future internet

MDPI

Article

Analyzing the Features, Usability, and Performance of Deploying a Containerized Mobile Web Application on Serverless Cloud Platforms

Jeong Yang * and Anoop Abraham

Department of Computational, Engineering, and Mathematical Sciences, Texas A&M University—San Antonio, One University Way, San Antonio, TX 78224, USA; anoopabraham25@gmail.com
* Correspondence: jeong.yang@tamusa.edu

Abstract: Serverless computing services are offered by major cloud service providers such as Google Cloud Platform, Amazon Web Services, and Microsoft Azure. The primary purpose of the services is to offer efficiency and scalability in modern software development and IT operations while reducing overall costs and operational complexity. However, prospective customers often question which serverless service will best meet their organizational and business needs. This study analyzed the features, usability, and performance of three serverless cloud computing platforms: Google Cloud's Cloud Run, Amazon Web Service's App Runner, and Microsoft Azure's Container Apps. The analysis was conducted with a containerized mobile application designed to track real-time bus locations for San Antonio public buses on specific routes and provide estimated arrival times for selected bus stops. The study evaluated various system-related features, including service configuration, pricing, and memory and CPU capacity, along with performance metrics such as container latency, distance matrix API response time, and CPU utilization for each service. The results of the analysis revealed that Google's Cloud Run demonstrated better performance and usability than AWS's App Runner and Microsoft Azure's Container Apps. Cloud Run exhibited lower latency and faster response time for distance matrix queries. These findings provide valuable insights for selecting an appropriate serverless cloud service for similar containerized web applications.

Keywords: serverless; cloud computing; performance; usability; Cloud Run; App Runner; Container App; cold start

Citation: Yang, J.; Abraham, A. Analyzing the Features, Usability, and Performance of Deploying a Containerized Mobile Web Application on Serverless Cloud Platforms. *Future Internet* **2024**, *16*, 475. https://doi.org/10.3390/fi16120475

Academic Editor: Gianluigi Ferrari

Received: 1 November 2024
Revised: 15 December 2024
Accepted: 16 December 2024
Published: 19 December 2024

1. Introduction

Cloud computing is a paradigm that enables the seamless utilization of services over the Internet, with the unique capability of dynamic scalability. In the past, the cloud has often been considered to represent a specific portion of the Internet that includes certain infrastructure. However, currently, the term "cloud" has evolved to serve as a metaphor for the wide range of services offered over the Internet [1]. The concept of cloud computing dates back to 1950–1960 [2,3]. There were dramatic developments in this space after IBM introduced its new operating system named VM, which allowed its mainframe systems to have multiple virtual systems, or 'virtual machines (VM)', on a single physical node [3]. The National Institute of Standards and Technology (NIST) defines cloud computing as follows. "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [4].

Serverless cloud computing is a model of cloud computing wherein developers can develop and deploy applications without configuring underlying infrastructures. It is a relatively new concept. According to [5], serverless cloud computing streamlines nearly all system administration tasks, effectively simplifying the cloud usage experience for

programmers. When cloud computing became popular for the first time, developers were drawn to solutions like Amazon EC2 because they offered full control over application instances [5]. However, this also meant that developers were responsible for managing and scaling their applications, which could be time-consuming and complex. Additionally, developers often did not want to make significant changes to their existing code to make it cloud friendly. As cloud computing matured, new technologies like FaaS (Function as a Service) emerged. FaaS provides a serverless computing platform that allows developers to run code without having to worry about infrastructure or scaling. This makes it ideal for running small, frequently invoked tasks, such as processing payments or sending emails.

Currently, there are many cloud service providers available, with Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft (MS) Azure being the main players. Despite their current popularity, there are not many similar research results available to provide a comparison of their usability and performance. In this study, we have conducted a comparative study of these services using a SmartSAT Django web application with a heavy reliance on Google Maps and distance matrix APIs. This application was selected for the study as part of the grant-funded project to develop a customizable mobile app for San Antonio (SA) Transit that provides critical services to transit users in SA. It is similar to other apps like Google Maps, Moovit, and Transit, but SmartSAT is designed specifically to improve the transit experience for lower-income people who rely on San Antonio's public VIA Transit. The analysis incorporated and focused on three serverless platforms: Google's Cloud Run, AWS' App Runner, and Azure's Container Apps. With this, the significant contributions of the research include the following:

- An overview of recent academic research related to serverless cloud computing from 2015 to 2023;
- A comparative analysis of the serverless service features offered by Cloud Run, App Runner, and Container Apps;
- A usability analysis of the serverless services offered by Cloud Run, App Runner, and Container Apps based on their ease of use and suitability for beginners in serverless computing;
- A performance analysis providing valuable insights for selecting an appropriate serverless platform to deploy similar applications such as a containerized Django web application with the heavy use of Google Maps APIs;
- Suggestions on the best serverless service to deploy containerized web applications with the heavy use of Google Maps APIs.

The rest of the paper is organized as follows: Section 2 reviews academic studies in the cloud computing and serverless computing domain conducted from 2015 onward, including three serverless services focused on the study; Section 3 discusses the Django SmartSAT application used for evaluating the services; Section 4 discusses the research methodology emphasized on three performance metrics for the performance study; Section 5 presents the results of the analysis on system features, performance, and usability; Section 6 presents overall discussion with results; and Section 7 concludes the work with a discussion of future work.

2. Literature Review

Cloud computing refers to servers accessed over the Internet and the software and databases on those servers; its services provide resources based on virtualization [6] and containerization [7]. Modern VM technologies [8–10] allow a single server that is divided into multiple virtual containers [11]. Serverless computing is a next-generation service delivery approach in which service providers offer just the resources required for the length of the requested services' execution [12–14]. The flexibility to perform this fine-grained resource allocation helps users and service providers. Serverless computing allows developers to focus only on developing the most exemplary front-end application code and business logic possible.

Developers just need to build application code and deliver it to containers controlled by a cloud service provider [14]. The remainder will be handled by the cloud provider, who sets up the infrastructure required to run the code and scales it up and down on demand. In addition, the cloud provider is in charge of all routine infrastructure administration and maintenance, such as operating system updates and patches, security management, capacity planning, system monitoring, and so on. Also, with serverless computing, developers never have to pay for idle capacity. When the code executes, the cloud provider spins up and supplies the necessary computer resources on demand and then spins them back down—a process known as 'scaling to zero'—when execution ends. Billing begins when execution begins and finishes when execution concludes; price is often dependent on execution time and resources required.

Zaid Al-Ali et al. proposed moving serverless computing beyond the current limited scope of an event-driven framework to support a broader range of programming paradigms [15]. Mingyu Wu et al. comprehensively assessed serverless computing in three areas: applications suited for serverless computing, performance challenges, and security concerns, in their paper [16]. Due to the availability of an overwhelming number of cloud providers' services, it is often difficult for developers to evaluate each service based on multiple parameters to choose one for their needs. Theo Lynn et al. presented a list of parameters on which they evaluated different services from the major primary cloud providers in their paper [12]. There are various surveys on serverless computing, each with different focuses and purposes. In one article [17], Yongkang Li et al. thoroughly reviewed the state-of-the-art, challenges, and opportunities in serverless cloud computing. Since cloud computing is an area of computer science where innovation happens daily, we observed that some of the shortcomings pointed out in articles that are two to three years old are not at all a problem in modern cloud computing.

2.1. Background on Serverless Cloud Computing

The idea of serverless computing has been around for many years, but it started to take off in the early 2010s. This was due in part to the increasing popularity of cloud computing, as well as the need for serverless platforms like AWS Lambda and Google Cloud Functions. In the early days, serverless computing was primarily used for small, event-driven tasks. However, as the technology has matured, it has become more widely adopted for a wider range of applications. Today, serverless computing is used by businesses of all sizes to build applications from simple web applications to complex microservice architectures.

There is a good amount of research happening in the serverless domain, exploring its advantages and disadvantages. After conducting a thorough analysis of the literature published in this field from 2015 to 2023, a brief overview is presented below. Zaid Al-Ali et al. [15] proposed a new abstraction for serverless computing named ServerlessOS. It abstracts away the details of resource management so developers can focus on their code. This allows processes to be seamlessly scaled across the data center, making serverless computing more serverless. R. Arokia Paul Rajan [18] provided a comprehensive study of the serverless computing architecture and the working principle of the serverless computing reference model adapted by AWS Lambda. Eivy and Weinman [19] analyzed the economics of serverless cloud computing, showing that the cost of a publicly hosted serverless service can grow quickly, even for low levels of traffic. They also noted that the free quota offered by most cloud service providers is insufficient for a decent public web service. Van Eyk et al. [20] discussed the early days of serverless computing and the obstacles and opportunities that existed at the time. Some of the obstacles, such as the lack of maturity of the technology and the limited availability of serverless platforms, have been addressed in recent years.

