrong thinking about cold start, 2018. `https://theburningmonk.com/2018/01/im-afraid-youre-thinking-about-aws-lambda-cold-starts-all-wrong/`, Date last accessed on 24-10-2019.

[36] DALY, J. Lambda Warmer, 2018. `https://pillow.readthedocs.io/en/latest/`, Date last accessed on 15-08-2019.

[37] DE HEUS, M., PSARAKIS, K., FRAGKOULIS, M., AND KATSIFODIMOS, A. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21)* (NY, USA, June 2021), Association for Computing Machinery, pp. 31–42.

[38] DE HEUS, M., PSARAKIS, K., FRAGKOULIS, M., AND KATSIFODIMOS, A. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21)* (Italy, June 2021), pp. 31–42.

[39] DE HEUS, M., PSARAKIS, K., FRAGKOULIS, M., AND KATSIFODIMOS, A. Transactions across serverless functions leveraging stateful dataflows. *Information Systems 108* (September 2022).

[40] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Symposium on Operating Systems Principles(SOSP'07)* (2007).

[41] DIRKMERKEL. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal 2014*, 02 (March 2014).

[42] DIWAAKAR, L. Resolving cold start in AWS Lambda, 2017. `https://medium.com/@lakshmanLD/resolving-cold-start-804512ca9b61`, Date last accessed on 25-09-2019.

[43] FARAHABADY, M. R. H., LEE, Y. C., ZOMAYA, A. Y., AND TARI, Z. A QoS-Aware resource allocation controller for function as a service (FaaS) platform. In *International Conference on Service-Oriented Computing (ICSOC)* (Malaga, Spain, November 2017), ACM, pp. 241–255.

[44] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding,

Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. *In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (March 2017), 363–376.

[45] FUERST, A., AND SHARMA, P. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)* (NY, USA, April 2021), ACM, pp. 386–400.

[46] GIRINATHAN, J., AND BRECKINRIDGE, R. Simple Serverless Video On Demand (VOD) Workflow. `https://aws.amazon.com/blogs/networking-and-content-delivery/serverless-video-on-demand-vod-workflow/`, 2018. Date last accessed on 13-11-2019.

[47] GOTTLIEB, N. State of the Serverless-Community Survey Results, 2019. `https://www.serverless.com/blog/state-of-serverless-community`, Date last accessed on 23-10-2020.

[48] HASAN, T., IMRAN, A., AND SAKIB, K. A case-based framework for self-healing paralysed components in distributed software applications. In *The 8th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)* (Dhaka, Bangladesh, December 2014), IEEE, p. 1–7.

[49] HAUKE, J., AND KOSSOWSKI, T. M. Comparison of values of Pearson's and Spearman's correlation coefficient on the same sets of data. *Quaestiones Geographicae 30* (June 2011), 87–93.

[50] HELLERSTEIN, J. M., FALEIRO, J. M., GONZALEZ, J. E., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless computing: One step forward, two steps back. *CoRR abs/1812.03651* (2018).

[51] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing(Hot Cloud'16)* (Denver, CO, USA, June 2016), USENIX Association.

[52] HEROKU. The heroku platform. `https://www.heroku.com/platform`, 2021. Date last accessed on 14-07-2021.

[53] HONG, S., SRIVASTAVA, A., SHAMBROOK, W., AND DUMITRAS, T. Go Serverless: Securing Cloud via Serverless Design Patterns. In *10th USENIX Workshop on Hot Topics in Cloud Computing(Hot Cloud'18)* (Boston, MA, USA, July 2018), pp. 48–53.

[54] Ivanescu, A. M., Wichterich, M., and Seid, T. ClasSi: Measuring Ranking Quality in the Presence of Object Classes with Similarity Information. In *PAKDD'11: Proceedings of the 15th international conference on New Frontiers in Applied Data Mining* (2011), pp. 185–196.

[55] Jadeja, Y., and Modi, K. Cloud computing - concepts, architecture and challenges. In *International Conference on Computing, Electronics and Electrical Technologies, ICCEET* (2012), pp. 877–880.

[56] Jegannathan, A. P., Saha, R., and Addya, S. K. A time series forecasting approach to minimize cold start time in cloud-serverless platform. In *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)* (Sofia, Bulgaria, June 2022), IEEE, pp. 325–330.

[57] Jia, Z., and Witchel, E. Boki: Stateful serverless computing with shared logs. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)* (Germany, October 2021), ACM, pp. 691–707.

[58] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. J., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. Cloud programming simplified: A berkeley view on serverless computing. *CoRR abs/1902.03383* (2019).

