Efficient Security for Resource-Constrained Devices with LPM: A Lightweight Protocol Utilizing Chaotic Map-Based S-Box

K. Ramesh[1] and K. Ramesh[1]

[1]National Institute of Technology Warangal

January 31, 2024

A Framework for Migration of Microservices based Applications to Serverless Platform with Efficient Cold Start Latency

Vinay Raj, *Member, IEEE,*

***Abstract*—Serverless computing has become a very popular cloud-based solution for designing, testing and deploying the applications as serverless functions. Serverless computing has become a buzzword in both industry and academia as many large IT giants have rolled out their own serverless platforms. Independent design, deployment and auto scalability are its major features with Function-as-a-Service (FaaS) as its popular implementation. On the other hand, microservices are an emerging trend for design of large enterprise applications as many companies like Amazon, Uber and Spotify have already migrated their existing applications to microservices style. However, with the advent of serverless platforms, to design efficient applications with simpler infrastructure management, reduced operational overhead and cost factor, migrating microservices to serverless has become inevitable. Additionally, existing studies report that serverless platforms suffer from cold start latency. Therefore, in this paper, a framework is proposed which has two phases: (i) an empirical investigation to find out the impact of programming language on the cold start of serverless functions and (ii) an approach for migration of microservices to serverless platforms. The evaluation results help us to identify the platform with low cold start latency and also recommend the choice of programming language with lower latency. After migrating the containerized microservices to serverless, a comparison in terms of performance is conducted. The applications designed as serverless functions exhibit better response time and throughput compared to containerized microservices.**

***Index Terms*—Serverless computing, microservices, cold start, function-as-a-service, performance.**

I. Introduction

Distributed systems have evolved rapidly from a monolithic style of client-server applications to today's trending serverless architectures. The demand for quick design and deployment of services with low downtime of the applications has made the evolution of different architectural styles such as Service Oriented Architecture (SOA), microservices and serverless architectures [1]. SOA has been the most popular style for designing large enterprise applications with the implementation of web services. It has been one of the successful architectural styles for designing internet based applications post the era of internet. However, the tight coupling with Enterprise Service Bus (ESB) and the services in SOA tending towards monolithic in size led to the evolution of the new architectural style, microservices [2]. Microservices definition clearly states that every service should implement only one business goal, i.e., every service should follow the Single Responsibility Principle (SRP) [34]. The major benefit of microservices over SOA is that the services are deployed in cloud containers instead of traditional servers at the developer end. The cloud containers are lightweight and they have the feature of auto scalability based on the demand of user requests. Currently, microservices is one of the trending styles of software design because of the diverse benefits and advantages of cloud platforms.

On the other hand, serverless computing is an emerging and potential cloud computing paradigm and it has gained popularity in recent times for its constantly available servers driving the web application behavior owing to its large influence in reducing costs of computing and storage space, decreasing latency, improving scalability, and eliminating server-side management [5]. The main intention behind the concept of serverless computing is to completely abstract away servers from the developers. The term serverless has a misconception that the servers are not available which is incorrect. The actual servers exist and all the development, testing and deployment is performed by the developer but at the cloud provider [55]. The complete infrastructure, hardware and software configurations are taken by the provider and the developer just needs to have good internet connection. In this computing, developers merely need to write functions in high-level languages such as Java, Python etc., define a few simple attributes, then upload these functions to a serverless platform. The resultant API or HTTP requests might then be used to conduct their well-defined computing tasks [6], [14]. In contrast to serverful computing models, developers employing serverless do not need to worry about managing infrastructure resources because platforms manage such things on their behalf. The heart of serverless computing is the cloud functions which are written by developers and invoked as units of execution via the Internet [56]. Serverless computing may also be seen from the perspective of developers as Function-as-a-Service (FaaS), which enables developers to create and execute their applications (or functions) without having to deal with the complexity of creating and managing the underlying infrastructure [57]. On the other hand, serverless service providers always provide their clients Backend-as-a-Service (BaaS), or application-dependent services, including Database and Object Storage Service (OOS). Accordingly, from a broad perspective, serverless computing is the merging of BaaS and FaaS to provide clients with a single service model [8].

Despite of all these benefits of serverless computing, the serverless functions suffer from a problem called cold start

Vinay Raj is with the Department of Computer Applications, National Institute of Technology Tiruchirappalli, Tamil Nadu, India (email: vinayraj@nitt.edu)

Manuscript received October 03, 2023;

problem [59]. It refers to the initial startup time of the functions or the containers after moving into idle state. If there are no requests to the serverless functions, after a certain time period, the functions are pushed into cold state. This is one of the features of serverless with pay as you go pricing model. Every provider has their own time limit for pushing the functions into idle state. Whenever a new request comes to the FaaS applications, it takes little more time to load all its dependencies, runtime initialization, and other factors such as code package size, programming language etc. have an impact on the cold start. It has become a challenge for the clients as the cold start time is very high for the FaaS applications. Therefore, in this work, an empirical investigation to find out whether programming language has any impact on the cold start is also presented.

Generally, microservices are deployed in cloud containers using the Docker software [22]. However, with the advent of serverless platforms, it has become the alternative for deployment of microservices [9], [26]. In a recent study, it has been found that the performance of microservices is better when deployed in serverless platforms compared to containerized microservices [21], [13]. Additionally, microservice architectural style is a popular alternative for creating applications for cloud since the updation, scaling and upgrading can be done individually for each microservice. Considering these facts, application development and cloud infrastructure management were merged into what is today known as DevOps. However, there is growing interest in using FaaS and serverless CaaS technologies for refactoring microservices-based applications because of the attention the serverless computing technology has gained and its many benefits, including no infrastructure management, a pay-per-use billing policy, and on-demand fine-grained autoscaling [23].

Because of the significant advantages of serverless computing, it has earned positive attention in the industry. In a recent survey, it is expected that the global industries will adopt serverless platforms by 2025 [58]. There has been a transition from microservices deployment in a containerized architecture to a serverless architecture over time. Despite these advantages, the time and money necessary to modify existing code restricts the accessibility of these applications and impedes attempts to develop serverless computing platforms [15]. Supporting the migration of current applications to serverless platforms will help developers while also broadening the scope of serverless computing. Furthermore, there is no effective algorithm for migrating microservices-based applications to serverless automatically [16]. However, migrating systems to serverless involves a number of challenges, including (i) not understanding the impact of migration, (ii) not having enough material on automated migration strategies [24], [27]. These difficulties prompted us to research and develop approaches for migrating Microservices-based applications to a serverless architecture.

A. Contributions

To the best of our knowledge, this is one of the first attempts to empirically assess the impact of programming languages on cold start latency of serverless platforms and also, migrating the existing microservices based applications to serverless platform. To summarize, the following contributions are made to achieve the proposed objective.

- An empirical investigation on whether choice of programming language has any impact on cold start of the serverless platforms.
- Identification of best programming language for serverless platforms with low cold start latency.
- An approach to automatically migrate the containerized microservices to serverless platform.
- Performance evaluation of containerized microservices and serverless applications.

The remaining part of the paper is organized as follows: Section II presents the preliminaries required for the proposed framework. Section III discusses the related work in comparison with the proposed framework and proposed approach is presented in Section IV. The empirical investigation of cold start in three different platforms is presented in Section V and migration of microservices to serverless is presented in Section VI. Section VII concludes the paper.

II. Background

In this section, the preliminaries required to understand the proposed framework are presented. Since, the primary aim of this paper is about migrating microservices based applications to serverless, both the architecture styles are presented along with the cold start problem in serverless platforms.

A. Microservices Architecture

The term microservices was first coined by Lewis and Fowler in the year 2014. The microservice architectural style [37] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. Microservices are designed and deployed independently with the principles of CI and CD. The microservices follow single responsibility principle which states that every service in the application should perform only one business task [38]. These services are deployed in cloud containers which have the inbuilt functionality of auto-scalability. The messages are exchanged among the microservices using advanced communication protocol such as HTTP/REST and JSON. The microservices developer have the freedom to choose any programming languages for designing microservices as it supports polyglot feature [39]. The applications uses API gateway for communication between services and each service has a different data storage. Many tools are involved in life cycle of microservices such as Docker for creating container images and Kubernetes for deploying and monitoring the microservices [60].

Because of these diverse benefits of microservices architectural style, many large IT firms have migrated their existing applications to this style and lot of research is going on in migrating existing monolithic and SOA based applications to microservices style [40]. From the literature, it is very clear that microservices is better than existing solutions in terms of

QoS parameters [41], [42], [43]. However, with the advent of serverless platforms and their path breaking benefits among all the available software architectures, it gives a new direction for using the advantages of serverless architectures.

B. Serverless Computing

Serverless computing is a prominent cloud computing model that has been used in a variety of disciplines such as machine learning [44], scientific computing [45], and video processing [46]. Serverless computing is expected to be used by 50% of global organizations by 2025 [47]. FaaS apps are created by developers who construct their applications as a mix of serverless services. The underlying serverless systems manage resources autonomously. As a result, developers do not need to manage servers or VM instances in order to operate FaaS apps. Instead, when events (such as an HTTP request) activate serverless functions, resources are dynamically assigned to them. If a serverless function is not utilized for an extended period of time, the platform will release the resources. This allows for lightweight and effective management of resources. Serverless functions and their dependent libraries are packed into a single bundle in FaaS applications and then delivered to serverless platforms. If the app's space crosses the deployment limit (for example, 250 MB uncompressed size on AWS Lambda), designers can launch apps utilising container images with greater sizes [17]. After the deployment is successful, serverless operations will be triggered by predetermined events, such as an HTTP request, a file change in cloud storage, or the activation of a timer. When serverless functions are invoked, the serverless platform dynamically assigns and starts dedicated function instances (e.g., VMs or containers) with limited resources (e.g., CPU and memory) to carry out their duties. When no new requests are received, launched instances and resources are automatically retired.

The execution of a serverless function can occur in two modes: cold start and warm start [49]. If the called function has not been utilized for a certain amount of time (keep-alive time), its invocation is in cold-start mode. In this state, the serverless platform needs to set up new VMs or containers, transfer the function code from distant cloud storage to instances over the network, load the necessary code to begin the application process, and ultimately run the serverless function. If, on the other hand, the called function has recently been utilised, the invocation is in warm start mode, in which the serverless platform uses previously launched instances of the same function. However, the cold start latency has become a challenge in serverless platforms.

1) Cold Start Latency: The time it takes for a computer program or application to start and become responsive after it has been idle or not utilised for a particular length of time is referred to as cold start latency [50]. This phrase is frequently used in conjunction with serverless computing systems and containerized applications. Functions in serverless computing systems are run in response to events (for example, HTTP requests). A cold start delay may occur when a function is initiated for the first time or after a period of inactivity. The time it takes to allocate resources, load dependencies, and execute code is included in this delay. In these contexts, reducing cold start latency is critical for improving user experience and optimising resource utilisation. To reduce cold start latency, several tactics and approaches may be used, such as pre-warming resources, employing smaller and more efficient runtime environments, and optimising code and dependencies.