With large organizations moving to the cloud, protecting the data in the cloud is equally important. Anaya Garde et al. [21] proposed a microservices-based approach to protect cloud-native assets using a serverless framework. This approach first discovers cloud assets and takes a snapshot-based backup of them. The backed-up assets can then be

used to recover the asset in the event of a disaster or data loss. In [22], Mileski and Gusev used Google Cloud's serverless services to experiment with using serverless computing for the real-time monitoring of thousands of patients with streaming electrocardiograms. This is an example of an embarrassingly parallel task, which means that it can be broken down into small, independent tasks that can be executed in parallel. They found that the serverless solution was able to achieve a speedup $S(W)$ of almost 40 compared to a sequential execution on a virtual machine, and a speedup of 23 compared to a parallel execution using virtual machines. The speedup $S(W)$ was calculated with $Ts(W)$ divided by $Tp(W)$ where they represent response times for VM sequential and serverless parallel versions, respectively, for a specific load W [22].

Initially, serverless computing was largely associated with Function as a Service (FaaS) offerings from various cloud providers. However, there has been limited research in this area analyzing and comparing the different services in this domain [23–25]. But, in 2023, there were more serverless offerings from most of the major cloud service providers. This major change happened with the introduction of containers in serverless cloud computing [26]. In [27], Rodriguez Cortes et al. discussed the open issues of the serverless architecture.

2.2. Serverless Cloud Computing Platforms

Google Cloud Run, AWS App Runner, and MS Azure Container Apps are serverless cloud computing platforms, which offer such services on hassle-free containerized applications running without infrastructure management. These platforms prioritize scalability, reliability, and cost-effectiveness. Specifically, Cloud Run allows developers to run stateless containers via HTTP requests, App Runner enables running containerized web apps and API services, and Container Apps deploys containerized applications from code or containers without complex infrastructure orchestration. Table 1 summarizes the different features offered by these serverless services. Although they offer various features, they also share many similarities. The following describes the specific details of each service.

Table 1. Different features offered by three serverless services.

Feature	Google Cloud Run	AWS App Runner	Microsoft Azure Container Apps
Automatic Build and Deployment	Yes	Yes	Yes
Load Balancing	Yes	Yes	Yes
Scalability	Automatic	Automatic	Automatic
Scaling to Zero	Yes	No	Yes
General Purpose	Yes	No	Yes
Security	High	High	High
Cost Effectiveness	Competitive	Competitive	Competitive
Ability to Split Traffic between Different Versions of the Application	Yes	Yes	Yes
Cloud Monitoring Options	Yes	Yes	Yes

2.2.1. Google Cloud Run

Google Cloud documentation defines Cloud Run as a serverless computing platform that allows developers to run stateless containers that are invocable via HTTP requests. Cloud Run is a fully managed service that abstracts away all infrastructure management, so developers can focus on what matters most—building great applications [28]. Cloud Run was first announced in 2018 and made generally available in 2019. Cloud Run is built on top of the Kubernetes container orchestration platform [29]. Kubernetes is a popular open-source project that is used by many organizations to manage containerized applications. Cloud Run offers a number of benefits [30], including:

Traffic Management: Cloud Run's traffic management allows you to customize how incoming traffic is routed to your application revisions [28]. You can route traffic to the latest revision, a previous revision, or even split traffic between multiple revisions simultaneously. You can also fine-tune the percentage of incoming traffic that goes to each revision. This

gives you a lot of flexibility in how you deploy and manage your applications, ensuring that they are always available and performing at their best.

Cost-effectiveness: Cloud Run charges customers for the CPU and memory allocated to a container instance, rounded up to the nearest 100 milliseconds. If a service is not in use, customers are not charged. This can lead to significant cost savings, especially for applications with variable or unpredictable usage patterns. There are two pricing models: request-based and instance-based. With request-based, customers are charged a per-request fee when a container instance is processing requests. With instance-based, customers are charged for the entire lifetime of a container instance, but there is no per-request fee.

Scalability: Cloud Run automatically adds and removes container instances to handle all incoming requests. This is known as auto-scaling. Cloud Run can scale to zero [28], which means that, if there are no incoming requests, even the last remaining container instance will be removed. This is the default behavior, and the default number of minimum instances is 0. This can help to improve application performance and reliability, and it can also help to save money by avoiding the over-provisioning of resources. However, setting the minimum number of instances to 0 may result in cold starts. A cold start occurs when a container is started for the first time since it was last shut down. Cold starts can add latency to requests because the container needs to load its application code and dependencies before it can start processing requests.

DevOps Automation: Cloud Run can help to automate many of the tasks involved in DevOps, such as provisioning and scaling infrastructure, managing application deployments, and monitoring application health. This can free up DevOps teams to focus on other tasks, such as improving application performance and security.

Security and Privacy: Cloud Run employs security and privacy measures with its Borg-based architecture to protect customer data and privacy [31]. Its key features include data encryption for all traffic in the network using Transport Layer Security (TLS) and secure sandboxed execution environments. The data is also encrypted at rest with Advanced Encryption Standard (AES) 256-bit keys. Cloud Run services are accessible from the Internet by default. To restrict access, customers can configure a security policy that specifies which IP addresses or hostnames are allowed to access the service. If no security policy is configured, all incoming traffic is allowed access, which is not recommended for production environments. With Identity and Access Management (IAM), customers can configure security policies that restrict access to their services to only authorized users or systems.

Load Balancing: Google Cloud Load Balancer is a fully distributed software-defined managed service that is integrated into Cloud Run [31]. Through secure and robust load distribution, load balancing in Cloud Run handles traffic distribution across multiple infrastructure components and manages incoming traffic either via custom domains or the run.app URL. This includes an HTTP proxy for zone load balancing and a scheduler to allocate requests to appropriate servers hosting sandboxed applications.

Agility: Cloud Run can help to improve the agility of application development and deployment. By eliminating the need to provision and manage servers, developers can focus on building and testing applications. This can help to shorten the time to market for new applications.

2.2.2. AWS App Runner

AWS App Runner is a fully managed service that helps to build, deploy, and run containerized web applications and API services without prior infrastructure or container experience [32]. It provides a high-level abstraction over the underlying infrastructure, allowing one to focus on the application code. App Runner supports a variety of programming languages and frameworks, including Node.js, Python, Java, Go, and .NET. It also supports a variety of container images, including Docker and Amazon ECS. App Runner also provides a number of features that make it easy to manage applications, including the following:

Automatic Scaling: App Runner's autoscale feature can help users to improve the performance, reliability, and cost-effectiveness of the application. App Runner automatically scales one's application up or down based on demand, so users only pay for the resources that they use [32,33]. App Runner can also automatically scale one's application to handle spikes in traffic, so one's users always have a good experience. Additionally, App Runner can automatically recover from failures by scaling one's application back up. Users can provide an auto-scaling configuration to customize the scaling behavior. If one does not provide one, App Runner provides a default configuration with recommended values. Users can share a single auto-scaling configuration across multiple App Runner services to ensure that they have the same auto-scaling behavior. Unlike Google Cloud Run, AWS App Runner does not allow the scale down to zero option.

Health Checks: App Runner automatically performs health checks on one's application. If an application fails a health check, App Runner will automatically restart it [33].

Logging and Monitoring: App Runner provides detailed logs and metrics for an application. This helps users to troubleshoot problems and identify performance bottlenecks [33].

Automatic Deployments: With App Runner, one can easily build and deploy one's application in a matter of minutes. Simply connect App Runner to the code repository or container image registry to get started. App Runner will then monitor the repository for any updates to the source code or container image. Once an update is detected, App Runner will automatically build and deploy the new version of the application [33]. This feature is an efficient way to keep one's application up to date, reducing the time it takes to deploy new features and bug fixes while improving the overall reliability of the application.

Security and Privacy: App Runner employs AWS's shared responsibility model, which involves the security and compliance responsibilities/requirements between AWS and customers [34]. Customers can determine their responsibilities by assessing the requirements using frameworks like the NIST Cybersecurity Framework (CSF) and International Organization of Standardization (ISO), reviewing service documentation for security configurations, and evaluating security and compliance services. To protect data and privacy, account credentials are enforced by IAM, which only grants access to users with the permissions required for specific roles. For additional protection, multi-factor authentication (MFA), TLS for communication, and AWS encryption with built-in security controls are utilized.

Load Balancing: App Runner offers the important capability of automatically balancing traffic across multiple containers [33]. This feature is highly significant in managing unexpected surges in traffic while ensuring optimal performance and availability. Toward this, AWS's Application Load Balancer operates at the request level, routing traffic to targets such as containers and IP addresses. Given this, it proves to be a viable option for applications that require a robust traffic capacity and dependable availability.

Networking: App Runner provides users with the flexibility to tailor the interaction between their service, applications, and resources. Users are afforded the option to restrict access to their service solely within an Amazon VPC or enable the service to communicate with other AWS services within a VPC. This degree of control enables users to satisfy their security and networking compliance requirements [33].

2.2.3. Microsoft Azure Container Apps

Azure Container Apps is a fully managed environment that enables one to run microservices and containerized applications on a serverless platform [35]. It is a good choice for applications that need to be deployed quickly and easily and that can scale dynamically based on demand. Container Apps can be used to deploy API endpoints, host background processing applications, handle event-driven processing, and run microservices. Applications built on Azure Container Apps can dynamically scale based on HTTP traffic, event-driven processing, or CPU or memory load.

