

Evaluating Serverless Function Deployment Models on AWS Lambda

Gabriel Duessmann and Adriano Fiorese^a

*Postgraduate Program in Applied Computing, Department of Computer Science,
Santa Catarina State University, Joinville, Brazil*
fi

Keywords: Serverless Function, AWS Lambda, Container Image, Evaluation, Performance, Cost, Cold Start Time, Metrics and Measurement.

Abstract: With the advancement of computing and serverless services in the last couple of years, this area has been growing rapidly. Currently, most cloud providers offer serverless services, in particular at Amazon, they have AWS Lambda to create Functions as a Service (FaaS). There are at least two ways to implement it: by compressing the source code and files into a compacted folder in a ZIP format; the second way is through a container image, which has the running application and its dependencies. Based on the approach selected, the function's performance, cost and initialization time may vary. This paper takes into account these metrics and compares the aforementioned ways of deployment. Furthermore, it aims to discover which approach is the most adequate. Experiments conducted at AWS Lambda show that functions created with compressed ZIP folders present advantages, regarding their initialization time during cold start mode, and cost.

1 INTRODUCTION


Serverless is a computing service in which cloud providers offer a dynamic provisioning service with pre-configured servers for their customers to run their applications. Therefore, the cloud providers are responsible for provisioning, scaling, and securing the applications deployed in this model (Nupponen and Taibi, 2020). This makes it easier for developers and organizations to implement their applications without the burden of infrastructure management, that would require hiring staff and maintaining the hardware needed for the application. Each application deployed in this model is called a function, which must be executed independently in the infrastructure offered by the provider.

When comparing the latest serverless models with monolithic applications that have been used broadly in the market for years, they present differences in how they are structured and deployed. Serverless functions have a smaller scope of code and functionalities, as there is no need for server configurations to be set and they auto-scale automatically according to the demand of the allocated resources. Monolithic applications, however, comprise all the code of a system, including server and database configurations, and therefore their codes and structures tend to be

more extensive. Auto-scaling is not present as a standard and might be cumbersome to implement due to the large amount of computational resources needed for the application.

Despite the ease of development offered by serverless functions, in the way of abstracting the infrastructure necessary to execute and handle the elastic demand of the application, deploying a serverless application on cloud providers requires attention, as environment configurations are managed by the provider, which limits the developers' access to modify them. This attention concerns configuring the resources required for their execution, as well as how the application is instantiated and deactivated based on demand (requests made by users) and idleness (period without requests). As the application becomes idle for a few minutes, the provider shuts down the function's computational resources, rendering it inoperative. When it is in this state, and a new user request is made to the function, the service allocates its resources again; this step is known as cold start. During this time, the response time also takes longer on the first request.

Via AWS Lambda website's interface, there are a few ways to deploy the application as a serverless function. Particularly, two models were selected for this work: through a compressed folder, and a container image. Given these deployment models, this paper aims to evaluate them through experiments con-

^a  <https://orcid.org/0000-0003-1140-0002>

ducted on the platform and analyse their characteristics based on the collected data. The evaluation metrics selected to perform the analysis are: cost, performance, and cold startup time. As a scientific contribution, this work seeks to address the following research questions regarding the two models analysed:

- **RQ-1.** Which one presents advantages in terms of performance and initiation time during cold start?
- **RQ-2.** How can the deployment choice impact the function's cost?

The answers to these questions will contribute to researchers, developers, and cloud engineers who want to leverage Lambda functions and deploy their applications on the cloud. The relevant metrics will indicate which deployment option presents to be the most adequate for AWS Lambda though not limited to this specific cloud provider as the experiments herein can be replicated to other platforms.

For the experimentation environment, a Java application was developed and deployed as a serverless function in both deployment models. The functions expect as input in the request a geographical region on Earth; then, it executes the application code (function) to obtain the local date and time; and finally, returns the result. The execution metrics data were collected by the AWS Lambda service itself, which provides a tab for testing. In this tab, not only the users can run their tests with different parameters, but also visualize the functions' metrics at each request.

Running those experiments incurred no costs, as the experimental AWS Lambda Free Tier access was used. This access allows running and testing several AWS services for free (AWS, 2024a). Thus, the cost analysis and comparison were based on the estimated values provided by AWS Pricing Calculator (AWS, 2024b).

This paper is structured as follows: Section 2 presents the theoretical framework to clarify terms related to the cloud and AWS provider. Section 3 lists related works relevant to the topic. Section 4 presents the methodology, environment configurations, the evaluation proposal, as well as the experiments performed, and the results of the garnered metrics. Finally, Section 5 concludes the evaluation and answers the research questions.

2 THEORETICAL FRAMEWORK

Cloud providers are dedicated to improving the quality and usability of their services to serve a greater number of customers. Serverless is one of the services widely available at most cloud providers, that runs ap-

plications without server and infrastructure configurations.

This service inception was only possible with the countless improvements of related technologies that helped reaching its current state. Such technologies are highlighted in this Section to provide an overview on their correlated connections.

2.1 Serverless Function

The serverless computing model, known as serverless functions or Function as a Service (FaaS) model, is a cloud computing service model in which providers make pre-configured environments available in various programming languages to run cloud applications. These environments are configured to primarily meet the elastic demand of the application, ensuring the smooth execution of functions. Yet, it is necessary that the serverless users set up the initial configuration regarding the hardware components and specific specifications, whereas applying and controlling more robust configurations is managed by the cloud provider.

One common strategy to manage resource allocation applied by AWS, as well as other cloud providers, is known as cold start. As the function stops being called by end users, the allocated hardware resources become idle. Thus, after a while, the provider deallocates these resources to alleviate the usage of its computational resources.