There are multiple factors causing the cold start in the serverless platforms such as choice of programming language, code size, dependencies required for running the serverless functions, and memory space etc. In this work, an experimental study for doing empirical analysis is presented to verify whether the choice of programming has any impact on the cold start time. Additionally, a recommendation is given to the serverless developers on selecting the suitable programming language for a particular platform.

2) Serverless Providers: Serverless solutions are used for a variety of applications, including web application development, microservices, real-time data processing, and more. They are well-known for their scalability since resources are automatically assigned depending on demand, and developers are only paid for the compute resources utilised during function execution. There are many companies which provide serverless platforms such as AWS Lambda, Microsoft Azure, Google Cloud Platform (GCP), IBM cloud functions etc [51]. In this study, three popular platforms: AWS Lambda, Azure and GCP are considered for empirical analysis.

- *AWS Lambda*: One of the first serverless systems was Amazon Web Services (AWS) Lambda. It enables you to execute code in reaction to AWS events like as HTTP requests, database updates, or file uploads. Lambda is compatible with a variety of programming languages and interfaces with other AWS services.
- *Azure functions*: Microsoft Azure offers this serverless facility. It supports a variety of event triggers, including HTTP requests, Azure Storage events, and updates to Azure Cosmos DB. Functions can be written in languages such as C#, Node.js, Python, and others.
- *Google cloud functions*: Google Cloud Functions is Google's serverless platform, which enables developers to run code in response to HTTP requests, Cloud Pub/Sub events, and more. It supports languages such as Node.js, Python, and Go.

III. Related Work

The existing literature on both microservices and serverless computing have focused more on the software engineering activities of the applications designed using those styles. The research on both microservices and serverless is trending as both the styles have emerged recently. In this section, the works related to cold start and its evaluation and approaches for migration to serverless are discussed.

A. Studies on cold start assessment

A detailed empirical evaluation of performance, scalability and cold start latency is performed in [33] with respect to three programming languages: Go, Python and Java. The experiments are performed only in AWS Lambda and the results state that Go language exhibited very low cold start compared to other languages. However, the limitation of this work is that it is limited only to three programming languages and tested only in AWS. In our work, all the programming languages supported by each platform and all three major serverless platforms are considered. A test suite to evaluate the performance of applications in AWS, Azure, GCP and IBM is presented in [34]. Similar to our proposed work, the authors have used applications of different programming languages to estimate the performance parameter. However, the focus of our work is to find out the cold start latency of applications written using different languages. Two other similar studies [35], [36] were conducted to estimate the performance, start-up delays and latency. However, the works are either limited to two programming languages or have been experimented only on two serverless providers. There exists many works in the literature which primarily focuses on mitigating the cold start latency in different platforms. Each existing work proposes a different and unique technique to over the cold start challenge in the serverless platforms. However, one of our important contribution is to recursively test the cold start in three major serverless providers: AWS Lambda, Azure and Google Cloud with many applications written using all the programming languages supported by each platform.

B. Studies on migration to serverless

The authors in [10] have presented the challenges in implementation of serverless applications and suggested few possible solutions for such problems. Also, automated migration process is highlighted as open problem. An automated process for migration of FaaS from one serverless to another platform is highlighted as a challenge. In [11], the authors present an empirical analysis of how migrating to serverless reduced the hosting costs between 66 to 95%. It speeds up time to market for delivery of new features. However, the authors have selected legacy monolithic applications for migration to serverless. The authors present how a FinTech application is migrated to serverless in [12] and also the performance analysis of pre and post migration of the application are presented. A tool, ToLambda, for automatic conversion of Java monolith application code into AWS Lambda Node.js microservices is proposed in [17]. A semi automated approach for migrating monolithic to microservices is proposed in [31] where different approaches for transforming code are presented. In all the above approaches for migrating to serverless have focused on migrating legacy monolithic applications. However, our focus is on migrating microservices based applications to serverless. Hence, in our work, an automated approach for migrating microservices to serverless is proposed.

A similar work of migrating microservices to serverless is proposed in [18] where a complex IoT platform application based on microservices is migrated to OpenWhisk (OW) and Google Cloud Run (GCR). However, the proposed approach is specific to only OW and GCR. The authors in [30] have presented a comparative study between different serverless platforms in terms of cost and performance by deploying a microservices application. It is also highlighted that there is a need for automatic approaches for migrating microservices to serverless and also migrating the applications from one serverless to other platforms. They have not proposed any mechanism for migration which is considered as a major contribution in our work. A partial migration of monolithic application to microservices and serverless platform is proposed in [32]. However, the serverless platform is only used to deploy the application. Also, finding an optimal automatic migration solutions for existing legacy systems is an interesting research direction [3]. Additionally, only a few applications have been migrated to serverless serverless either because of no proper mechanism for migration or lack of awareness of the benefits of serverless. In our work, an automated and generalized approach is proposed to migrate to any serverless platforms.

IV. APPROACH

In this section, the proposed framework for migration of microservices based applications to serverless platform is presented. The framework includes different tasks: (i) Empirical investigation of cold start latency and (ii) Migration of microservices based applications to best serverless platform. The benchmark applications selected for performing the empirical investigation to find out the relation between the cold start latency and its dependency on the programming language are identified and are presented in below sections. Also, the microservices based case study application considered for migration to serverless is also presented.

A. Proposed Framework

A framework includes all the set of guidelines, tools and procedure to design an application. The diagram in Figure 1 shows the detailed architecture of the proposed framework. It includes two phases: In the first phase, FaaS applications are deployed in three serverless platforms. Each application is written using all the programming languages supported by the serverless provider. Once the deployment is completed, each FaaS application is allowed to switch to idle state after it completes the waiting period. Then, cold start latency is captured for each application in all three platforms with respect to the programming language. The cold start latency are analyzed and mapped corresponding to the programming language of the platform.

After the first phase, the programming language with low cold start latency in each platform is identified. The platform with efficient cold start latency is considered for migration of existing serverless platform. Though microservices perform better compared to other existing architectural styles, due to the advantages of serverless in terms of cost of the infrastructure, its maintenance and auto scaling feature, it triggers to migrate existing microservices to serverless platforms. Migrating microservices to serverless is a complex process as services need to fit as per the serverless environment, i.e. FaaS needs to be designed from microservices code. It is required to break the microservices into smaller and functionally independent services called functions. However, in this work, it is assumed that the microservices application strictly follows the single

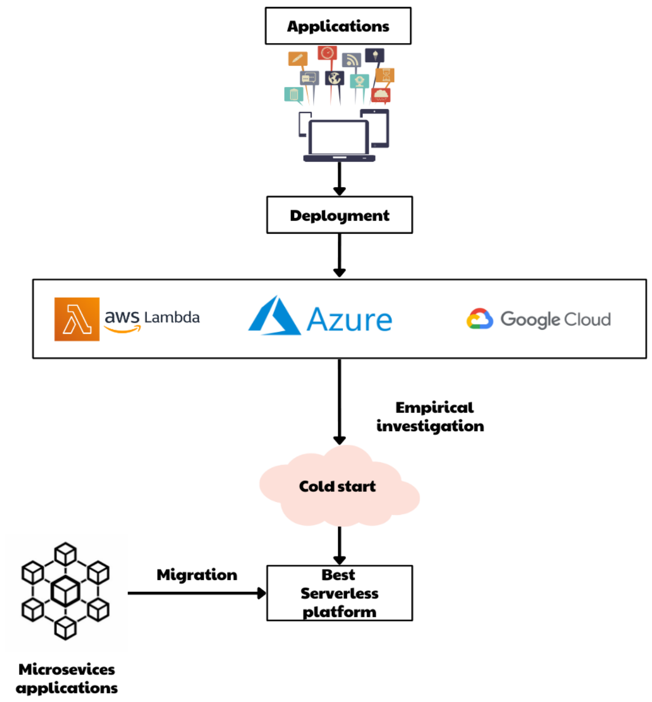


Fig. 1: Proposed framework diagram

responsibility principle and each services performs only one business task. Therefore, the microservices can be directly considered as serverless functions and it requires to update the environmental configuration settings. However, microservices and serverless can be used together by making few services as serverless functions to handle some tasks of the business requirement.

B. Benchmark Applications for Cold Start Estimation

To perform the empirical analysis, the applications written as serverless functions, also known as FaaS are chosen. In specific, ten different classical computer science problems which are written as serverless functions are considered. Since, this is one of the initial studies in identifying the cold start latency with different programming language, there is no specific criteria in identifying the applications except that they all perform different computational tasks. The details of the programs along with the naming convention used in the paper are presented in Table I.

App No	Name of the Function
App1	Binary Search
App2	Bubble Sort
App3	Fibonacci Series
App4	Round Robin
App5	Random Number Generator
App6	Dijkstra's algorithm
App7	Matrix Chain Multiplication
App8	Travelling Salesman Problem
App9	N-Queen's Problem
App10	Encryption Decryption Algorithm

TABLE I: List of serverless functions and corresponding notation

C. Benchmark Application for Migration to Serverless

A simple Node.js microservices application is considered to be migrated to serverless architecture. The application is a To-Do Manager which is divided into a set of services specialized in doing specified tasks using a certain set of protocols. The services communicate with each other over a network. A diagram showing all the microservices involved is shown below in Figure 2.

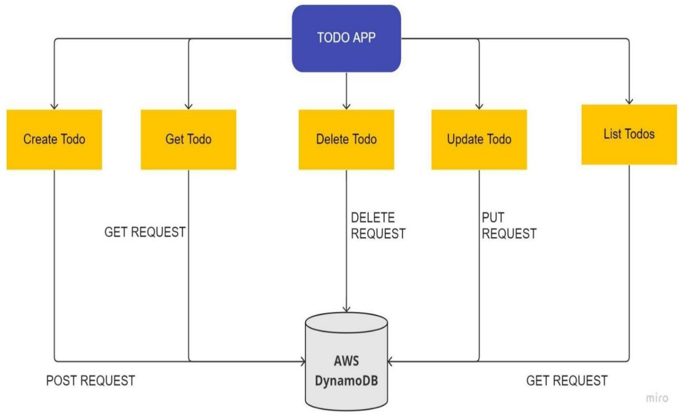


Fig. 2: Microservices of a To-Do application

The modules in the Figure 2 illustrates main functionalities of the application. However, there are multiple services involved in each of these modules which are running in separate containers like Frontend, Database, Authentication services etc. For easy understanding and implementation of the application, a demo skeleton of a basic microservice is created which includes different functions as listed below.