Serverless Hosting: Container Apps provides a serverless architecture that eliminates the need for managing the underlying infrastructure. This translates to a worry-free

experience for developers as they do not have to be concerned with server provisioning, capacity management, or application scaling. Instead, they can focus solely on developing their applications with ease.

Automatic Scaling: Container Apps provides a solution to the problem of having to constantly monitor and adjust application capacity. With this technology, the applications can be scaled automatically based on various factors, such as CPU usage, memory usage, and network traffic [35]. This ensures that the applications are always accessible to users without any interruptions. The dynamic scaling feature of Azure Container Apps is a great feature for businesses that need to handle varying levels of demand for their applications. By removing the burden of manual capacity provisioning, companies can focus on other aspects of their operations that are critical to success.

Security and Privacy: Container Apps allows customers to configure sensitive values as secrets that can be used for revisions in container apps [35]. A secret can be used from Azure Key Vault in a container app, enabling managed identity for the app and granting it access to the Ky Vault by creating an access policy with permission [36]. Container Apps also provide robust security features like role-based access control and network isolation to keep your apps secure from unauthorized access. They are an excellent choice for maintaining app security.

Load Balancing: Azure Container Apps environment has a Load Balancer that can distribute network traffic to appropriate servers and applications [37]. The load balancer is accessible through an IP address, which serves as an entry point for all incoming internal and external traffic, regardless of which replica handles requests. Standard Load Balancer is part of a private virtual network and built on the Zero-Trust network security model [37].

Simple to use: Container Apps offers a user-friendly experience, featuring a straightforward interface that simplifies the process of deploying and managing one's applications. This makes it an excellent choice for developers who are just starting with containerization. Through the Azure Container Apps portal, one can easily create and manage one's applications. Additionally, users have the option to manage their applications using Azure CLI or Azure PowerShell [35].

Cost-effective: Deploying and scaling applications is made cost-effective with Azure Container Apps. Users only pay for the resources they utilize, with no upfront costs [35,38]. The pricing for Azure Container Apps is determined by the number of containers run and the amount of memory consumed.

Flexible: Azure Container Apps offers a versatile platform for deploying a diverse range of applications, making it an excellent choice for various use cases. One can deploy web applications, microservices, and serverless functions with ease using Container Apps.

3. Deployment of an Application on Serverless Platforms

The SmartSAT application was developed as part of the grant project [30], whose key objectives are to provide convenient and equitable access to San Antonio residents for its public transit service. This makes it compatible with the selected services in this study, as it was already deployed and containerized on a targeted serverless platform on Google Cloud.

3.1. SmartSAT Django Web Application

SmartSAT is a customizable mobile app for San Antonio Transit that provides critical services to transit users. It is similar to other apps like Google Maps, Moovit, and Transit, but SmartSAT is designed specifically to improve the transit experience for lower-income people who rely on San Antonio's public VIA Transit [39]. Its main objective is to provide real-time bus location information to reduce riders' wait times and thus the length of their daily commutes. The application tracks bus locations in real time, ultimately improving the prediction accuracy of bus arrival time. SmartSAT offers a variety of features that are not available on other apps, such as the following:

- Real-time bus arrival information;

- Seat capacity information;
- Instant alert messages on schedule changes;
- Secure data collection and feedback from riders on their commute experience.

These features can help lower-income people who rely on public transportation to plan their trips more efficiently and avoid delays. They can also help riders to provide feedback to VIA Transit so that the agency can improve its services.

Google Maps JavaScript API was used to showcase the real-time location of the buses on the selected route. Additionally, the SmartSAT app has the capacity to calculate and display the estimated arrival time at a selected bus stop on the route. The Google Distance Matrix API was used for this purpose, taking into consideration various factors such as traffic patterns, road accidents, and other elements that may impact the trip time. The app currently supports about ten VIA bus routes, but only three were used for the experiment of the performance analysis. Directions API was used to showcase routes based on travel time and modes as well as Places and Geocoding API for addressing current locations.

To use the app, users first select their desired bus route from the home screen. The app will then load all the bus stops on that route, including the order in which they appear. By clicking on any of the bus stop markers, users can view the estimated arrival time for that stop. The app uses a Postgres database in Google Cloud SQL to maintain data related to users and bus routes.

3.2. Deployment of SmartSAT Application on Serverless Platforms

SmartSAT was initially deployed on Cloud Run, Google's serverless platform, which offers services on hassle-free containerized applications running without the user's infrastructure management. Using containers in computing has a rich and extensive history [40,41]. Unlike hypervisor virtualization, where an intermediation layer enables running one or more independent machines virtually on physical hardware, containers operate in user space atop an operating system's kernel [42,43]. Thus, container virtualization is often referred to as operating system-level virtualization [42]. This technology enables the creation of numerous isolated user-space instances on a single host.

Docker is an app platform that streamlines the process of building, deploying, and running apps by bundling all the necessary elements into containers. The advantages of this approach include easy mobility, efficient resource utilization, and a quick, streamlined development cycle [42,44]. Because of these benefits, we opted to package our application in a docker container when the application was deployed on a serverless platform. We followed the instructions outlined in [45] to establish the initial configuration of the docker file and build it for Google's Cloud Run platform. We then made minor tweaks to the docker file to address various application requirements. To guarantee deployment on other serverless cloud services on AWS App Runner and Azure Container Apps, we modified the settings files to include the appropriate port numbers and parameter values.

4. Research Methodology

The primary goal of the study was to analyze and evaluate the system features, usability, and performance when deploying a containerized SmartSAT mobile web application [30,39,46] on three different serverless cloud platforms: Google's Cloud Run, AWS's App Runner, and Microsoft Azure's Container Apps. Based on the advantages of using serverless cloud services discussed in Section 2: Literature Review, these three serverless platforms were chosen to conduct the comparative analysis in deploying the SmartSAT application. The evaluation analyzed the results of the system features, usability, and performance when running the application on each of the serverless platforms. The study aimed to guide the implementation of the SmartSAT application by addressing three specific research questions:

- RQ1: What are the key differences in system features between serverless computing services provided by AWS, GCP, and Azure, as determined by comparing their service configurations, pricing model, and memory and CPU specifications?

- RQ2: Which cloud provider among GCP, AWS, and Azure offers the most beginner-friendly documentation and learning resources for serverless computing services, as accessed for a usability study?
- RQ3: How do serverless computing services on AWS, GCP, and Azure differ in the performance of running a containerized web application, as determined by measuring container request latency, API response time, and CPU utilization?

4.1. Measurements on System Features and Usability

A comparative analysis of system features across three serverless services involves examining configurations, pricing models, and key system attributes such as memory and CPU specifications. We identified significant differences among serverless computing services offered by GCP, AWS, and Azure by comparing how each service's configuration and pricing models make them suitable for deploying similar applications and use cases. We also studied measuring the usability of the services, specifically from the perspective of a beginner cloud user, focusing on the effectiveness of deploying the application and the ease or difficulty of learning and performing the deployment tasks on each service.

4.2. Performance Measurements

The SmartSAT application was slightly modified to enable its deployment on each serverless platform. Nothing related to the application functionality was modified. Only the way that is used to load the database user credentials was modified. For the original Cloud Run service, the values were stored in the secret manager service and loaded from there. However, during the time of running the tests, these values were hard-coded. This was just to reduce the extra setup time required for the additional service configurations. Thus, all three platforms were tested with the database user credentials hard coded.

This application was chosen for its ease of deployment, as it was already developed and containerized as part of the project [30], making it compatible with the selected services. To create a required load on the application, an Android emulator was used to simulate the GPS (Global Positioning System) movement of a bus along a selected bus route. The following steps were manually performed on each serverless platform (Cloud Run, App Runner, and Container Apps):

- Navigate to the deployed service link from the Android emulator and log in to the website as a bus driver;
- Start two different bus routes. (This will emulate the same scenario when a real bus moves in that route);
- Access the application as a normal user (Bus Rider) from different devices;
- Click on a different bus stop icon so that the website shows the estimated arrival time for that stop. (a) Repeat this task for both the active routes from different tabs opened in the web browser. This will mimic multiple people accessing the application; (b) Once both buses reach their destination stops, stop the bus driving from the driver screen;
- Navigate to the cloud monitoring option in Google Cloud for the corresponding service and note the results.

Traditionally, latency is measured as the time between a user's request and the response to that request. In a cloud environment, latency can be affected by factors such as user's network congestion on the client side and physical processing delays on the server side. Response time is similar to latency, but broadened to include not only latency but also server processing and queuing delays. A high latency typically relates to a high response time.