[59] Karedla, R., Love, J. S., and Wherry, B. G. Caching Strategies to Improve Disk System Performance. In *Computer* (1994).

[60] Khandelwal, A., Tang, Y., Agarwal, R., Akella, A., and Stoica, I. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)* (RENNES, France, April 2022), USENIX Association, pp. 697—-713.

[61] Kim, K. Android Zygote and Dalvik VM, 2017. `http://davinci-michelangelo-os.com/2017/02/06/android-zygote-dalvik/`, Date last accessed on 13-12-2019.

[62] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)* (June 2014), 361–372.

[63] Kosteiko, G. How to keep AWS Lambda function containers warm. `https://stackoverflow.com/questions/51210445/`

`how_to_keep_desired_amount_of_aws_lambda_function_containers_warm`, 2018. Date last accessed on 17-10-2019.

[64] KOTNI, S., NAYAK, A., GANAPATHY, V., AND BASU, A. Faastlane: Accelerating function-as-a-service workflows. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC' 21)* (July 2021), USENIX Association, pp. 957–971.

[65] KRAFT, P., LI, Q., KAFFES, K., SKIADOPOULOS, A., KUMAR, D., CHO, D., LI, J., REDMOND2, R., WECKWERTH, N., XIA, B., BAILIS, P., CAFARELLA, M., GRAEFE, G., KEPNER, J., KOZYRAKIS, C., STONEBRAKER, M., SURESH, L., YU, X., AND ZAHARIA, M. Apiary: A dbms-backed transactional function-as-a-service framework. unpublished, July 2021.

[66] KUMAR, G., AND KUMAR, R. R. A correlation study between meteorological parameters and COVID-19 pandemic in Mumbai, India. *Journal of Diabetes & Metabolic Syndrome: Clinical Research & Reviews 14* (November 2020), 1735–1742.

[67] KUMARI, P., AND KAURU, P. A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences 33* (2021), 1159–1176.

[68] LEDMI, A., BENDJENNA, H., AND HEMAM, S. M. Fault tolerance in distributed systems: A survey. In *3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)* (Algeria, October 2018), IEEE, pp. 1–5.

[69] LI, C., SHILANE, P., DOUGLIS, F., AND WALLACE, G. Pannier: A Container-based Flash Cache for Compound Object. In *Proceedings of the 16th Annual Middleware Conference, Middleware'15* (2015), pp. 50–62.

[70] LI, J., ZHAO, L., YANG, Y., ZHAN, K., AND LI, K. TETRIS: Memory-efficient serverless inference through tensor sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC' 22)* (CA, USA, July 2022), USENIX Association, pp. 473–488.

[71] LI, W., AND KANSO, A. Comparing Containers versus Virtual Machines for Achieving High Availability. In *IEEE International Conference on Cloud Engineering* (Tempe, AZ, USA, March 2015).

[72] LI, Z., CHENG, J., CHEN, Q., GUAN, E., BIAN, Z., TAO, Y., ZHA, B., WANG, Q., HAN, W., AND GUO, M. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC' 21)* (CA, USA, July 2022), USENIX Association, pp. 53–68.

[73] LI, Z., GUO, L., CHEN, Q., CHENG, J., XU, C., ZENG, D., SONG, Z., MA, T., YANG, Y., LI, C., AND GUO, M. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '22)* (CA, USA, July 2022), USENIX Association, pp. 69–84.

[74] LIN, C., MAHMOUDI, N., FAN, C., AND KHAZAEI, H. Fine-grained performance and cost modeling and optimization for FaaS applications. *IEEE Transactions on Parallel and Distributed Systems* (January 2021), 1–20.

[75] LIN, P., AND GLIKSON, A. Mitigating cold starts in serverless platforms: A pool-based approach. In *Computing Research Repository (CoRR)* (2019), vol. abs/1903.12221.

[76] LISTON, B. Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway, 2017. `https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/`, Date last accessed on 21-11-2019.

[77] MAHGOUB, A., SHANKAR, K., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. SONIC: Application-aware data passing for chained serverless applications. In *Proceedings of the 2021 USENIX Annual Technical Conference* (Carlsbad, CA, United States, July 2021), USENIX Association, pp. 973–988.

[78] MAHGOUB, A., YI, E. B., SHANKAR, K., ELNIKETY, S., CHATERJI, S., AND BAGCHI, S. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)* (CA, USA, July 2022), USENIX Association, pp. 303–320.

[79] MAISSEN, P., FELBER, P., KROPF, P., AND SCHIAVONI, V. FaaSdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20)* (Montreal, Quebec, Canada, July 2020), pp. 73–84.