1) **Hello**: This represents just a home page of the application.
2) **Create Todo**: A function to create a new to-do in the database by user.
3) **Get Todo**: A function that helps to fetch any existing to-do from the database.
4) **Delete Todo**: This function allows users to delete any existing to-do, permanently erasing from the database.
5) **Update Todo**: This function allows to update any kind of details for created to-do's and store the updated details back to the database.
6) **List Todos**: This function lists down the complete set of created to-do's fetching them from the database.

V. Impact of Programming Language on Cold Start

In this section, an empirical investigation to find out whether programming language has any impact on the cold start of the functions in serverless platforms is presented. Cold start latency refers to the time it takes to start up and respond to the first request after moving into idle state. There are many factors with respect to the programming language which influence the cold start time such as initialization time, runtimes, and the size of the program code. However, programming language alone is not a factor for latency in serverless platforms. In this work, the prime focus is only on to identify the choice of programming language for the serverless platform which has less cold start latency.

For this investigation, three popular serverless platforms, AWS Lambda, Azure and GCP are considered. Every platform supports multiple programming languages to design the FaaS applications and developers can choose any language of their choice. In each platform, ten FaaS applications listed in Section IV are developed in all the programming languages supported and tested for cold start time. For detailed analysis, total five invocations are given to the functions to observe how the latency changes from first request to the first request. All the details of the investigation for each platform is presented in below sections.

A. AWS Lambda

Amazon Web Services (AWS) offers Lambda, a serverless computing service. It allows you to execute code in response to certain events without the need for server or infrastructure management. Lambda is a core component of serverless computing that is intended to be highly scalable, affordable, and simple to use [52]. It supports multiple programming languages, including Node.js, Java, Python, Go, Ruby and .NET. For the chosen FaaS applications, four popular programming languages are chosen namely, Python, Ruby, Node.js and Java. All the ten applications are designed and tested for cold start. The details of the investigations of ten apps of different languages are presented in Table II.

The outcome of the experiments performed on AWS Lambda for the chosen ten applications which are written in different programming languages is shown graphically in Figures 3 to 12.

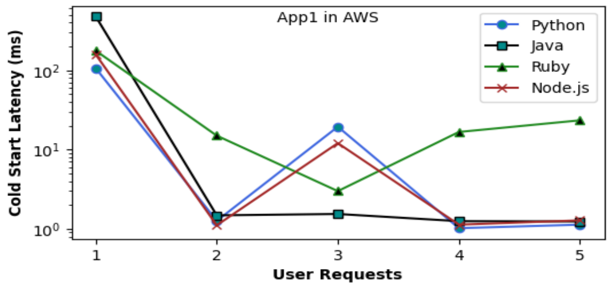


Fig. 3: Cold start latency of App1 in AWS

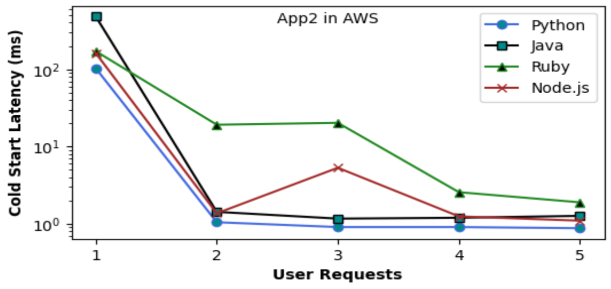


Fig. 4: Cold start latency of App2 in AWS

TABLE II: Cold start latency in AWS

App No	Request	Python	Ruby	Node.js	Java
App1	1	106.06	176.03	157.89	476.07
	2	1.26	14.95	1.11	1.48
	3	19.37	2.99	12.03	1.54
	4	1.02	16.64	1.13	1.25
	5	1.13	23.32	1.28	1.24
App2	1	103.28	171.44	162.27	482.9
	2	1.05	19.17	1.37	1.43
	3	0.91	20.28	5.32	1.17
	4	0.91	2.57	1.25	1.2
	5	0.88	1.9	1.1	1.27
App3	1	108.54	168.78	140.55	377.45
	2	1.23	14.96	2.14	1.99
	3	1.17	2.84	1.05	1.39
	4	0.95	2	0.98	1.32
	5	0.89	2.13	1.15	1.32
App4	1	109.61	186.33	157.67	497.1
	2	0.93	12.06	2.41	1.54
	3	0.9	1.99	1.2	1.42
	4	0.74	1.83	1.14	1.45
	5	0.97	5.17	1.1	1.42
App5	1	106.16	170.67	175.67	366.96
	2	1.03	12.76	19.72	1.16
	3	1.22	44.32	96.8	1.19
	4	1.65	4.71	53.7	1.02
	5	1.14	2.34	1.78	1.3
App6	1	388.56	220.53	142.2	417.66
	2	1.04	11.34	1.21	2.17
	3	1.39	1.45	160.51	1.58
	4	1.04	1.32	1.13	1.45
	5	1.28	0.98	1.18	1.34
App7	1	106.64	192.81	169.46	359.43
	2	1.06	9.63	15.47	1.05
	3	1.1	2.67	108.39	1.14
	4	0.95	1.4	43.85	1.17
	5	0.98	1.03	1.1	1.14
App8	1	106.12	226.04	175.97	484.59
	2	3.91	18.1	23.92	1.2
	3	1	8.33	113.83	1.22
	4	0.96	2.87	48.1	1.32
	5	1.02	1.2	1.1	1.16
App9	1	105.59	170.76	139.59	466.4
	2	1.1	18.5	1.32	2.61
	3	1.08	2.71	0.99	1.56
	4	1.03	2.88	1.54	1.63
	5	0.88	1.91	2.1	1.55
App10	1	148.12	168.37	162.36	417.72
	2	1.88	10.19	1.19	1.98
	3	1.04	18.43	57.86	1.68
	4	0.97	17.79	1.26	1.63
	5	0.87	2.03	1.18	1.59

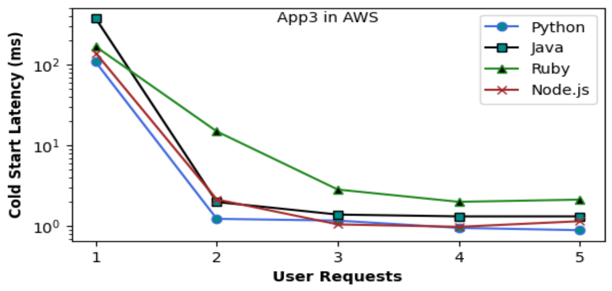


Fig. 5: Cold start latency of App3 in AWS

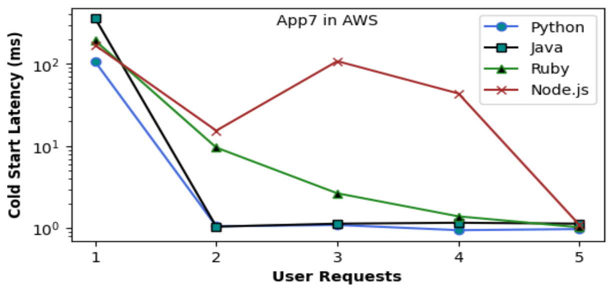


Fig. 9: Cold start latency of App7 in AWS

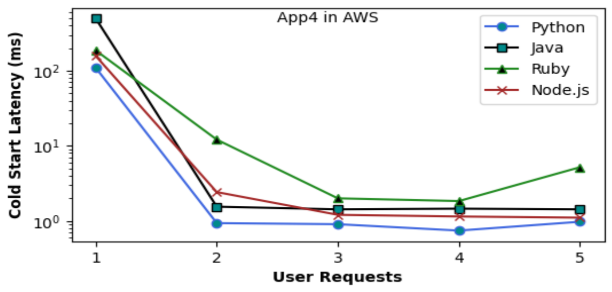


Fig. 6: Cold start latency of App4 in AWS

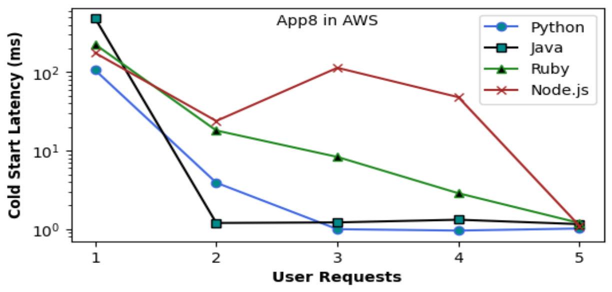


Fig. 10: Cold start latency of App8 in AWS

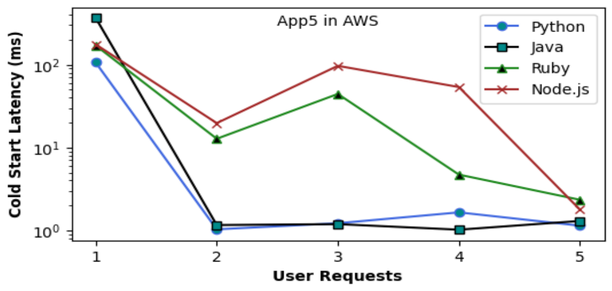


Fig. 7: Cold start latency of App5 in AWS

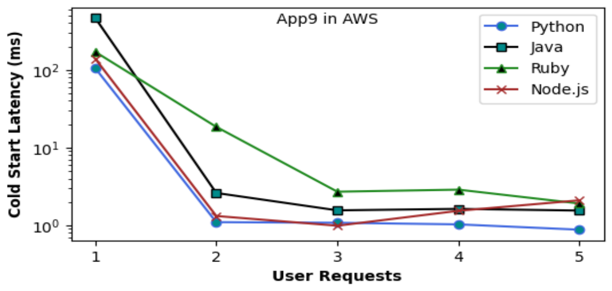


Fig. 11: Cold start latency of App9 in AWS

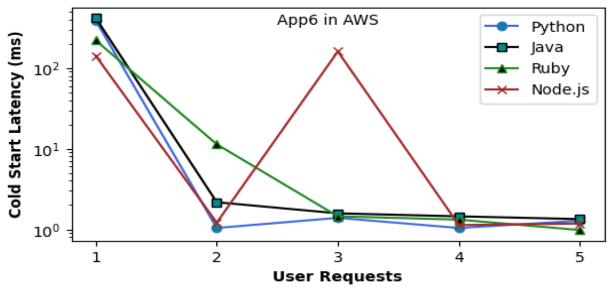


Fig. 8: Cold start latency of App6 in AWS

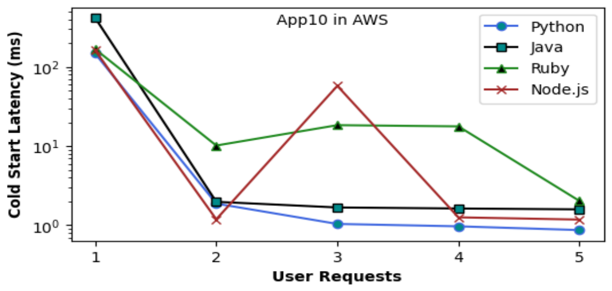


Fig. 12: Cold start latency of App10 in AWS

As observed from the results, except for App 6 as shown in Figure 8, it is clear that the cold start latency is very low from python language for all the chosen applications and is worst in case of Java programming language. Also, the cold start latency of all the ten applications are close to each other for a same programming language. Hence, python programming language is recommended for designing FaaS applications in AWS Lambda.