For this study, we evaluated the performance of running the SmartSAT application on serverless platforms. As the application was deployed in a container, key metrics such as 'Container Request Latency', 'Container CPU Utilization', and 'API Response Time' were measured during the performance analysis of the application. Unlike the traditional latency, the 'Container Request Latency' in this measurement does not include user-side network

congestion. The ranges of these metrics with a visual representation of their concepts are presented in Figure 1. A detailed description of the metrics is provided below:

- *Container Request Latency (L)*: When a client (rider) sends a request to a server and waits for the corresponding response, the time it takes for this process is known as request latency. This is a crucial performance metric in the realm of computer networks and distributed systems. Factors such as distance between the client and server, network congestion, and hardware limitations can impact the latency. In this study, we conducted performance testing for three services from the same device and network. It is important to note that the latency values captured primarily reflect the container request latency and not the client-side network latency. Additionally, the cloud provider's network load can influence the service's response time to user requests. By maintaining consistent user-side network and hardware configurations, the latency measurements captured by the cloud provider's application monitoring services provide a clear picture of the application latency for the specific service. Mathematically, the container request latency (L) can be defined as

$$L = T_{response} - T_{start} \tag{1}$$

where T_{start} is the time that the client sends a request for an arrival time and $T_{response}$ is the time that the response to the request is received. Again, this only captures the container request latency and not the client-side network latency. For multiple requests of n , the average container request latency L_{avg} can be calculated as

$$L_{avg} = \frac{1}{n}\sum_{i=1}^{n}(T_{response,i} - T_{start,i}) \tag{2}$$

- *API Response Time (R)*: API response time is defined as the duration between the time that the client sends a request and the moment that the system responds with the intended output on the server-side processing. This focuses on obtaining the distance matrix API's response to the user's request for the arrival time required to travel between two locations based on the distance. A graphical representation of the API response time is shown in Figure 1. The significance of response time in modern computing cannot be overstated, as it directly impacts the user experience. A slow response time can lead to frustration, decreased productivity, and negative perceptions of the system or application. Therefore, it is imperative to measure or optimize response time to enhance user satisfaction. Various techniques such as caching, load balancing, and the optimization of code can be employed to reduce response time. The API response time (R) can be mathematically defined as

$$R = T_{response-from-API} - T_{call-to-API} \tag{3}$$

where $T_{call\text{-}to\text{-}APT}$ is the time when the client request calls the distance matrix API for an arrival time and $T_{response\text{-}from\text{-}API}$ is the time that the response to the request returns to the call. For multiple requests of n, the average API response time R_{avg} can be calculated as

$$R_{avg} = \frac{1}{n}\sum_{i=1}^{n}(T_{response-from-API,i} - T_{call-to-API,i}) \tag{4}$$

- *CPU Utilization*: When evaluating the performance of the containerized application, it is essential to consider its CPU usage. During the test load, the CPU usage for the container can provide valuable insights into its efficiency and effectiveness. By analyzing the CPU usage patterns, it is possible to identify any bottlenecks or performance issues that may be impacting the container's performance. It is important to note that CPU usage can vary depending on the workload and the hardware configuration of the system. Therefore, it is essential to establish a baseline for CPU usage and compare

it against the actual usage during the test load. This comparison can help to identify any abnormal behavior that may be indicative of performance issues.

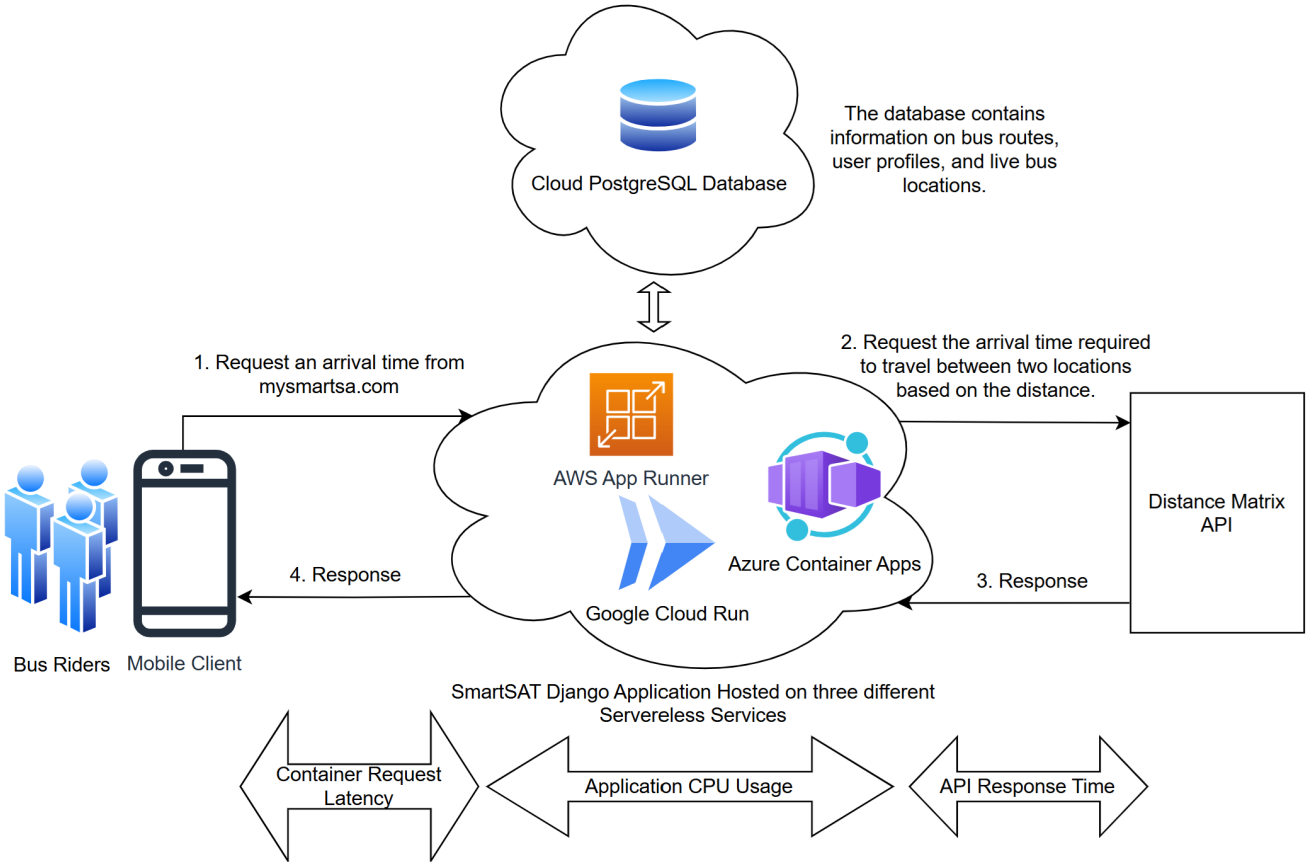

Figure 1. Visual representation of request latency, CPU usage, and API response time concepts.

5. Evaluation Results

This section presents the results of the analysis in terms of system features, usability, and performance using the three serverless services: Google's Cloud Run, AWS's App Runner, and Microsoft Azure's Container Apps. Tables 2–4 show the results of the analysis. The numbers in Table 4 are mostly sourced from configuring the serverless services.

5.1. Results of System Feature Analysis

Comparing system features on each of the services is associated with service configuration, pricing, and relevant system features such as memory and CPU. For the posed question RQ1, we successfully identified the key differences in system features between the serverless computing services. Table 2 compares the service configurations, pricing, and memory and CPU of the three services. These services offer different configurations and pricing models, making them suitable for different types of applications and use cases. While the application deployment used default resources on each of the three serverless platforms, the pricing model may vary if deployed in a different tier or with GPU resources.

Cloud Run is Google's serverless computing platform that allows for the easy deployment of containerized applications. It is priced based on the number of requests and compute time used. The first 120 min of build time per day is free. This experiment used an instance with 0.5 GB of memory per container and a single CPU. The application was able to handle a high volume of requests without any performance degradation.

AWS App Runner, on the other hand, does not offer free quotas. It costs around $0.064 per vCPU-hour and $0.007 per GB-hour. It supports two pricing models: a provisioned plan and an active container instance plan. The maximum number of instances for the service

is limited to 25, but the maximum concurrency is 200, compared to 100 for Cloud Run. The maximum concurrency for an instance of the AWS App Runner can be found during the configuration stage of the service. We tried setting a large value on the configuration page and found the system suggesting 200 as the maximum value allowed for the selected instance. Thus, the value 200 was added. The maximum memory size available for a container is 12 GB, while Cloud Run has a capacity of 34 GB. App Runner also has a maximum CPU capacity of 4, while Cloud Run has 8.

Table 2. Comparison results for system features.

Service Configuration	Google Cloud Run	AWS App Runner	Microsoft Azure Container Apps
No. of CPUs	1		1
Memory	0.5 GB/container		2.0 GB
Min Number of Instances	0	1	0
Max Number of Instances	100 by default, but depends on CPU and memory configurations	25	300
Max Concurrency per Instance	1000	200	Supports custom value
Pricing			
Compute Time Unit	100 ms	1 s	1 s
Pricing Models	Request-based, instance-based	Provisioned, and active container instances	Plans available based on resource consumption and a dedicated plan option
Free Tier	CPU: first 180,000 vCPU-s per month, Memory: first 360,000 GiB-s per month, Requests: 2 million requests/month	No Free tier, CPU: $0.064/vCPU-h Memory: $0.007/GB-h	CPU: first 180,000 vCPU-s, Memory: 360,000 GiB-s, Requests: 2 million requests each month are free
Memory and CPU			
Min Memory	1 GB	0.5 GB	0.5
Max Memory	34 GB	12 GB	4
Min vCPUs	<1	0.25	0.25
Max vCPUs	8	4	2

Microsoft Container Apps is a relatively new service launched in May 2022 [47]. It offers two pricing options: one based on resource utilization and one based on a fixed pricing plan. The free-tier quotas are the same for both Cloud Run and Container Apps. Container Apps support scaling to zero if the scaling setting is not based on CPU utilization. The maximum number of instances supported is 300, which is higher than the other two services. In this test, a configuration of 1 CPU core and 2 GB of memory was used for the container. There was a good amount of documentation available about getting started with the hosting service, but due to the recent release of the service, there were not many external resources available about the service.