[80] MALISHEV, N. Lambda Cold Starts, A Language Comparison, 2018. `https://medium.com/@nathan.malishev/lambda-cold-starts-language-comparison-a4f4b5f16a62`, Date last accessed on 30-10-2019.

[81] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My VM is Lighter (and Safer) than your Container. *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP'17* (October 2017), 218–233.

[82] McAnlis, C. How to keep desired amount of AWS Lambda function containers warm, 2018. `https://medium.com/@duhroach/improving-cloud-function-cold-start-time-2eb6f5700f6`, Date last accessed on 16-10-2019.

[83] McGrath, G., and Brenner, P. R. Serverless Computing: Design, Implementation, and Performance. In *IEEE 37th International Conference on Distributed Computing Systems Workshops* (2017).

[84] McKinney, W. Pandas. `https://pandas.pydata.org/`, 2008. Date last accessed on 23-11-2019.

[85] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile Cold Starts for Scalable Serverless. *Proceedings of The 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19).* (July 2019), 57–67.

[86] Murat. Metadata. `http://muratbuffalo.blogspot.com/2017/05/paper-review-serverless-computation.html`, 2009. Date last accessed on 17-08-2019.

[87] NGINX. NGINX as HTTP load balancer. `http://nginx.org/en/docs/http/load_balancing.html`, 2009. Date last accessed on 06-09-2019.

[88] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Pipsqueak: Lean Lambdas with large libraries. In *IEEE Internatinal Conference Distributed Computer System Workshops(ICDCSW)* (2017).

[89] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. *the Proceedings of USENIX Annual Technical Conference (USENIX ATC '18).* (July 2018), 57–67.

[90] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 69–84.

[91] Pan, L., Wang, L., Chen, S., and Liu, F. Retention-aware container caching for serverless edge computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (London, UK, May 2022), IEEE, pp. 1069–1078.

[92] Pancham, Chaudhary, D., and Gupta, R. Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance. In *International Journal of Computer Applications* (2014).

[93] PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Fractured Processes: Adaptive, Fine-Grained Process Abstractions. In *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS '14)* (2014).

[94] RANDELL, B. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975).

[95] RANI, D., AND RANJAN, R. K. A Comparative Study of SaaS, PaaS and IaaS in Cloud Computing. *International Journal of Advanced Research in Computer Science and Software Engineering 4*, 6 (June 2014), 458–461.

[96] RAWAT, A., SUSHIL, R., AGARWAL, A., SIKANDER, A., AND BHADORIA, R. S. A new adaptive fault tolerant framework in the cloud. *IETE Journal of Research* (April 2021).

[97] RISTOV, S., HOLLAUS, C., AND HAUTZ, M. Colder than the warm start and warmer than the cold start! experience the spawn start in faas providers. In *ApPLIED '22: Proceedings of the 2022 Workshop on Advanced tools, programming languages, and Platforms for Implementing and Evaluating algorithms for Distributed systems* (Sofia, Bulgaria, July 2022), ACM, pp. 35–39.

[98] SARATHY, V., NARAYAN, P., AND MIKKILINENI, R. Next Generation Cloud Computing Architecture: Enabling Real-Time Dynamism for Shared Distributed Physical Infrastructure. *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'10)* (June 2010), 48–53.

[99] SERVICE, A. W. AWS Metrics, 2010. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html`, Date last accessed on 21-02-2020.

[100] SERVICE, A. W. Improving startup performance with Lambda SnapStart. `https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html`, 2022. Date last accessed on 18-12-2022.

[101] SETTY, S., SU, C., LORCH, J. R., ZHOU, L., CHEN, H., PATEL, P., AND REN, J. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (GA, USA, November 2022), USENIX Association, pp. 501–516.

[102] SHAHRAD, M., FONSECA, R., ÍÑIGO GOIRI, CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provide.

In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '22)* (CA, USA, July 2020), USENIX Association, pp. 205–218.

[103] SHILLAKER, S., AND PIETZUCH, P. FAASM: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC' 20)* (CA, USA, July 2020), USENIX Association, pp. 419–433.

[104] SILVA, P., FIREMAN, D., AND PEREIRA, T. E. Prebaking functions to warm the serverless cold start. In *21st International Middleware Conference (Middleware '20)* (Delft, Netherlands, November 2020), ACM, pp. 1–35.

[105] SOLAIMAN, K., AND ADNAN, M. A. WLEC: A not so cold architecture to mitigate cold start problem in serverless computing. In *IEEE International Conference on Cloud Engineering (IC2E)* (Sydney, Australia, April 2020), IEEE, pp. 144–153.