B. Azure Functions

Microsoft Azure Functions is a serverless computing solution that allows developers to construct and deploy event-driven functions without having to manage the underlying infrastructure. Azure Functions is intended to make it simple to write tiny bits of code that reacts to events such as HTTP requests, database updates, file uploads, and others. Node.js, .NET, Python, and Java are among the programming languages supported by Azure Functions. Azure Functions is a flexible and powerful serverless computing platform that is tightly connected with the Azure platform [53]. It may be used for everything from basic HTTP APIs to large event-driven applications and processes. To investigate the cold start latency in Azure, four programming languages are considered including Python, .NET, Node.js and Java. The details of the findings of ten apps written using different languages are presented in Table III.

The findings of the experiments carried on Azure platform for all the applications are shown graphically in Figures 13 to 22.

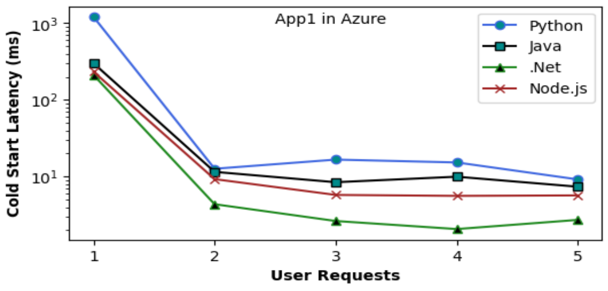


Fig. 13: Cold start latency of App1 in Azure

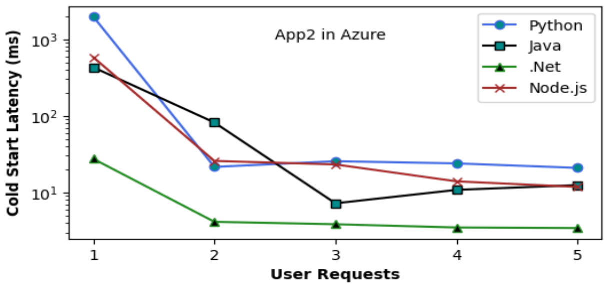


Fig. 14: Cold start latency of App2 in Azure

From the results presented in Table III and graphs, it is observed that .NET based FaaS application exhibit low latency compared to other programming languages. Out of

TABLE III: Cold start latency in Azure

App No	Request	Python	.Net	Node.js	Java
App1	1	1208.39	211.2878	233.0536	300.5885
	2	12.6612	4.3641	9.2684	11.5317
	3	16.6612	2.6364	5.7719	8.4519
	4	15.2976	2.0698	5.6082	9.9744
	5	9.192	2.7268	5.6882	7.3837
App2	1	1976.3024	27.3978	582.3396	430.1126
	2	21.4879	4.1179	25.727	81.7168
	3	25.513	3.8274	23.142	7.1881
	4	23.8869	3.4723	13.9164	10.7641
	5	20.8624	3.4152	11.7639	12.4233
App3	1	109.8126	96.4478	97.3636	221.1956
	2	47.8784	5.5316	20.6003	10.2047
	3	21.6472	4.2071	56.1972	10.0586
	4	13.59	5.5642	11.3768	9.2454
	5	19.6454	4.375	53.7805	7.7266
App4	1	1789.36196	132.3012	127.6848	359.0856
	2	59.6606	3.0137	9.8123	92.0028
	3	10.2039	2.231	7.6918	8.0407
	4	33.6337	1.9513	9.5791	12.777
	5	27.5866	2.2599	7.7278	8.3208
App5	1	20.2047	61.7047	151.9499	111.1516
	2	6.4607	2.051	32.3296	37.9257
	3	8.6043	4.8251	22.1659	9.0787
	4	3.7666	2.5145	9.222	6.5395
	5	5.0166	2.0799	12.1322	8.0905
App6	1	2318.025	38.8971	231.0558	225.5467
	2	54.5376	3.4311	9.6335	67.3493
	3	6.6494	2.0682	6.1433	14.014
	4	12.3834	2.8294	6.1948	11.2093
	5	8.436	2.7549	6.4808	7.1536
App7	1	2178.4853	25.9795	192.1277	254.0821
	2	57.8733	3.5518	9.8856	56.7708
	3	14.4021	2.1716	5.2187	12.0267
	4	16.5342	2.0125	5.2556	29.8186
	5	12.6525	2.8085	5.4202	11.1692
App8	1	2408.5559	122.825	78.0096	353.0733
	2	28.5449	1.9522	10.221	27.4601
	3	33.1856	2.2344	5.3812	8.5084
	4	40.7756	2.056	6.0257	9.3902
	5	29.6055	3.2536	6.7379	9.0331
App9	1	1951.5208	17.7898	25.4152	269.2903
	2	15.3996	5.9762	16.8295	47.3495
	3	30.0608	3.2495	13.7276	17.7801
	4	16.2486	3.3141	19.0869	8.6922
	5	47.3693	3.1401	10.8018	16.4285
App10	1	1510.9101	77.4328	24.6841	583.5832
	2	22.7408	5.3048	14.6535	163.2269
	3	35.3422	6.0761	11.2605	12.494
	4	13.843	4.1862	10.1972	19.9526
	5	28.4856	3.25	14.7314	10.2384

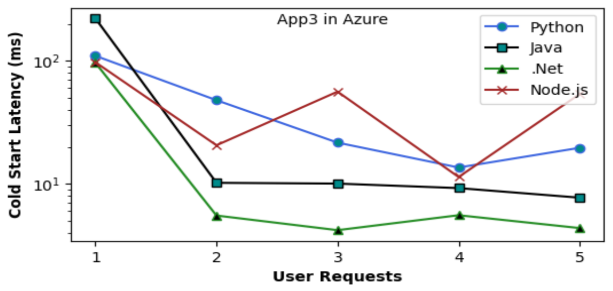


Fig. 15: Cold start latency of App3 in Azure

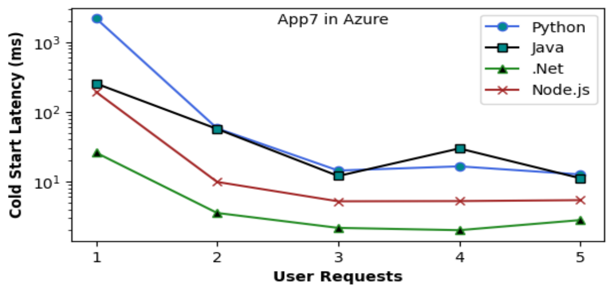


Fig. 19: Cold start latency of App7 in Azure

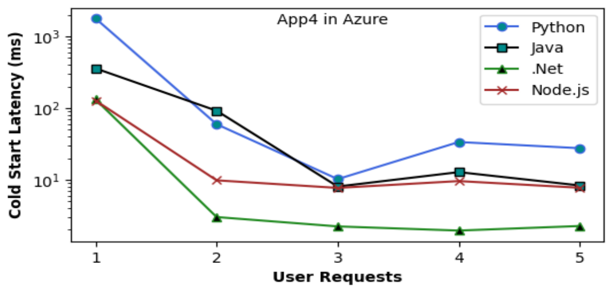


Fig. 16: Cold start latency of App4 in Azure

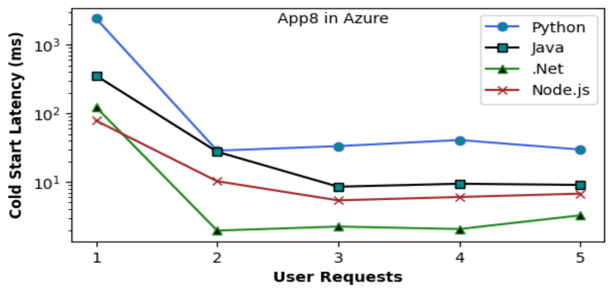


Fig. 20: Cold start latency of App8 in Azure

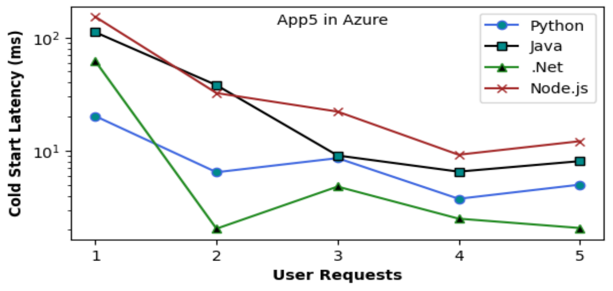


Fig. 17: Cold start latency of App5 in Azure

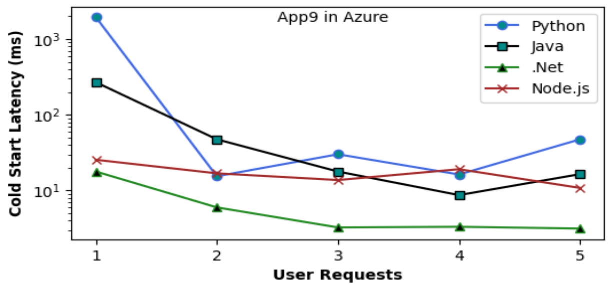


Fig. 21: Cold start latency of App9 in Azure

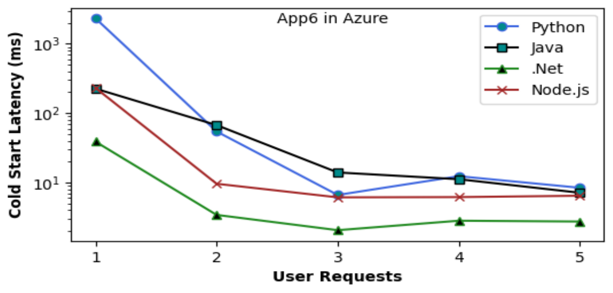


Fig. 18: Cold start latency of App6 in Azure

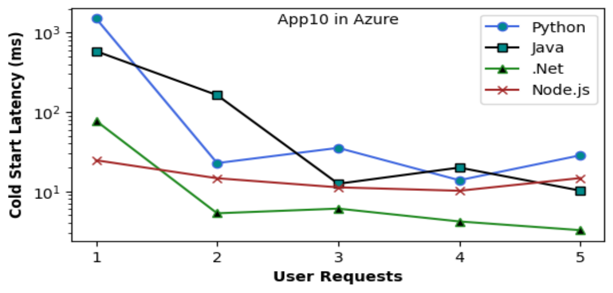


Fig. 22: Cold start latency of App10 in Azure

the ten chosen applications, .NET has low latency for seven applications and Node.js performs better for remaining three applications. One important point of discussion would be, as .NET is a framework offered by Microsoft and Azure is also a product of Microsoft, the FaaS applications designed with .NET are having low latency. Hence, it is recommended to utilize .NET as the programming language for serverless functions in Azure.