5.2. Results of Usability Analysis

ISO 9241-11 [48,49] is a standard with many uses for the usability testing of software systems. The framework comprises three components: system effectiveness, which assesses the users' ability to accomplish the assigned tasks; system efficiency, which relates to the time it takes for users to complete tasks; and system satisfaction, which records the users' opinions and feedback [49–51]. This standard defines the word usability as "the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" [51], while Oxford Languages refers to "the degree to which something is able or fit to be used".

This study focuses on analyzing the usability of serverless services, specifically from the perspective of a beginner cloud user. The authors analyzed the effectiveness of deploying the application on each respective service. The ease or difficulty of learning and

performing the deployment task on each service, as well as the time it took for the authors to complete it, were considered measures of efficiency. The satisfaction experienced throughout the task, from beginning to end, is the determining factor for recommending services. Two significant factors that affect the time that it takes for a user to complete tasks on a new system are documentation and the learning curve. The "scale-to-zero" feature eliminates the need for additional configurations on services to adjust their capacity based on the system's request load. These three factors can be attributed to the "effectiveness" and "efficiency", as discussed in ISO 9241-11.

Cloud Run, App Runner, and Container Apps are serverless container platforms that allow one to deploy and run containerized applications without having to worry about managing servers. Analyzing the easy-to-use usability of three serverless services is associated with its documentation, learning curve for beginners, and scale-to-zero capability. This is to address RQ2: Which cloud provider among GCP, AWS, and Azure offers the most beginner-friendly documentation and learning resources for serverless computing services? The results of the usability analysis are summarized in Table 3.

Table 3. Comparison results on ease-of-use usability.

Ease of Use [1]	Google Cloud Run	AWS App Runner	Microsoft Azure Container Apps
Documentation	Excellent 4.5/5	Good 3/5	Good 3.5/5
The learning curve for a beginner	Gradual learning curve. The availability of a lot of getting-started templates makes learning faster.	Pretty steep. Too much information to grasp.	Gradual learning curve. External resources were less as it is relatively new.
Scale to zero	Yes [2]	No	Yes [2]

[1] The ratings of the documentation and learning curve presented were solely based on the authors' experience during the study with the three cloud services. They may not necessarily reflect the objective views or opinions of others. [2] Applications that scale on CPU or memory load cannot scale to zero.

For this analysis, we have evaluated the quality of documentation available for various cloud services by primarily referring to their official documentation pages. Additionally, we have taken into consideration the availability of open learning resources on the Internet. Through the analysis, we have discovered that Google Cloud run service has the most detailed and user-friendly official documentation, along with several third-party resources readily available online. AWS App Runner service, on the other hand, has documentation that may be challenging for beginners to follow due to the abundance of information on the cloud console dashboard. However, there are still resources available online for application deployment on the service. Azure has a more straightforward cloud console and clear documentation compared to AWS, but, due to its recent launch, there are fewer online resources available as compared to the other two services. It should be noted that these ratings were solely based on the authors' experience during the study with the three cloud services. They may not necessarily reflect the objective views or opinions of others. As they are subjective, the evaluation results may be not very significant and representative.

Scale to zero is another factor that is considered for the usability analysis of the services. This feature allows a service to shut down the container if it does not receive any requests for a specified amount of time. This is particularly beneficial in terms of cost savings when the end user is not utilizing the application. While this capability is only available on GCP and Azure, it proves to be useful when scaling based on requests instead of CPU and memory usage. Container Apps support scaling to zero when the scaling setting is not based on CPU utilization.

Table 4. Comparison results on request latency, response time, and CPU utilization.

Performance Metrics	Google Cloud Run	AWS App Runner	Microsoft Azure Container Apps
Request Latency (Container)	50 ms	91 ms	-
Response Time (Distance Matrix API)	63.7 ms	71.38 ms	66.8 ms
CPU Utilization	23%	47%	0.000773% [1]

Note: The corresponding graphs for each parameter can be found in Figures 2–7. [1] Azure App monitoring graph shows the values in millicores. Thus, this value is converted to a percentage using Formula (5).

5.3. Results of Performance Analysis

The performance of the serverless service was evaluated using a containerized Django web application described in Section 4. The performance metrics 'Container Request Latency', 'Response Time (Distance Matrix API)', and 'CPU Utilization' of the application were measured when the application was hosted on each of the three serverless platforms. This analysis is to address RQ3: How do serverless computing services on AWS, GCP, and Azure differ in terms of the performance of running a containerized application? Table 4 shows the results of the evaluation.

To obtain the values of the metrics, two different bus routes were emulated in the application with the help of an Android emulator. Then, the application was accessed by various clients (bus riders) emulating more load on the service. The 'Container Request Latency' is the time taken for a process when a client sends a request to a server and waits for the corresponding response. The latency values were captured primarily reflecting the container request latency and not the client-side network latency. API response time is the duration between the time a user sends a request and the moment the system responds with the intended output. This is to obtain the distance matrix API's response to the user's request. The CPU utilization represents the CPU usage for the execution of the application during the test load. These three metrics were measured by using the cloud monitoring services provided by the corresponding cloud provider while the application was running.

5.3.1. Results Analysis

In terms of the 'Container Request Latency', Google's Cloud Run demonstrated a lower latency compared to AWS's App Runner. Cloud Run has latency, with an average request latency of 50 milliseconds (Figure 2). AWS App Runner has a higher latency, with an average request latency of 91 milliseconds (Figure 3). Azure Container Apps monitoring services do not have the ability to capture the container latency for the applications deployed. Due to this, the values for the Azure Container apps are not available. The total load on the cloud provider for a specific time can impact the performance of that provider. The 'total load on a cloud provider' refers to the capacity of the provider's entire infrastructure, not just the specific service we have built. If a cloud provider's physical resources in the region where our application is located are heavily utilized, this could negatively affect our application's overall performance. This is because increased network traffic can sometimes lead to slight performance degradation. The performance test for the Cloud Run service and the remaining two services were performed over two days. We cannot attribute any specific behavior from the Google Cloud as a key factor for better results on the latency value.

Regarding the response time to distance matrix API, the average response time was better when the application was hosted on Cloud Run than being hosted on App Runner and Container Apps (Figure 4). The response time for Cloud Run is 63.7 milliseconds on average, while App Runner and Container Apps have an average response time of 71.38 and 66.8 milliseconds, respectively. Tests for AWS were run from 10:50 p.m. to 12:30 a.m. and for Azure from 11:30 a.m. to 12:30 p.m. Therefore, a single graph is presented to cover the results for both services during the test duration.

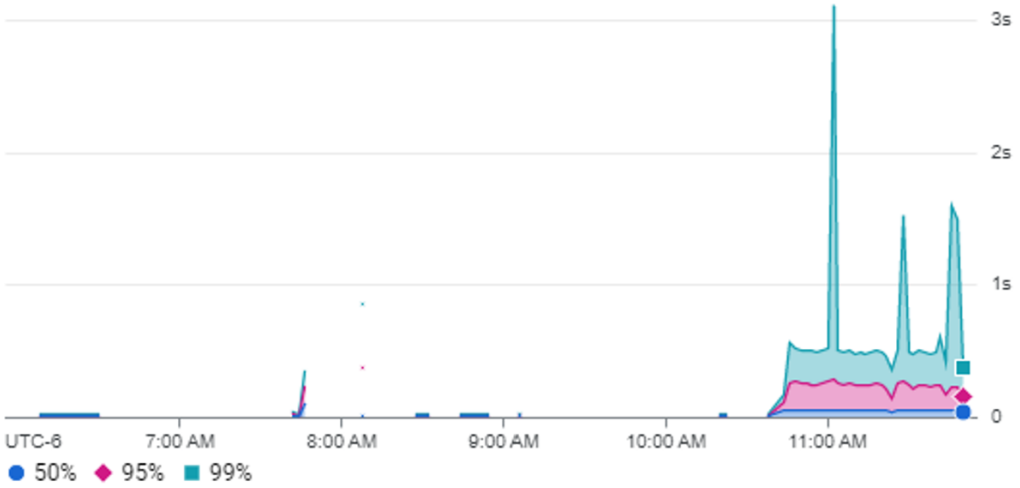

Figure 2. Request latency for Google Cloud Run.