[106] SPEARMAN, C. The proof and measurement of association between two things. *American Journal of Psychology 15* (1904), 72–101.

[107] SPENGER, J., CARBONE, P., AND HALLER, P. Portals: An extension of dataflow streaming for stateful serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22)* (Auckland, New Zealand, December 2022), ACM.

[108] SREEKANTI, V., WU, C., CHHATRAPATI, S., GONZALEZ, J. E., HELLERSTEIN, J. M., AND FALEIRO, J. M. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)* (New York, USA, April 2020), pp. 1–15.

[109] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIERSMITH, J., FALEIRO, J. M., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment (PVLDB) 13*, 11 (2020), 2438–2452.

[110] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'01* (2001), pp. 149–160.

[111] SURENDER, CHAUHAN, R., AND KUMAR, P. Hierarchical non-blocking coordinated checkpointing algorithms for mobile distributed computing. *International Journal of Computer Science and Security (IJCSS) 3* (2010), 518–524.

[112] TARSITANO, A. Comparing The Effectiveness Of Rank Correlation Statistic. Tech. rep., Università della Calabria, Dipartimento di Economia, Statistica e Finanza ”Giovanni Anania”- DESF, Italy, 2009.

[113] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS Containers. In *Proceedings of 2018 USENIX Annual Technical Conference, USENIX ATC'18* (2018), pp. 199–212.

[114] THORPE, J., QIAO, Y., EYOLFSON, J., TENG, S., HU, G., JIA, Z., WEI, J., VORA, K., NETRAVALI, R., KIM, M., AND XU, G. H. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)* (July 2021), USENIX Association, pp. 495–514.

[115] THUNDRA. Thundra: Serverless Observability for AWS Lambda. `https://docs.thundra.io/`, 2018. Date last accessed on 06-09-2020.

[116] TIAN, H., LI, S., WANG, A., WANG, W., WU, T., AND YANG, H. OWL: Performance-aware scheduling for resource efficient function-as-a-service cloud. In *Symposium on Cloud Computing (SoCC '22)* (San Francisco, CA, USA, November 2022), ACM.

[117] TOSEPU, R., GUNAWAN, J., EFFENDY, D. S., AHMAD, L. O. A. I., LESTARI, H., BAHAR, H., AND ASFIAN, P. Correlation between weather and Covid-19 pandemic in Jakarta, Indonesia. *Journal of Science of The Total Environment 725* (July 2020).

[118] TOTOY, G., BOZA, E. F., AND ABAD, C. L. An Extensible Scheduler for the OpenLambda FaaS Platform. In *Min-Move'18, ACM* (2018).

[119] VAN KESTEREN, A. Xmlhttprequest. `https://xhr.spec.whatwg.org/`, 2004. Date last accessed on 05-07-2019.

[120] VERMA, A., AHUJA, P., AND NEOGI, A. pMapper: Power and migration cost aware application placement in virtualized systems. In *Middleware: ACM/IFIP/USENIX 9th International Conference on Distributed Systems Platforms and Open Distributed Processing* (Leuven, Belgium, December 2008), USENIX Association, pp. 243—-264.

[121] WAGNER, T. Understanding container reuse in AWS Lambda. `https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/`, 2014. Date last accessed on 02-11-2019.

[122] WALSH, L., AKHMECHET, V., AND GLUKHOVSKY, M. RethinkDB — Rethinking Database Storage. In *Hexagram 49* (2009).

[123] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC' 21)* (July 2021), USENIX Association, pp. 443–457.

[124] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of 2018 USENIX Annual Technical Conference, USENIX ATC'18* (2018), pp. 133–145.

[125] WANG, S., LIAGOURIS, J., NISHIHARA, R., MORITZ, P., MISRA, U., TUMANOV, A., AND STOICA, I. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)* (Huntsville, Ontario, Canada, October 2019), pp. 338–352.

[126] WANG, X., WU, Q., AND ZHANG, Y. T-db: Toward fully functional transparent encrypted databases in dbaas framework. *ArXiv abs/1708.08191* (2017).

[127] WEINHARDT, P. D. C., ANANDASIVAM, W. A., BLAU, D. B., BORISSOV, N., MEINL, T., MICHALK, W. W., AND STÖSSER, D. J. Cloud Computing – A Classification, Business Models, and Research Directions. *Business & Information Systems Engineering 1*, 5 (October 2009), 391–399.

[128] WEN, J., WANG, Y., AND LIU, F. StepConf: SLO-aware dynamic resource configuration for serverless function workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (London, UK, May 2022), IEEE, pp. 1868–1877.