C. Google Cloud Functions

Google Cloud Functions is a Google Cloud Platform (GCP) serverless computing service. It enables developers to design and deploy event-based functions that grow dynamically in reaction to a variety of events. Google Cloud Functions is built to respond to a wide range of event triggers, both from GCP services and from outside sources. HTTP requests, Cloud Storage events, Cloud Pub/Sub messages, Firestore database updates, and other triggers are all supported [54]. Node.js, Python, Go, Java, and Ruby are among the programming languages supported by Google Cloud Functions. In this platform, all the programming languages supported by the GCP are considered for the investigations. The results of the investigations are presented in Table IV.

The results are presented graphically to show how the cold start latency reduces from the first invocation to the next 4 invocations done on the same function. The findings are presented in Figures 23 to 32.

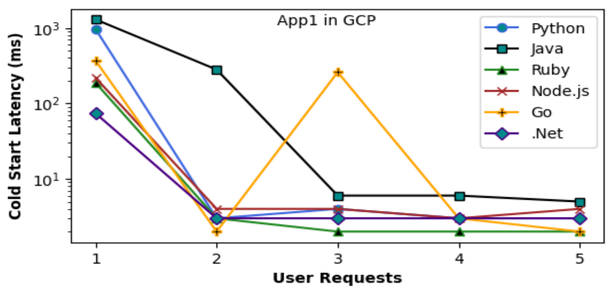


Fig. 23: Cold start latency of App1 in GCP

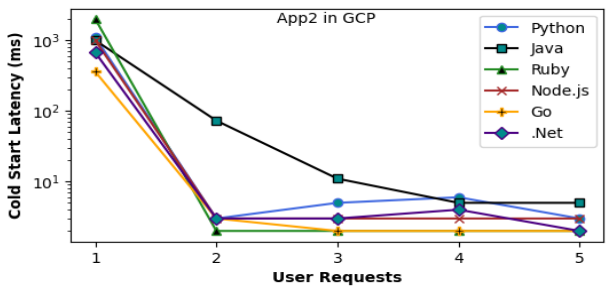


Fig. 24: Cold start latency of App2 in GCP

From the results, it is observed that the Apps designed with Go programming language exhibit lower latency compared to other languages supported in GCP. For seven applications, the cold start latency is lower for Go and python showcases lower latency for other FaaS applications. Again, it raises a point of

TABLE IV: Cold start latency in Google Cloud Platform

App No	Request	Python	Ruby	Node.js	Java	Go	.Net
App1	1	969	188	219	1300	370	74
	2	3	3	4	281	2	3
	3	4	2	4	6	260	3
	4	3	2	3	6	3	3
	5	3	2	4	5	2	3
App2	1	1100	2000	1000	1000	360	673
	2	3	2	3	73	3	3
	3	5	2	3	11	2	3
	4	6	2	3	5	2	4
	5	3	2	3	5	2	2
App3	1	10	110	172	160	82	735
	2	3	2	4	4	1	6
	3	3	4	5	4	2	9
	4	5	2	3	4	9	4
	5	3	2	4	5	2	4
App4	1	65	327	197	144	394	681
	2	6	4	3	7	2	9
	3	4	6	4	3	1	5
	4	4	5	5	6	1	5
	5	4	5	4	7	59	2
App5	1	84	48	10	18	3	576
	2	9	2	4	4	2	5
	3	3	5	3	6	2	4
	4	2	2	5	6	2	3
	5	3	2	3	5	2	3
App6	1	956	81	1300	218	21	593
	2	96	2	5	5	6	5
	3	3	3	6	3	3	5
	4	3	2	4	6	2	3
	5	3	4	4	4	1	4
App7	1	757	108	143	169	12	428
	2	4	2	6	8	3	13
	3	3	7	4	6	2	8
	4	3	3	7	5	1	3
	5	4	2	4	4	2	4
App8	1	49	91	213	51	9	1300
	2	3	2	4	9	1	5
	3	3	2	3	5	3	4
	4	3	2	5	6	1	4
	5	3	2	4	5	1	3
App9	1	883	2200	1100	1300	309	1200
	2	3	3	4	6	2	4
	3	2	2	13	5	1	4
	4	2	2	5	6	2	4
	5	4	2	5	8	2	4
App10	1	1100	2300	1300	1400	356	1300
	2	3	2	4	15	6	8
	3	3	3	4	5	1	3
	4	4	2	8	4	2	2
	5	4	2	10	5	2	4

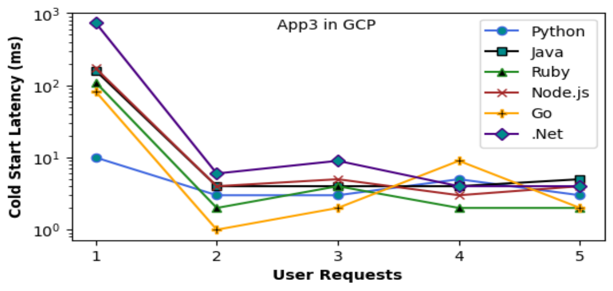


Fig. 25: Cold start latency of App3 in GCP

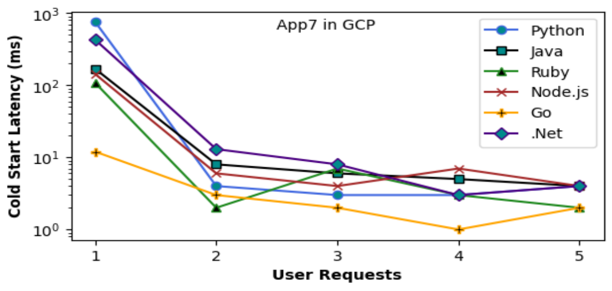


Fig. 29: Cold start latency of App7 in GCP

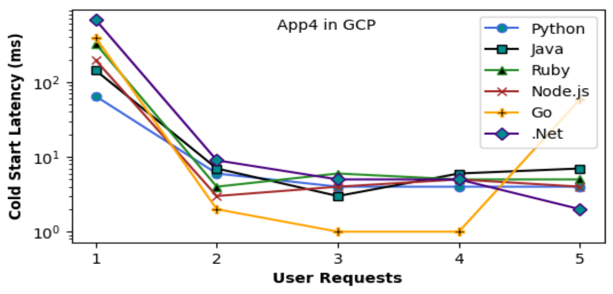


Fig. 26: Cold start latency of App4 in GCP

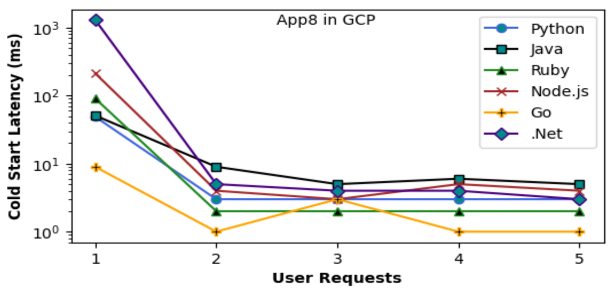


Fig. 30: Cold start latency of App8 in GCP

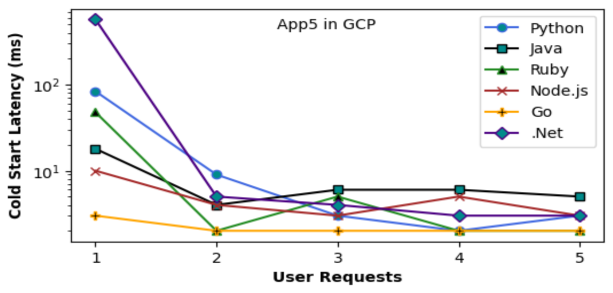


Fig. 27: Cold start latency of App5 in GCP

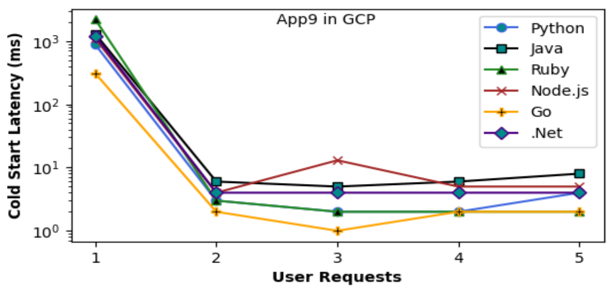


Fig. 31: Cold start latency of App9 in GCP

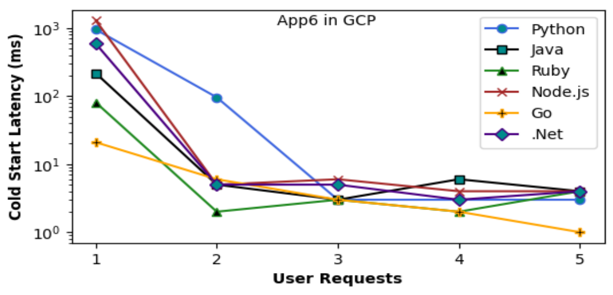


Fig. 28: Cold start latency of App6 in GCP

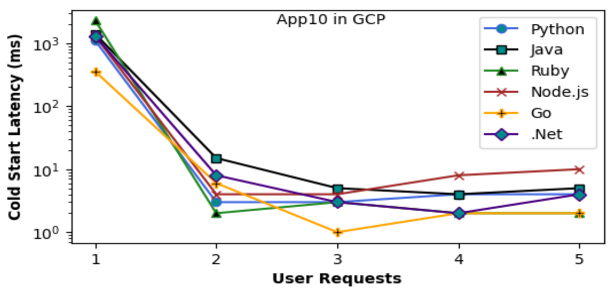


Fig. 32: Cold start latency of App10 in GCP

discussion and gives new direction of research to investigate whether the programming language developed for the parent serverless provider company shows lower cold start latency for the FaaS applications. Since, Go is a product of Google and GCP is the platform for designing serverless functions, it might be the reason for lower latency.