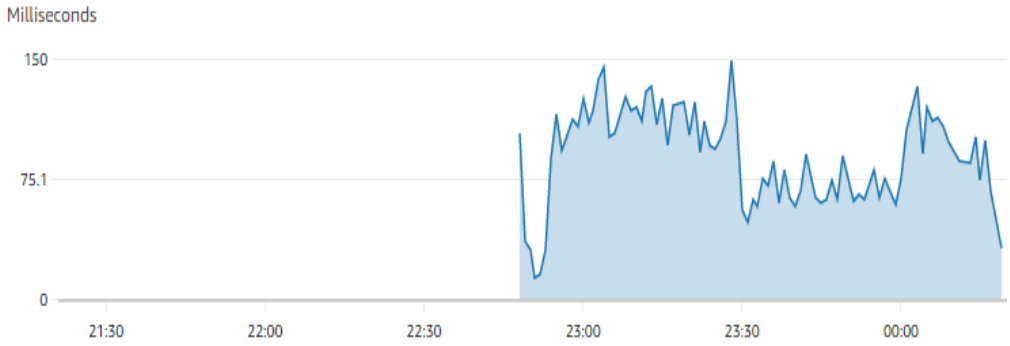

Figure 3. Request latency for AWS App Runner.

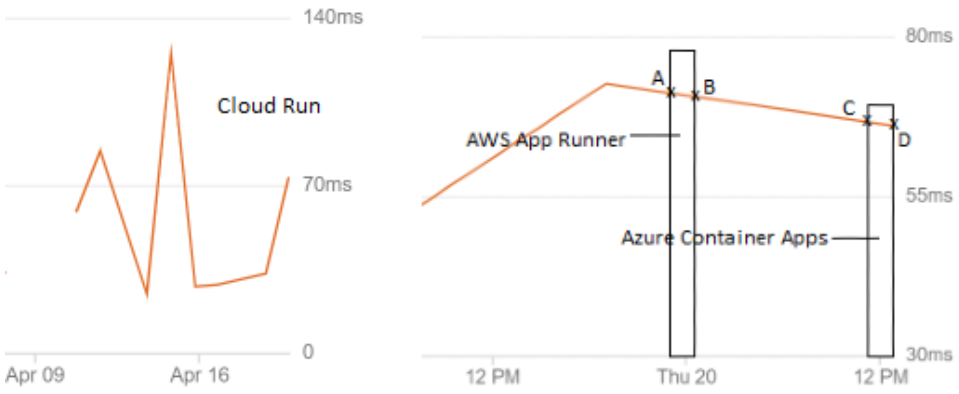

Figure 4. Response time for Google Cloud Run, AWS App Runner, and Azure Container Apps (A = 71.66, B = 71.11, C = 67.22, D = 66.38).

As shown in Figure 4 and Table 4, the average values of A and B represent the response time for AWS App Runner, while the averages of C and D represent the response time for Azure Container Apps. We observe that the better API response time when hosted on Google Cloud Run may be because the distance matrix API was also from Google, which may have helped in faster communication between the application and API. This result may not be surprising because the services from the same provider might have an upper hand as the communications between these services might be faster as compared to a different service outside the provider.

In terms of CPU utilization, Google Cloud Run uses an average of 23% (Figure 5). AWS App Runner uses the highest amount of CPU, with an average utilization of 47% (Figure 6). Azure Container Apps uses the lowest amount of CPU, with an average utilization of 0.000773% (Figure 7). Built-in monitoring tools automatically gathered data about CPU utilization based on the percentage of the allocated resources that were being used. As the Azure Container Apps monitoring graph shows the values in millicore, this value is converted to a percentage using Formula (5) below, as also indicated in Table 4. The lower CPU utilization values for Azure could be because of how the CPU was allocated for the application. Cloud Run and App Runner use the CPU the entire time that the container runs, whereas the Azure container apps request CPU only when it needs to process a request. Therefore, directly comparing this with Cloud Run and AWS is not right when considering the total container CPU usage.

$$percentage = \frac{number\ of\ cores}{10^6} \times 100 \tag{5}$$

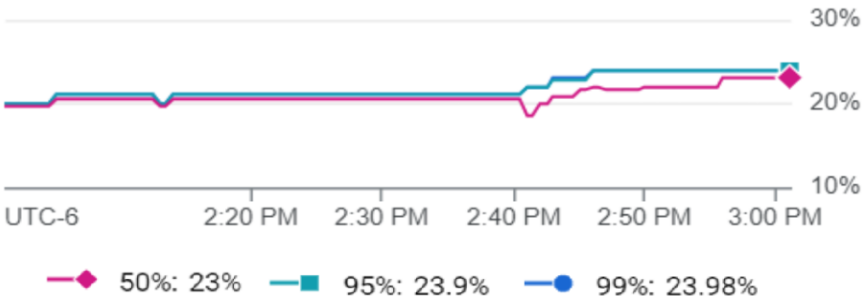

Figure 5. CPU utilization for Google Cloud Run.

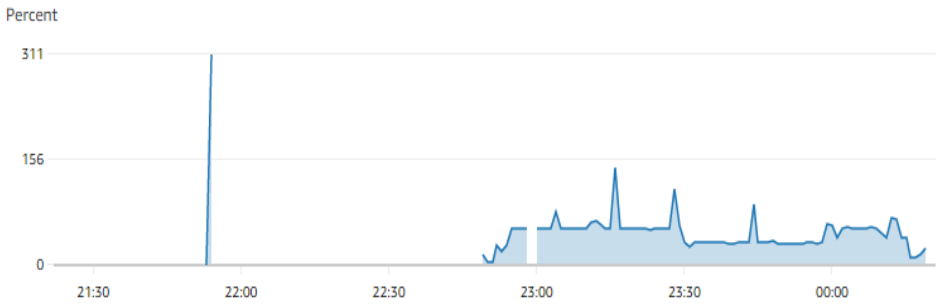

Figure 6. CPU utilization for AWS App Runner.

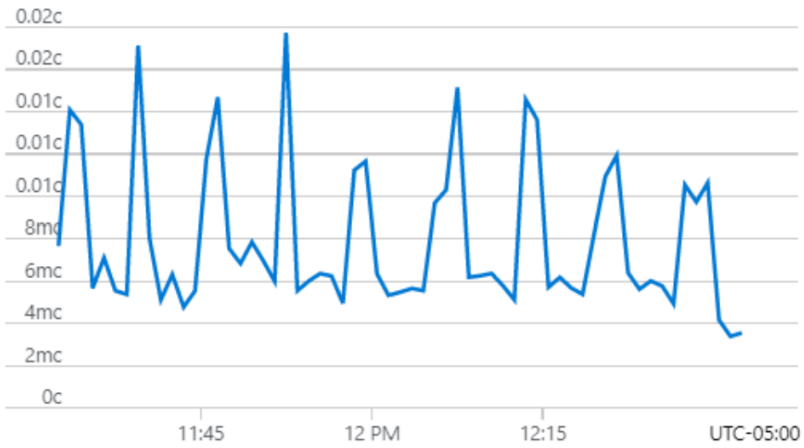

Figure 7. CPU utilization for Azure Container Apps.

5.3.2. Latency with Cold Starts

While the evaluation of performance metrics with container request latency, API response time, and CPU utilization provides valuable insights, these results can be further detailed by discussing additional factors that can influence overall performance. One such consideration with serverless cloud computing is cold starts, a system-level challenge faced when a serverless instance is initializing itself and its run-able code, typically exacerbated by serverless platforms that enact scale-to-zero policies [52]. However, one study notes [53] that the cold start problem also arises "when the auto scaler provisions additional instances to handle the traffic". This therefore means that cold starts are a near-unavoidable issue with serverless cloud computing and that optimizations must be considered to lessen the impact of cold starts that, when occurring, introduce latency to requests.

To better facilitate these optimizations, we might first consider understanding each of the platforms' cold start policies regarding when a container is set for recycling such that cold starts will now occur as new containers are spun up. This study proposes such an experiment to find these values [54], wherein a log file from a container's /tmp directory is read to determine whether a machine is cold or warm. Table 5 lists the results of this testing, showing how AWS appears to support less container reuse compared to GCP and Azure since AWS yielded a significantly higher percentage of cold starts. However, within [54]'s testing, there was a noticeable difference in cold start times for AWS, wherein the average function invocation time was lower for AWS, with an average 246.146 ms duration for invocation in contrast to GCP's 6284.838 ms duration. This therefore shows that AWS has likely applied its optimizations to favor cold starts, whereas GCP likely favors longer-living containers and avoiding cold starts if possible.

Table 5. Summary of cold-start testing (adapted from [54]).

Platform	Function Invocations	Cold Starts	Percentage of Cold Starts
AWS	175,477	156,847	89.383%
GCP	175,357	21,253	12.120%
Azure	36,176	1392	3.848%

Of note is that these findings contrast with those who found in [55] that, while Azure had a relatively consistent 20-min container recycling time and AWS had a steady preference for recycling containers given last-function invocation durations greater than 28 min, GCP had no discernible pattern and cold starts were observed to occur randomly from 3 min of this duration to 1 h. Given the prevalence of the shorter durations for which containers were observed to be enduring a cold start, this then challenges the claim made in the previous paragraph, which concludes that GCP favors keeping containers warm.

Since a cold start will occur within the lifetime of an application and thus introduce latency which an end user might potentially find undesirable, we must still consider the means by which we might optimize for cold start conditions. Byrro suggested the methods of performance and log monitoring for indicators of startup bottlenecks [56]: increasing the memory allocated to a container, choosing faster runtimes, keeping shared data in memory, shrinking the package size to contain only the minified and optimized code that is to be run, and keeping a pool of pre-warmed functions to ensure that, for some number of requests, there is that number of containers that are in a warm state and able to immediately handle those requests.