[129] WIKIMEDIA. Cache replacement policies. `https://en.wikipedia.org/wiki/Cache_replacement_policies`, 2010. Date last accessed on 15-08-2019.

[130] WIKIMEDIA. User defined function. `https://en.wikipedia.org/wiki/User-defined_function`, 2012. Date last accessed on 15-10-2019.

[131] XU, F., QIN, Y., CHEN, L., ZHOU, Z., AND LIU, F. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers 71* (January 2021), 450–463.

[132] XU, Z., ZHANG, H., GENG, X., WU, Q., AND MA, H. Adaptive function launching acceleration in serverless computing platforms. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)* (Tianjin, China, December 2019), IEEE, pp. 9–16.

[133] YILMAZ, E., ASLAM, J. A., AND ROBERTSON, S. A New Rank Correlation Coefficient for Information Retrieval. In *The 31st Annual International ACM SIGIR Conference* (Singapore, July 2008), pp. 587–594.

[134] Yu, H., Irissappane, A. A., Wang, H., and Lloyd, W. J. FaaSRank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (Washington, DC, USA, September 2021), IEEE, pp. 31–40.

[135] Yu, M., Cao, T., Wang, W., and Chen, R. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 23)* (MA, USA, April 2023), USENIX Association. accepted to be appeared.

[136] Zhang, H., Cardoza, A., Chen, P. B., Angel, S., and Liu, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Banff, Alberta, Canada, November 2020), pp. 1187–1204.

[137] Zhang, Y., Zheng, Z., and Lyu, M. R. Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing. In *2011 IEEE 4th International Conference on Cloud Computing* (Washington, DC, USA, July 2011), IEEE, pp. 444–451.

[138] Zhou, D., and Tamir, Y. HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay. *Computing Research Repository (CoRR) abs/2101.0958* (January 2021).

[139] Zhou, Y., Chen, Z., and Li, K. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference(USENIX ATC'01)* (2001).

[140] Zhuang, S., Li, Z., Zhuo, D., Wang, S., Liang, E., Nishihara, R., Moritz, P., and Stoica, I. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21)* (USA, August 2021), ACM, pp. 641–656.

List of Publication

[1] SOLAIMAN, K., and ADNAN, M.A., “WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem for Serverless Computing,” *IEEE International Conference on Cloud Engineering(IC2E)*, (April, 2020), Sydney, Australia, pp. 144-153.

Index

Appendix A

Algorithms

A.1 Algorithm for Initialization function

In Algorithm 1 we show how to initialize a new container in WLEC

Algorithm 1 Algorithm for container initialization

```
$create$ container $c$
$hit \longleftarrow hit$++
$q_{type} \longleftarrow 0$
$w_{time} \longleftarrow now() + a_{time}$
$Q_c.push(c)$
**return** c
```

A.2 Algorithm for OnRequest function

In Algorithm 2 we show how WLEC selects a container when a request is issued

A.3 Algorithm for QueuePlacing function

In Algorithm 3 we show how WLEC determines the placement of a container after a request in served by that container

Algorithm 2 Algorithm for container selection

```
if request is valid then
    for each $c \in Q_w$ do
        if $c.f_{type} = \lambda_{type}$ and $c.status = 0$ then
            $c.hit \longleftarrow c.hit$++
            return $c$
        else if $c.f_{type} = \lambda_{type}$ and $c.status = 1$ then
            $c.Initialization()$
            $L_t.LRU \longleftarrow c$
            return $c$ from $L_t$
        end if
    end for
    for each $c \in Q_c$ do
        if $c.f_{type} = \lambda_{type}$ then
            $c.hit \longleftarrow c.hit$ ++
            return $c$
        end if
    end for
    $c.Initialization()$
    $c.hit \longleftarrow c.hit$++
    return $c$
else
    invalid request
end if
```

Algorithm 3 Algorithm for container placement

```
if $c.hit = 0$ or $c.hit < t_{value}$ then
    $Q_c.LRU \longleftarrow c$
    $c.w_{time} \longleftarrow now() + a_{time}$
else if $c.hit < t_{value}$ and $q_{type} = 0$ then
    $Q_c.MRU \longleftarrow c$
    $c.w_{time} \longleftarrow now() + a_{time}$
else if $c.hit > t_{value}$ and $q_{type} = 0$ then
    $Q_c.pop()$
    $Q_w.MRU \longleftarrow c$
    $c.w_{time} \longleftarrow now() + a_{time}$
    $L_c \longleftarrow c$
else if $c.a_{time} < now()$ and $q_{type} = 1$ then
    $Q_c.MRU \longleftarrow c$
    $c.w_{time} \longleftarrow now() + a_{time}$
else
    no change
end if
```