D. Comparison

To further demonstrate the empirical investigations of cold start and its impact with respect to the programming language, a comparison for each programming language and how it exhibits cold start latency in all three platforms is presented. Since there are common programming languages in all three platforms, the comparison is presented for common languages in all three platforms. All the languages supported by these platforms are very popular and each of them are widely used for design of many applications. Hence, this comparison will also give some underpinning ideas regarding the choice of the programming language when the serverless platform is fixed and developers have the choice only to select the language.

Python: The cold start performance of apps designed using python language are represented in Figure 33. It can be observed that Apps designed using python has lower cold start latency in AWS and it is higher in Azure.

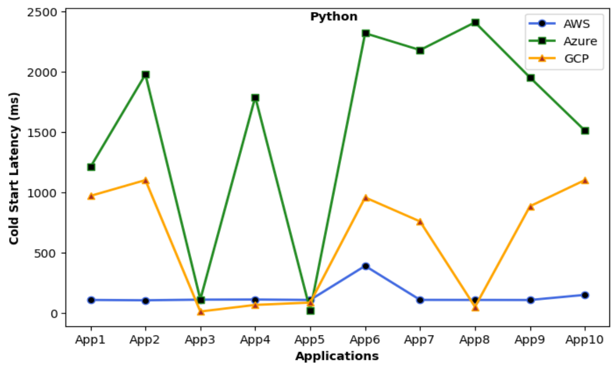


Fig. 33: Cold start latency of Python in all 3 platforms

Java: For the Apps designed using Java, the results of cold start latency is presented in Figure 34. The results for the chosen ten Apps shows different behavior in different platforms for Java language. For six applications, Java shows lower latency in GCP compared to other two platforms. However, considering the average latency for all ten Apps, Java performs better in Azure and it maintains stability.

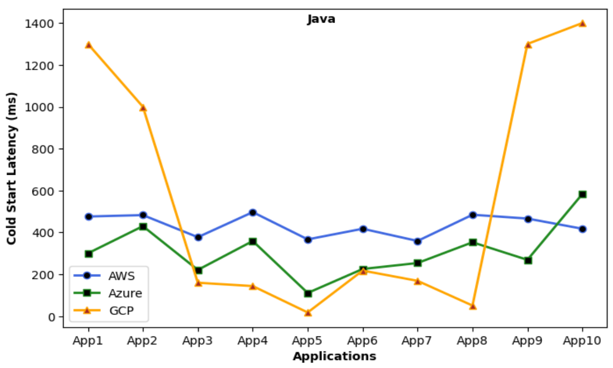


Fig. 34: Cold start latency of Java in all 3 platforms

Node.js: The cold start latency of apps designed using Node.js performs better in AWS platform for all the ten chosen functions. It could be observed from the Figure 35 that there is stable cold start for all the ten applications and its very low compared to other two platforms.

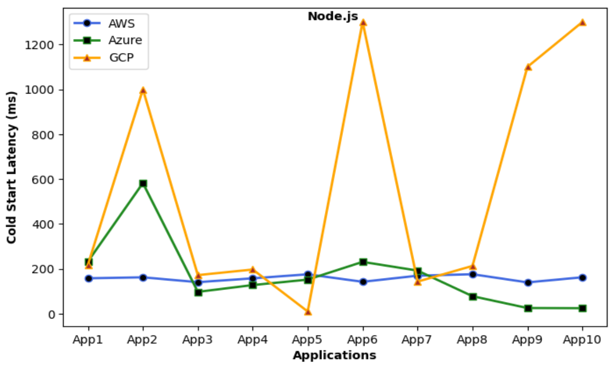


Fig. 35: Cold start latency of Node.js in all 3 platforms

Ruby: This language is supported only in two platforms, AWS and GCP. The results from Figure 36 clearly state that Ruby has very low latency when compared to functions in GCP.

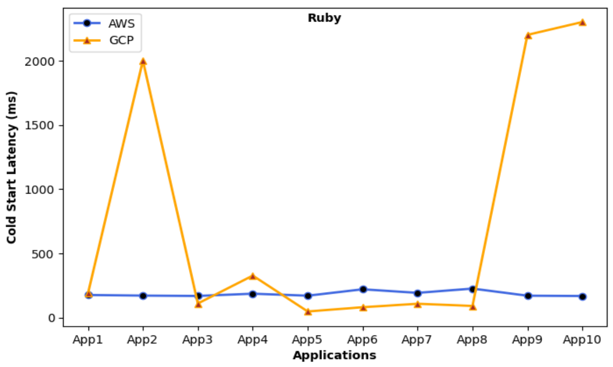


Fig. 36: Cold start latency of Ruby in AWS and GCP

.NET: Similarly, .NET is supported only by Azure and GCP and it is clear from the graph in Figure 37 that the cold start is very minimal for FaaS applications designed using .NET in Azure.

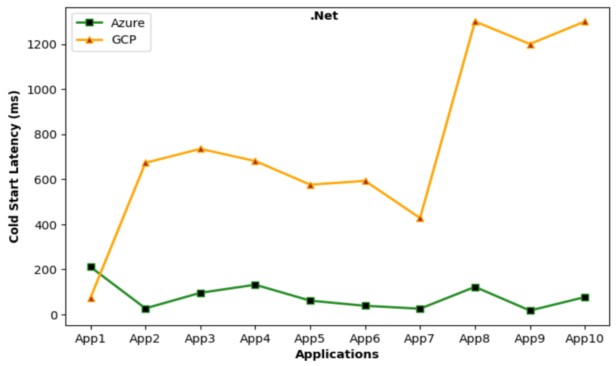


Fig. 37: Cold start latency of .NET in Azure and GCP

Though the cold start time varies for FaaS applications with the choice of programming language and also varies from one platform to another for a FaaS application designed using same programming language, it might not always be possible for the developer to select the platform of their choice and also the language of their interest. There are many factors influencing the choice of platform and corresponding language such as the cost of the infrastructure, rental for the CPU and memory consumption and also the number of million requests per sec. Hence, this work only gives the recommendation about the choice of platform as well as the programming language in each platform which has lower cold start latency. The next phase of the framework is to propose a mechanism for migrating the existing microservices to serverless platforms. The details of the proposed migration approach is presented in next sections.

VI. Migration Approach

Serverless computing has changed the way applications are designed and deployed. It has been a best alternative for deploying microservices applications [25]. The component which plays a major role in the automated migration approach is the generation of yml file. The serverless.yml file is a YAML configuration file that defines the functions, resources, plugins, and other configuration information for our serverless application. In this section, an automated approach for extracting the required serverless yml file from containerized microservices and deployment into serverless platforms using the yml file. The complete process of migrating the containerized microservices till the deployment in serverless platforms is presented in Figure 38. Since, AWS is very popular amomg all other providers considering the cost factors and cold start latency, in this initial study, AWS Lambda is considered as the serverless platform in the proposed migration work. The term serverless platform refers to AWS Labmda in the migration process. The structure and the configurations included in the yml file of the AWS Lambad are discussed here.

A. *Structure of yml file*

This file is used to configure a service and contains information about your functions, the events that cause them, and the AWS resources you should use [28]. Each service configuration is managed in the file. Following are the main features that should be considered in yml file:

- Define a serverless service
- Provide the cloud platform details to which the service will be deployed to like Google Cloud, AWS, Azure etc.
- Write one or more functions
- Specify the events that will trigger each function (e.g. HTTP requests)
- Define any plugin to use that extends behavior of serverless frameworks
- Define a set of AWS resources to create
- Allow events listed in the events section to automatically create the resources required for the event upon deployment
- Allow flexible configuration using variables

B. *Migration Algorithm*

This is the most critical phase in the whole process of migration also called the automation phase. A file named serverless.yml needs to be created which contains all the functions and infrastructure resources. It acts as a service configuration. This creation is automated by a Python script that extracts parameters for serverless.yml file. Every microservice project has one such file containing all the routes and requests along with other parameters of interest like path , handler function etc. It is assumed that this file is in the root folder of the project. Algorithm 1 contains the procedure that parses "routes.js" file first of all to get the methods of requests that exist, their paths that are used by API in the web browser and their respective handlers defined where the main logic of microservice rests. All these extracted values are populated in serverless.yml file under functions section. This automated algorithm which inputs the routing information of the containerized microservices generates a yml file which can be directly deployed in serverless platform.

C. *Serverless deployment*

Once the serverless.yml file is generated and handler functions are ready, the next step is to deploy the modified microservices into serverless platforms such as AWS. In the process of deployment, different lambda functions are created from individual microservices starting with the least critical part first. For each microservices, separate HTTP gateways are created and lambda functions are invoked using URLs. The general process of serverless deployment is as shown in below Figure 39.

D. *Experimental Analysis*

In this section, a containerized microservices is considered as a case study application as discussed in Section IV.C and the proposed approach is illustrated using this application. Also, a comparative analysis of performance parameters including response time and throughput is conducted for both containerized microservices and the ones migrated to serverless, the details of which are presented in below sections.