The suggestion of decreasing the package size when possible is further substantiated by Shilkov [57], who found that, when deploying containers of sizes 1 KB, 14 MB, and 35 MB, there was a nearly 5 s average difference in the cold start times of AWS and GCP and, for Azure, there was roughly 10 s of average startup duration difference. These differences show the importance of reducing the code bundle such that request latency is reduced, ushering in a better user experience.

Considering these optimizations, we might conclude that latency from cold starts, while a factor, is relatively manageable, the incidence of which may be controllable depending on the platform used and if tactics such as keeping containers warm are employed. There still do exist those times for which a cold start will occur and thus we might start by optimizing the size of the bundle used to deploy a container. This will likely be the lowest-cost solution to mitigating higher cold-start latency compared to other potential solutions such as increasing the amount of allocated memory or the runtime used.

6. Discussion

Following RQ1, we found that Google Cloud Run offers more memory and CPU options and supports more requests per container instance than AWS App Runner. Microsoft Container Apps has two pricing options and supports up to 300 instances per service. Ultimately, the choice of service will vary based on the individual needs of the user. Cloud Run may be the preferable option for those requiring greater memory and CPU capabilities. Regarding RQ2, as a beginner cloud service user, we found that Google Cloud has the most detailed and user-friendly documentation for their serverless service, with several third-party resources available online. AWS App Runner's documentation may be challenging for beginners, but there are still online resources available. Azure has a straightforward cloud console and clear documentation, but fewer online resources compared to the other two services. In accordance with RQ3, overall, the findings of the analysis revealed that Google's Cloud Run exhibited superior performance and usability compared to AWS's App Runner and Microsoft Azure's Container Apps on this particular application. Cloud Run demonstrated the lowest latency at 50 ms and a faster response time of 63.7 ms for distance matrix queries. These results provide valuable insights for individuals seeking to select an appropriate cloud service for similar containerized web applications. Google Cloud Run might be the best service to host a Django containerized web application with Google Maps and distance matrix API usage.

Limitations of Study

This paper explored the growing field of serverless cloud computing by comparing system features, usability, and performance across three leading platforms. While it provides an in-depth analysis with valuable insights for developers and organizations in selecting the most suitable service for similar application deployment, the findings may not be generalized. The evaluation of the three cloud platforms should be based on a comprehensive and objective assessment. Our ratings for documentation and learning curve, presented in Table 3, are derived solely from the authors' experience with the three serverless cloud services throughout the in-depth study, which includes the entire application deployment process as well as subsequent maintenance activities. It is important to note that these ratings may not necessarily reflect the objective views or opinions of others. The performance analysis was primarily conducted using the cloud monitoring service offered by Google, AWS, and Azure to measure the application's metrics. It is possible that lower CPU utilization values were due to insufficient load on the web application. In particular, the lower CPU utilization values for Azure Container Apps could be because of how the CPU was allocated for the application. Cloud Run and App Runner use CPU the entire time that the container runs, whereas the Azure Container Apps request CPU only when they need to process a request. Thus, directly comparing this with Cloud Run and App Runner is not right when considering the total container CPU usage. Also, the performance matrix values are specific to a containerized mobile web application with a heavy use of Google Maps APIs. These values may be different for other applications. Additionally, the container approach designed to track the real-time bus locations of San Antonio public buses on a specific route and provide expected arrival times at selected bus stops cannot be presented as an accurate evaluation. Since the analysis focused on the SmartSAT application, a mobile app for tracking bus locations as a practical use case, the findings may not be applicable to other types of applications or broader contexts within serverless computing.

7. Conclusions and Future Work

This research aimed to identify the key differences in system features, usability, and performance between Google Cloud Run, AWS App Runner, and Microsoft Azure Container Apps serverless cloud computing systems with respect to a containerized mobile web application. After thoroughly examining the features of serverless services offered by GCP, AWS, and Azure, it was observed that the Google Cloud Run service stands out with its extensive range of memory and CPU options. It also supports a greater number of containers, instances per service, and maximum concurrency per instance compared to its competitors. The AWS App Runner service and Azure Container Apps were found to be the next-best options. When it comes to the ease of use and usability of services, Google Cloud Run provides the easiest-to-understand documentation and educational materials for beginners. Azure and AWS also offer helpful resources, but Google Cloud Run stands out as the most beginner-friendly option.

From the performance analysis of the services, we found that the performance of the Google Cloud Run service was better in regards to container request latency and distance matrix API response time. Cloud Run exhibited the lowest latency at 50 ms, while AWS App Runner had a higher latency, with an average request latency of 91 ms. Azure Container Apps monitoring services cannot capture the container latency for the applications. Cloud Run also showed a faster response time of 63.7 ms for distance matrix queries while App Runner and Container Apps have an average response time of 71.38 and 66.8 ms respectively. Meanwhile, Azure Container Apps exhibited the lowest CPU memory utilization with an average utilization of 0.000773% compared to the other two options. Cloud Run uses an average of 23% and AWS App Runner uses the highest amount of CPU, with an average utilization of 47%.

From these findings, our suggestion is to opt for GCP as the primary choice, followed by Azure, for those developers who are new to the concept of serverless computing. On the other hand, AWS is a suitable option for developers already well-acquainted with serverless computing. These key contributions will help future academic researchers and developers when deciding to choose a serverless cloud platform for deploying similar containerized applications. The analysis of performance results in this comparative study was focused solely on the particular mobile web application. However, there is potential for conducting further research on other application domains and exploring a broader range of cloud providers beyond those examined in this study.

While the findings of the study are practical, the results have implications for future research and development practices in serverless cloud computing. First, to support the study's findings more applicably and objectively to other types of applications or broader contexts in serverless computing, a more thorough examination of factors affecting performance differences such as platform architecture, underlying infrastructure, and/or specific optimization in future studies could enhance the depth of analysis and generalization. A detailed discussion on network effects and the impact of scaling on performance measures along with a more rigorous statistical analysis could further ensure the reliability of the results. Second, including a balanced discussion with challenges and disadvantages such as cost implications, security concerns, and limitations in specific use cases would offer a more comprehensive perspective. Additionally, a subjective analysis of each platform's documentation and learning curve with surveys and experiences from a broader range of users could strengthen the usability study.

Author Contributions: All authors contributed equally to this project: writing—original draft, software, investigation, A.A.; conceptualization, methodology, writing—review and editing, supervision, funding acquisition, J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This material is based upon work supported by the National Science Foundation's Grant No. 2131193. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of National Science Foundation.

Data Availability Statement: Data are contained within the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CSF	Cybersecurity Framework
DevOps	Development Operations
ECS	Elastic Container Service
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
ISO	International Organization of Standardization
NIST	National Institute of Standards and Technology
GCP	Google Cloud Computing
GPS	Global Positioning System
SmartSAT	Smart San Antonio Transit
SQL	Structured Query Language
TLS	Transport Layer Security
VM	Virtual Machine
VPC	Virtual Private Cloud

References

1. Surbiryala, J.; Rong, C. Cloud computing: History and overview. In Proceedings of the 2019 IEEE Cloud Summit, Washington, DC, USA, 8–10 August 2019; pp. 1–7. [CrossRef]
2. Garfinkel, S. *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT*; MIT Press: Cambridge, MA, USA, 1999.
3. IBM Cloud Infrastructure Center. A Brief History of Cloud Computing. Available online: https://www.ibm.com/cloud/blog/cloud-computing-history (accessed on 5 June 2023).
4. Mell, P.; Grance, T. *The Nist Definition of Cloud Computing*; NIST: Gaithersburg, MD, USA, 2011.
5. Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.-C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.; Krauth, K.; Yadwadkar, N.; et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv* **2019**, arXiv:1902.03383.
6. Yu, R.; Yang, X.; Huang, J.; Duan, Q.; Ma, Y.; Tanaka, Y. QoS-aware service selection in virtualization-based cloud computing. In Proceedings of the 2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS), Seoul, Republic of Korea, 25–27 September 2012; IEEE: New York, NY, USA, 2012; pp. 1–8.
7. Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [CrossRef]
8. Callegati, F.; Cerroni, W.; Contoli, C.; Santandrea, G. Performance of Network Virtualization in cloud computing infrastructures: The OpenStack case. In Proceedings of the 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet), Luxembourg, 8–10 October 2014; IEEE: New York, NY, USA, 2014; pp. 132–137.
9. Seibold, M.; Wolke, A.; Albutiu, M.; Bichler, M.; Kemper, A.; Setzer, T. Efficient deployment of main-memory DBMS in virtualized data centers. In Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, 24–29 June 2012; IEEE: New York, NY, USA, 2012; pp. 311–318.
10. Shea, R.; Liu, J. Performance of virtual machines under networked denial of service attacks: Experiments and analysis. *IEEE Syst. J.* **2012**, *7*, 335–345. [CrossRef]
11. Barik, R.K.; Lenka, R.K.; Rao, K.R.; Ghose, D. Performance analysis of virtual machines and containers in cloud computing. In Proceedings of the 2016 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 29–30 April 2016; pp. 1204–1210. [CrossRef]
12. Rosati, P.; Lejeune, A.; Emeakaroha, V. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 11–14 December 2017; IEEE: New York, NY, USA, 2017; pp. 162–169.
13. Nupponen, J.; Taibi, D. Serverless: What it is, what to do and what not to do. In Proceedings of the 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 16–20 March 2020; IEEE: New York, NY, USA, 2020; pp. 49–50.