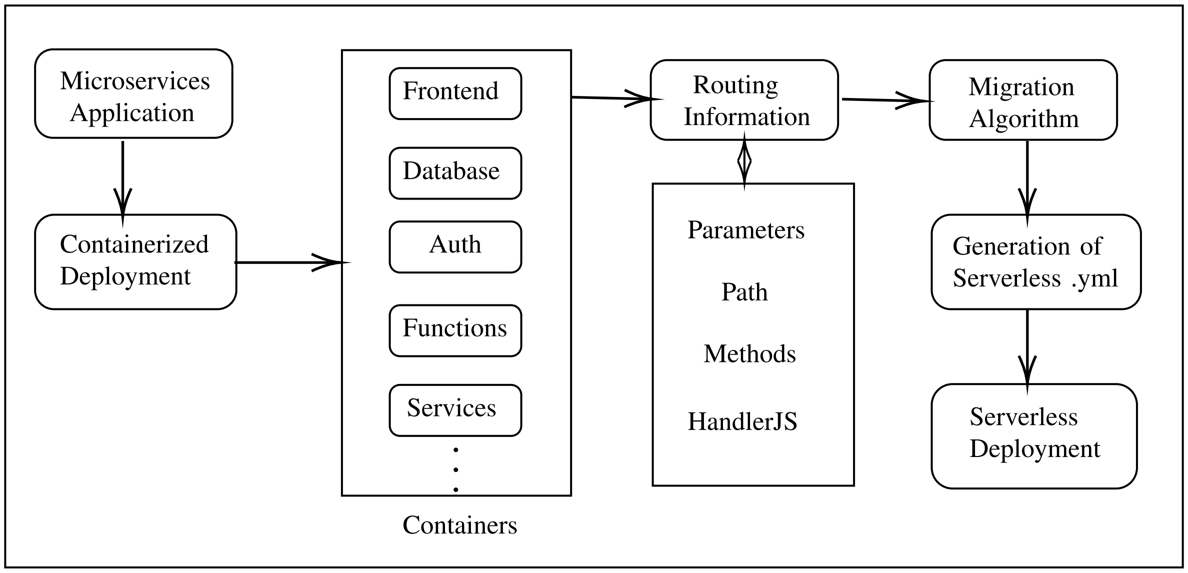


Fig. 38: Migration of microservices to serverless platforms

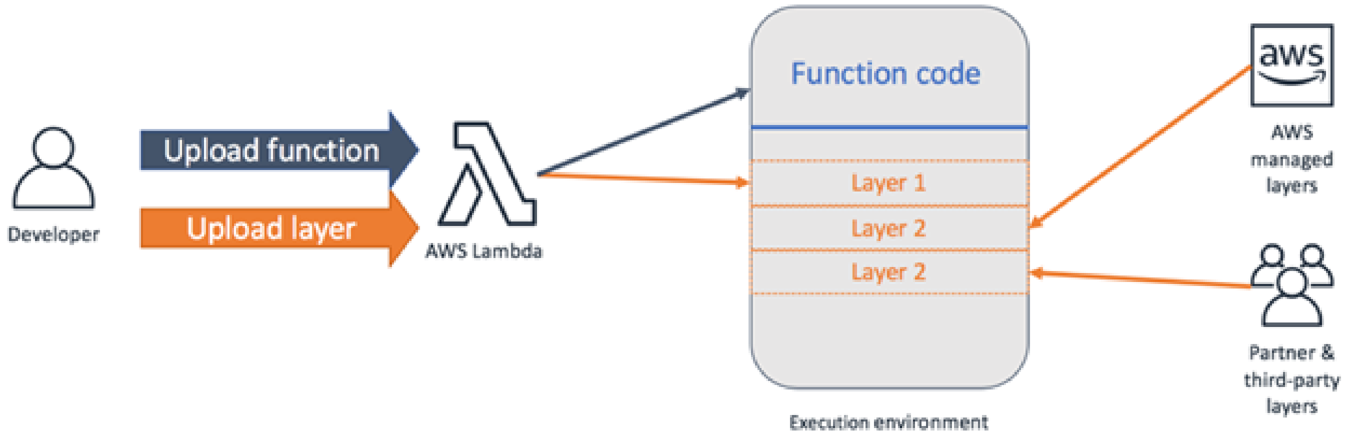
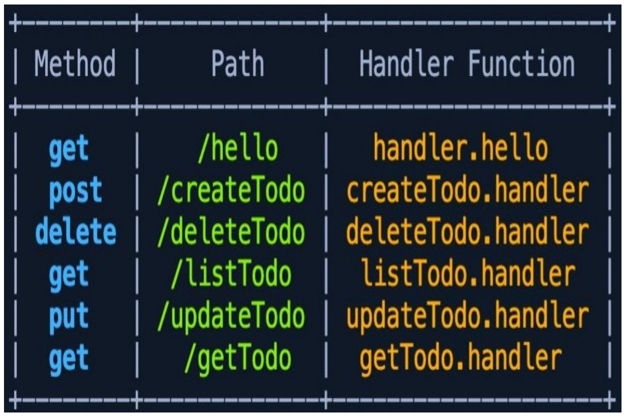


Fig. 39: Serverless Deployment Framework

By applying the algorithm on the case study application discussed in Section VI.B, the following output is generated with the method and the handler function. The handler functions are the methods in the code that activates the events upon request and once a particular handler returns a response, it activates the other events to satisfy the request. As mentioned in above sections, AWS Lambda is considered as the serverless platform in this paper. The below handler functions are generated suitable for AWS platform.


```
+--------+-------------+--------------------+
| Method |    Path     |  Handler Function  |
+--------+-------------+--------------------+
|  get   |   /hello    |   handler.hello    |
|  post  | /createTodo | createTodo.handler |
| delete | /deleteTodo | deleteTodo.handler |
|  get   |  /listTodo  |  listTodo.handler  |
|  put   | /updateTodo | updateTodo.handler |
|  get   |  /getTodo   |  getTodo.handler   |
+--------+-------------+--------------------+
```

1) Empirical analysis: Generally, to migrate applications from one architecture to another or from one programming language to another, it is very important to study the behavior of the applications before and after the migration. In this

Algorithm 1 serverless.yml generation

```
Input: Routing information file ”routes.js”
Output: Serverless.yml file
 1: Begin
 2: Initialize routes file to zero
 3: Initialize each route to zero
 4: Input the AppType
 5: if AppType == NodeJS then
 6:     Search for ‘package.json’ in directory
 7:     if Directory found then
 8:         Search for string “scripts” and “start”
 9:         Set path to value of filename written next to start
10:     end if
11:     Search for routes file in the path found by above statement
12:     if lines contain “get” or “post” or “put” or “delete” then
13:         Insert values of method, path and handler js function in each route.
14:     else
15:         Discard lines
16:     end if
17: else
18:     print “Not supported”
19: end if
20: Export eachroute to serverless.yml file return .yml file
21: End
```

regard, since the application is migrated from containers to serverless platform, an empirical analysis is carried out to assess the performance of the application post migration. Performance metrics such as response time and throughput are considered for analysis and since the applications are based on services, SoapUI is the suitable tool for analysis. SoapUI is a tool for testing Web Services; these can be the SOAP Web Services as well RESTful Web Services or HTTP based services. SoapUI is an Open Source and completely free tool with a commercial companion -ReadyAPI- that has extra functionality for companies with mission critical Web Services. It can be used to do functional testing, performance testing, interoperability testing, regression testing etc. Additionally load testing is performed for the chosen application to check latency, utilization and various other parameters.

In order to analyze the parameters, different test scenarios are defined. Initially, the microservices deployed in containers undergo load testing and then the serverless applications are tested for cold start latency and load testing. Finally, a comparison between both the styles of deployment is presented.

2) Load testing for containerized microservices: The To-do application deployed in containers are tested by sending HTTP requests to the microservices (API Calls) using SoapUI. These requests are sent in random order by user and this test is aimed to estimate the performance of the application with throughput and response time as metrics. The average response time captured for the containerized microservices is presented in Table V.

API call	**Average Response Time**
Hello_api	3 ms
create_todo	91 ms
update_todo	122 ms
delete_todo	129 ms
get_todo	129 ms
list_todos	105 ms

TABLE V: Average response time for microservices deployed in containers

3) Cold start for serverless functions: The most important factor of discussion is cold start latency in serverless platforms, especially in AWS [19], [20]. It is the time taken by the serverless functions to load all the required configurations and warming the instances. The functions of the chosen case study application is tested with different loads and the cold start latency is observed for each of the microservices. To analyze the performance of serverless functions with containerized microservices, cold start latency is observed and the results are presented in Table VI.

API call	**Maximum value**	**Average value**
Hello_api	9 ms	6 ms
create_todo	354 ms	250 ms
update_todo	349 ms	200 ms
delete_todo	310 ms	233 ms
get_todo	355 ms	261 ms
list_todos	296 ms	145 ms

TABLE VI: Cold start latency for each API call

4) Load testing on serverless functions: Using SoapUI tool, API calls are submitted for the case study application deployed in serverless platform. In order to analyze the results in detail, different loads with 500 and 1000 users are submitted through SoapUI and the performance metrics such as response time and throughput are captured. The results are presented in Table VII.

Performance metric	**500 users**	**1000 users**
Average response time (ms)	4620.6	8126
Throughput (transactions/sec)	4.06	0.68

TABLE VII: Load testing results with 500 and 1000 users

5) Comparison & Discussion: The results of performance testing conducted on both containerized microservices and corresponding serverless functions are compared in terms of average response time and throughput. The comparison is shown in Table VIII. It very clear from the results that the serverless deployment model has better response time and more number of transactions are executed successfully. This states that the serverless platforms are best suited for deploying microservices based applications instead of containers.

Performance metric	**Serverless**	**Containerized**
Average response time (ms)	200	389
Throughput (transactions/sec)	156.85	17.06

TABLE VIII: Comparison of metrics for serverless and containerized deployment

With the current buzz about the serverless computing in both industry and academia, and with the comparative results, serverless computing will attain high attention and most of the internet applications will be designed and deployed using serverless. Though container based approach for microservices is running fine for applications, looking into the factors affecting the customer satisfaction, serverless will dominate the other existing cloud solutions.

VII. Conclusion

A framework for migration of microservices based applications to serverless platform with efficient cold start latency is proposed. Specifically, an empirical investigation to study the behavior of cold start with respect to the programming language in three major serverless platforms including AWS, Azure and GCP is presented. From the results, it is very clear that programming language has an impact on the cold start time of the serverless functions. However, only the choice of the language does not directly impacts the cold start. Later, an approach for migrating the existing microservices based applications to serverless platforms is provided. Also, a comparison between the microservices deployed in containers and microservices which are designed as serverless functions is also presented. It is observed that the microservices which are designed and deployed as serverless functions has better response time and throughput values. This study can be further extended by investigating the various causes for cold start and proposing techniques for mitigating the cold start problem in serverless platforms avoiding the vendor lock-in problem.

References

[1] Raj V, Sadam R. Evaluation of SOA-based web services and microservices architecture using complexity metrics. *SN Computer Science*. 2021 Sep;2:1-0.

[2] Raj, V., 2021. Framework for Migration of SOA based Applications to Microservices Architecture. Journal of Computer Science & Technology, 21.

[3] Hassan, H.B., Barakat, S.A. and Sarhan, Q.I., 2021. Survey on serverless computing. Journal of Cloud Computing, 10(1), pp.1-29.

[4] Raj V, Ravichandra S. Microservices: A perfect SOA based solution for Enterprise Applications compared to Web Services. In2018 3rd IEEE International Conference on recent trends in electronics, information & communication technology (RTEICT) 2018 May 18 (pp. 1531-1536). IEEE.

[5] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. and Suter, P., 2017. Serverless computing: Current trends and open problems. Research advances in cloud computing, pp.1-20.

[6] Cassel, G.A.S., Rodrigues, V.F., da Rosa Righi, R., Bez, M.R., Nepomuceno, A.C. and da Costa, C.A., 2022. Serverless computing for Internet of Things: A systematic literature review. Future Generation Computer Systems, 128, pp.299-316.

[7] Shafiei, H., Khonsari, A. and Mousavi, P., 2022. Serverless computing: a survey of opportunities, challenges, and applications. ACM Computing Surveys, 54(11s), pp.1-32.

[8] Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. and Guo, M., 2022. The serverless computing survey: A technical primer for design architecture. ACM Computing Surveys (CSUR), 54(10s), pp.1-34.

[9] Li, Y., Lin, Y., Wang, Y., Ye, K. and Xu, C., 2022. Serverless computing: state-of-the-art, challenges and opportunities. IEEE Transactions on Services Computing, 16(2), pp.1522-1539.

[10] Yussupov, V., Breitenbücher, U., Leymann, F. and Müller, C., 2019, December. Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends. In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (pp. 273-283).

[11] Adzic, G. and Chatley, R., 2017, August. Serverless computing: economic and architectural impact. In Proceedings of the 2017 11th joint meeting on foundations of software engineering (pp. 884-889).

[12] Goli, A., Hajihassani, O., Khazaei, H., Ardakanian, O., Rashidi, M. and Dauphinee, T., 2020, April. Migrating from monolithic to serverless: A fintech case study. In Companion of the ACM/SPEC International Conference on Performance Engineering (pp. 20-25).

[13] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S., 2018, April. Serverless computing: An investigation of factors influencing microservice performance. In 2018 IEEE international conference on cloud engineering (IC2E) (pp. 159-169). IEEE.

[14] Kaushik, N., Kumar, H. and Raj, V., 2023. Empirical Evaluation of Microservices Architecture. Communication and Intelligent Systems: Proceedings of ICCIS 2022, Volume 2, 689, p.241.

[15] Wen, J., Chen, Z., Liu, Y., Lou, Y., Ma, Y., Huang, G., Jin, X. and Liu, X., 2021, August. An empirical study on challenges of application development in serverless computing. In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 416-428).

[16] Wen, J., Chen, Z., Jin, X. and Liu, X., 2023. Rise of the Planet of Serverless Computing: A Systematic Review. ACM Transactions on Software Engineering and Methodology.

[17] Kaplunovich, A., 2019, May. ToLambda–Automatic Path to Serverless Architectures. In 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR) (pp. 1-8). IEEE.

[18] Chadha, M., Pacyna, V., Jindal, A., Gu, J. and Gerndt, M., 2022, November. Migrating from microservices to serverless: an IoT platform case study. In Proceedings of the Eighth International Workshop on Serverless Computing (pp. 19-24).

[19] Lin, P.M. and Glikson, A., 2019. Mitigating cold starts in serverless platforms: A pool-based approach. arXiv preprint arXiv:1903.12221.

[20] Vahidinia, P., Farahani, B. and Aliee, F.S., 2020, August. Cold start in serverless computing: Current trends and mitigation strategies. In 2020 International Conference on Omni-layer Intelligent Systems (COINS) (pp. 1-7). IEEE.

[21] Jin, Z., Zhu, Y., Zhu, J., Yu, D., Li, C., Chen, R., Akkus, I.E. and Xu, Y., 2021, August. Lessons learned from migrating complex stateful applications onto serverless platforms. In Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (pp. 89-96).

[22] Capuano, R. and Muccini, H., 2022, March. A Systematic Literature Review on Migration to Microservices: a Quality Attributes perspective. In 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C) (pp. 120-123). IEEE.

[23] Jambunathan, B. and Yoganathan, K., 2018, March. Architecture decision on using microservices or serverless functions with containers. In 2018 International Conference on Current Trends Towards Converging Technologies (ICCTCT) (pp. 1-7). IEEE.

[24] Lloyd, W., Vu, M., Zhang, B., David, O. and Leavesley, G., 2018, December. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion) (pp. 195-200). IEEE.

[25] Heikkinen, J., 2023. Serverless and microservice architecture in modern software development.

[26] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S., 2018, April. Serverless computing: An investigation of factors influencing microservice performance. In 2018 IEEE international conference on cloud engineering (IC2E) (pp. 159-169). IEEE.

[27] Nupponen, J. and Taibi, D., 2020, March. Serverless: What it is, what to do and what not to do. In 2020 IEEE International Conference on Software Architecture Companion (ICSA-C) (pp. 49-50). IEEE.

[28] Casale, G., Artač, M., Van Den Heuvel, W.J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A. and Srirama, S.N., 2020. Radon: rational decomposition and orchestration for serverless computing. SICS Software-Intensive Cyber-Physical Systems, 35, pp.77-87.

[29] Raj V, Ravichandra S. A service graph based extraction of microservices from monolith services of service-oriented architecture. Software: Practice and Experience. 2022 Jul;52(7):1661-78.

[30] Menéndez, J.M., Gayo, J.E.L., Canal, E.R. and Fernández, A.E., 2023. A comparison between traditional and Serverless technologies in a microservices setting. arXiv preprint arXiv:2305.13933.

[31] Osman, M.H., Saadbouh, C., Sharif, K.Y. and Admodisastro, N., 2022. From Monolith to Microservices: A Semi-Automated Approach for Legacy to Modern Architecture Transition using Static Analysis. International Journal of Advanced Computer Science and Applications, 13(10).

[32] Bajaj, D., Bharti, U., Goel, A. and Gupta, S.C., 2020, May. Partial migration for re-architecting a cloud native monolithic application into microservices and faas. In International Conference on Information, Communication and Computing Technology (pp. 111-124). Singapore: Springer Singapore.

[33] Cordingly, R., Yu, H., Hoang, V., Sadeghi, Z., Foster, D., Perez, D., Hatchett, R. and Lloyd, W., 2020, December. The serverless application analytics framework: Enabling design trade-off evaluation for serverless software. In Proceedings of the 2020 Sixth International Workshop on Serverless Computing (pp. 67-72).

[34] Martins, H., Araujo, F. and da Cunha, P.R., 2020. Benchmarking serverless computing platforms. Journal of Grid Computing, 18, pp.691-709.

[35] Xie, D., Hu, Y. and Qin, L., 2021, September. An evaluation of serverless computing on x86 and arm platforms: Performance and design implications. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD) (pp. 313-321). IEEE.

[36] Scheuner, J., Deng, R., Steghöfer, J.P. and Leitner, P., 2022, December. CrossFit: Fine-grained Benchmarking of Serverless Application Performance across Cloud Providers. In 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC) (pp. 51-60). IEEE.

[37] Thönes, J., 2015. Microservices. IEEE software, 32(1), pp.116-116.

[38] Raj, V. and Sadam, R., 2021. Performance and complexity comparison of service oriented architecture and microservices architecture. International Journal of Communication Networks and Distributed Systems, 27(1), pp.100-117.

[39] Raj, V. and Sadam, R., 2021. Patterns for Migration of SOA Based Applications to Microservices Architecture. Journal of Web Engineering, pp.1229-1246.

[40] Raj, V. and Srinivasa Reddy, K., 2022, February. Best Practices and Strategy for the Migration of Service-Oriented Architecture-Based Applications to Microservices Architecture. In Proceedings of Second International Conference on Advances in Computer Engineering and Communication Systems: ICACECS 2021 (pp. 439-449). Singapore: Springer Nature Singapore.

[41] Raj, V. and Bhukya, H., 2023. Assessing the Impact of Migration from SOA to Microservices Architecture. SN Computer Science, 4(5), p.577.

[42] Henry, A. and Ridene, Y., 2020. Migrating to microservices. Microservices: Science and Engineering, pp.45-72.

[43] Auer, F., Lenarduzzi, V., Felderer, M. and Taibi, D., 2021. From monolithic systems to Microservices: An assessment framework. Information and Software Technology, 137, p.106600.

[44] Feng, L., Kudva, P., Da Silva, D. and Hu, J., 2018, July. Exploring serverless computing for neural network training. In 2018 IEEE 11th international conference on cloud computing (CLOUD) (pp. 334-341). IEEE.

[45] Shankar, V., Krauth, K., Vodrahalli, K., Pu, Q., Recht, B., Stoica, I., Ragan-Kelley, J., Jonas, E. and Venkataraman, S., 2020, October. Serverless linear algebra. In Proceedings of the 11th ACM Symposium on Cloud Computing (pp. 281-295).

[46] Ao, L., Izhikevich, L., Voelker, G.M. and Porter, G., 2018, October. Sprocket: A serverless video processing framework. In Proceedings of the ACM Symposium on Cloud Computing (pp. 263-274).

[47] Castro, P., Ishakian, V., Muthusamy, V. and Slominski, A., 2019. The rise of serverless computing. Communications of the ACM, 62(12), pp.44-54.

[48] Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R. and Cheng, Y., 2021. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21) (pp. 443-457).

[49] Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H. and Jin, X., 2023. FaaSLight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology.

[50] Suo, K., Son, J., Cheng, D., Chen, W. and Baidya, S., 2021, September. Tackling cold start of serverless applications by efficient and adaptive container runtime reusing. In 2021 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 433-443). IEEE.

[51] Kaushik, P., Rao, A.M., Singh, D.P., Vashisht, S. and Gupta, S., 2021, November. Cloud computing and comparison based on service and performance between Amazon AWS, Microsoft Azure, and Google Cloud. In 2021 International Conference on Technological Advancements and Innovations (ICTAI) (pp. 268-273). IEEE.

[52] Mishra, P., 2023. Advanced AWS Services. In Cloud Computing with AWS: Everything You Need to Know to be an AWS Cloud Practitioner (pp. 247-277). Berkeley, CA: Apress.

[53] Sawhney, R. and Chanumolu, K., 2023. Best Practices for Working with Azure Functions. In Beginning Azure Functions: Building Scalable and Serverless Apps (pp. 165-171). Berkeley, CA: Apress.

[54] Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H. and Jin, X., 2023. FaaSLight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology.

[55] Cicconetti, C., Conti, M. and Passarella, A., 2020. A decentralized framework for serverless edge computing in the internet of things. IEEE Transactions on Network and Service Management, 18(2), pp.2166-2180.

[56] Paraskevoulakou, E. and Kyriazis, D., 2023. ML-FaaS: Towards exploiting the serverless paradigm to facilitate Machine Learning Functions as a Service. IEEE Transactions on Network and Service Management.

[57] Ristov, S., Kimovski, D. and Fahringer, T., 2022. Faascinating resilience for serverless function choreographies in federated clouds. IEEE Transactions on Network and Service Management, 19(3), pp.2440-2452.

[58] Perez, R., Benedetti, P., Pergolesi, M., Garcia-Reinoso, J., Zabala, A., Serrano, P., Femminella, M., Reali, G., Steenhaut, K. and Banchs, A., 2022. Monitoring platform evolution toward serverless computing for 5G and beyond systems. IEEE Transactions on Network and Service Management, 19(2), pp.1489-1504.

[59] Raza, A., Akhtar, N., Isahagian, V., Matta, I. and Huang, L., 2023. Configuration and Placement of Serverless Applications using Statistical Learning. IEEE Transactions on Network and Service Management.

[60] Xu, M., Song, C., Ilager, S., Gill, S.S., Zhao, J., Ye, K. and Xu, C., 2022. CoScal: Multifaceted scaling of microservices with reinforcement learning. IEEE Transactions on Network and Service Management, 19(4), pp.3995-4009.