14. Sarathi, T.V.; Reddy, J.S.N.; Shiva, P.; Saha, R.; Satpathy, A.; Addya, S.K. A Preliminary Study of Serverless Platforms for Latency Sensitive Applications. In Proceedings of the 2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, 8–10 July 2022; IEEE: New York, NY, USA, 2022; pp. 1–6.
15. Al-Ali, Z.; Goodarzy, S.; Hunter, E.; Ha, S.; Han, R.; Keller, E.; Rozner, E. Making serverless computing more serverless. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; pp. 456–459.
16. Wu, M.; Mi, Z.; Xia, Y. A Survey on Serverless Computing and Its Implications for JointCloud Computing. In Proceedings of the 2020 IEEE International Conference on Joint Cloud Computing, Oxford, UK, 3–6 August 2020; pp. 94–101. [CrossRef]
17. Li, Y.; Lin, Y.; Wang, Y.; Ye, K.; Xu, C. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Trans. Serv. Comput.* **2022**, *16*, 1522–1539. [CrossRef]
18. Rajan, R.A.P. Serverless architecture-a revolution in cloud computing. In Proceedings of the 2018 Tenth International Conference on Advanced Computing (ICoAC), Chennai, India, 13–15 December 2018; pp. 88–93.
19. Eivy, A.; Weinman, J. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Comput.* **2017**, *4*, 6–12. [CrossRef]
20. Van Eyk, E.; Toader, L.; Talluri, S.; Versluis, L.; Uță, A.; Iosup, A. Serverless is more: From paas to present cloud computing. *IEEE Internet Comput.* **2018**, *22*, 8–17. [CrossRef]
21. Garde, A.; Gandhale, S.; Dharankar, R.; Sangtani, B.S.; Deshmukh, N.; Deshpande, S.; Rathi, R.; Srivastava, A. Serverless data protection in cloud. In Proceedings of the 2023 6th International Conference on Information Systems and Computer Networks (ISCON), Mathura, India, 3–4 March 2023; pp. 1–6.
22. Mileski, D.; Gusev, M. Serverless implementations of real-time embarrassingly parallel problems. In Proceedings of the 2022 30th Telecommunications Forum (TELFOR), Belgrade, Serbia, 15–16 November 2022; pp. 1–4.
23. Veuvolu, R.; Suryadevar, A.; Vignesh, T.; Avthu, N.R. Cloud computing based (serverless computing) using serverless architecture for dynamic web hosting and cost optimization. In Proceedings of the 2023 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 23–25 January 2023; pp. 1–6.
24. Sewak, M.; Singh, S. Winning in the era of serverless computing and function as a service. In Proceedings of the 2018 3rd International Conference for Convergence in Technology (I2CT), Pune, India, 6–8 April 2018; pp. 1–5.
25. Kumar, N.S.; Selvakumara, S.S. Serverless computing platforms performance and scalability implementation analysis. In Proceedings of the 2022 International Conference on Computer, Power and Communications (ICCPC), Chennai, India, 14–16 December 2022; pp. 598–602.
26. Savage, N. Going serverless. *Commun. ACM* **2018**, *61*, 15–16. [CrossRef]
27. Rodriguez Cortes, L.M.; Paul Guillen, E.; Rojas Reales, W. *Serverless Architecture: Scalability, Implementations and Open Issues*; IEEE: New York, NY, USA, 2022.
28. Team, G.C. What Is Cloud Run. Available online: https://cloud.google.com/run/docs/overview/what-is-cloud-run (accessed on 8 June 2023).
29. Team, G.C. The Most Scalable and Fully Automated Kubernetes Service. Available online: https://cloud.google.com/kubernetes-engine/ (accessed on 8 June 2023).
30. Abraham, A.; Yang, J. A comparative analysis of performance and usability on serverless and server-based google cloud services. In *Proceedings of the 2023 International Conference on Advances in Computing Research (ACR'23)*; Daimi, K., Al Sadoon, A., Eds.; Springer: Cham, Switzerland, 2023; pp. 408–422. [CrossRef]
31. Google Cloud Run, Security Design Overview. Available online: https://cloud.google.com/run/docs/securing/security (accessed on 2 December 2024).
32. AWS. AWS App Runner. Available online: https://aws.amazon.com/apprunner/ (accessed on 2 December 2024).
33. AWS. AWS App Runner Features. Available online: https://aws.amazon.com/apprunner/features/?refid=12eea001-bcfd-40ce-9788-748f73400e32 (accessed on 2 December 2024).
34. AWS App Runner, Security in App Runner. Available online: https://docs.aws.amazon.com/apprunner/latest/dg/security.html (accessed on 2 December 2024).
35. Azure. Azure Container Apps Overview. Available online: https://learn.microsoft.com/en-us/azure/container-apps/overview (accessed on 2 December 2024).
36. Azure Security, Manage Secrets in Azure Container Apps. Available online: https://learn.microsoft.com/en-us/azure/container-apps/manage-secrets?tabs=azure-portal (accessed on 2 December 2024).
37. What Is Azure Load Balancer? Available online: https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview (accessed on 2 December 2024).
38. Azure. Quotas for Azure Container Apps. Available online: https://learn.microsoft.com/en-us/azure/container-apps/quotas (accessed on 28 June 2023).
39. Yang, J.; Lee, Y.; Noon, W.; Abraham, A. Demo abstract: Smartsat—A customizable secure app for san antonio transit pilot project. In Proceedings of the 20th ACM International Symposium on Mobility Management and Wireless Access. MobiWac'22, Montreal, QC, Canada, 24–28 October 2022; pp. 123–127; Association for Computing Machinery: New York, NY, USA, 2022. [CrossRef]
40. Randal, A. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Comput. Surv.* **2020**, *53*, 1–31. [CrossRef]

41. Marquez, E. The History of Container Technology. Available online: https://www.pluralsight.com/resources/blog/cloud/history-of-container-technology (accessed on 7 July 2023).
42. Turnbull, J. *The Docker Book: Containerization Is the New Virtualization*; James Turnbull: Brooklyn, NY, USA, 2014; pp. 6–9.
43. Pahl, C. Containerization and the paas cloud. *IEEE Cloud Comput.* **2015**, *2*, 24–31. [CrossRef]
44. Cdockersdocs. Docker Overview. Available online: https://docs.docker.com/get-started/overview/ (accessed on 28 October 2024).
45. Google Cloud.Running Django on the Cloud Run Environment. Available online: https://cloud.google.com/python/django/run (accessed on 28 October 2024).
46. Lee, Y.; Yang, J.; Al-Ramahi, M.; Delgado, D. SmartSAT: A customizable mobile web application toward improving the efficiency and equitable access of San Antonio's public transit services. *Softw. Impacts* **2024**, *22*, 100714. [CrossRef]
47. Azure. Generally Available: Azure Container Apps. Available online: https://azure.microsoft.com/en-us/updates/generally-available-azure-container-apps/ (accessed on 8 June 2023).
48. *Ergonomics of Human-System Interaction—Part 110: Interaction Principles*; International Organization for Standardization: London, UK, 2018.
49. Wikipedia. ISO 9241. Available online: https://en.wikipedia.org/wiki/ISO9241 (accessed on 11 July 2023).
50. Yang, J.; Lee, Y.; Gandhi, D.; Valli, S.G. Synchronized UML diagrams for object-oriented program comprehension. In Proceedings of the 2017 12th International Conference on Computer Science and Education (ICCSE), Houston, TX, USA, 22–25 August 2017; pp. 12–17. [CrossRef]
51. *ISO 9241-11:2018*; Ergonomics of Human-System Interaction—Part 11: Usability: Definitions and Concepts. Chap. 3.1.1.; ISO: Geneva, Switzerland, 2018.
52. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*; Springer: Singapore, 2017; pp. 1–20.
53. Lin, P.-M.; Glikson, A. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv* **2019**, arXiv:1903.12221.
54. Kelly, D.; Glavin, F.; Barrett, E. Serverless Computing: Behind the Scenes of Major Platforms. In Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 18–24 October 2020; pp. 304–312. [CrossRef]
55. Shilkov, M. Serverless: Cold Start War. 2018. Available online: https://mikhail.io/2018/08/serverless-cold-start-war/ (accessed on 28 October 2024).
56. Byrro, R. Can We Solve Serverless Cold Starts. 2019. Available online: https://dev.to/byrro/can-we-solve-serverless-cold-starts-39hi (accessed on 28 October 2024).
57. Shilkov, M. *Comparison of Cold Starts in Serverless Functions Across AWS, Azure, and GCP. 2021*. Available online: https://mikhail.io/serverless/coldstarts/big3/ (accessed on 28 October 2024).