Once these resources are released, they can also be used in other serverless functions owned by different clients. As new user requests are made to the function, the provider re-instantiates the resources and goes into a warm-start mode, which invokes immediately the serverless instance upon client incoming requests.

The cold start is typically considered a disadvantage. Although not the focus of this paper, there are studies that aim at finding optimizations to help reduce it in order to avoid latency delays to the end users. Nevertheless, there are challenges in achieving such solutions since most of the allocation and resource configurations are managed by the cloud providers (Vahidinia et al., 2020; Kumari et al., 2022; Vahidinia et al., 2023; Dantas et al., 2022), and are not available for clients to try their own experiments and optimization.

2.2 Containers

Containers provide applications with a computing environment for running applications, configured with all their dependencies to be executed. Containers iso-

late the desired application from external programs and processes that run on the machine's host operating system. Thus, it can be defined as an application packaging mechanism (Siddiqui et al., 2019).

In order to be used, an application image must first be created with its specifications. Then, a container is built on top of that image, like a self-contained machine that runs the application. This container image with the specifications can be shared among different hosts, and each machine can build their own running container with the same configurations, regardless of hardware and operating system (OS). This is possible because containers are a form of lightweight virtualization, which can include its own OS (Scheepers, 2014).

The use of containers has become widely popular due to their portability to migrate applications among different environments, without the occurrence of problems caused by different configurations in distinct machines. To be portable, the host machine must have its operating system prepared to run containers. Generally, an installation and configuration of a container manager software is required for those OSs that do not offer it out of the box.

2.3 AWS Services

One of the largest cloud providers known nowadays is Amazon Web Service (AWS), which offers hundreds of services available on the internet. This work uses the following services: AWS Lambda and AWS ECR (Elastic Container Registry). Amazon's serverless service is called AWS Lambda (AWS, 2024g), and it allows creating functions to run applications without having to provision and manage servers.

The Lambda service manages most of the computing configuration, which provides computing resources for memory, CPU, network, and necessary resources to run the application code (function). Functions are instantiated on demand based on user requests, prompting the cloud provider to dynamically allocate computing resources from its infrastructure to meet the demand. As functions become idle for a few minutes, allocated computing resources are deactivated. This allows customers to pay only for the time the application is active and being invoked, with the resources allocated to the serverless function (AWS, 2024g).

AWS provides several ways to deploy (create) serverless functions through its website in a visual and intuitive manner. In particular, as this work's object of study, the available and analysed models are: compressed folder in ZIP format and container image. There is a third model, through the IDE (Inte-

grated Development Environment) integrated into the website on AWS. However, it was not analysed since browser IDEs are not as intuitive and do not facilitate development as much as traditional IDEs, and therefore are not frequently used by developers and companies in the software development sector. Another factor that led to this choice is that the integrated IDE is only available for a few programming languages on the platform (NodeJS, Python and Ruby), which limits its scope of use.

The hardware architecture where the serverless service will be executing can be chosen when creating the function. It can be either under the *arm64* or *x86_64* architecture, and as a default it is pre-selected *x86_64*, though can be changed to the *arm64* architecture, which stands out for having a lower execution cost and achieving good performance results (AWS, 2024c).

AWS ECR is the repository service for storing container images. Some of the container tools that can be used for this purpose are: Podman (AWS, 2024e) and Docker (AWS, 2024d). Given Docker's great popularity, it was opted as the container tool for the experiments. The developer must create the image to run the application on their local machine from a *Dockerfile* file and publish it to AWS ECR (AWS, 2024f), to become available on AWS. By having the image available in the AWS ECR repository, it can be used in several of the provider's services, and specifically for this work, to create functions in AWS Lambda.

3 RELATED WORKS

FaaS models are not particularly new, and though bring ease to the implementation of applications, there is still room for studies to analyse areas for improvement.

The work of (Dantas et al., 2022) addresses strategies to reduce cold start and compares the impact on time when instantiating a function through a compressed file and via a container image. Despite proposing solutions to minimize the init time, the cited work does not address cost.

It is by evaluated by (Elsakhawy and Bauer, 2021) the factors that affect the performance of serverless functions, the results of container options, different programming languages, and compilation alternatives. However, the authors do not take into account the cost to execute the function and the init time of the cold start.

The authors (Villamizar et al., 2017) compared the costs of running applications in monolith, microser-

vice and Lambda function on AWS. Among the three architectures, AWS Lambda had the lowest cost, reducing infrastructure costs by 70%.

A case study conducted by (Rodríguez et al., 2024) evaluated memory usage, scalability, and cold start according to the choice of programming languages on AWS Lambda. The experiments were conducted based on CRUD (create, read, update and delete) operations, and the results showed that Python and NodeJS obtained the best results. Nevertheless, the deployment model is not highlighted.

The paper (Maharjan, 2022) does a performance analysis on AWS Lambda, and evaluates the performance, cost, latency, and cold startup overhead in the two architectures available: *x86_64* and *arm64*. For the experiments, applications with different complexity levels were assessed to evaluate how they impacted the functions' performance. This work found that memory usage is nearly identical for both architectures, but the input size impacts directly the performance and cost of the function. Moreover, the paper concluded that *arm64* performed better than *x86* for their experiments.

The author (Bagai, 2024) carries out a meticulous comparison among different deployment services in AWS, highlighting the strengths and weaknesses of AWS SageMaker, AWS Lambda, and AWS ECS (Elastic Container Service). AWS Lambda was nominated as the best choice for event-driven architectures and cheaper tasks. The main factors taken into account are: performance, scalability, customization, and cost. Cold start is not discussed as it is only particular to AWS Lambda.

Based on the related papers that compared deployment models and metrics evaluations, this one stands out for garnering the most relevant metrics in AWS Lambda for two deployment models and evaluating them, to find out how a set of configurations may be more suitable than another. To do so, metrics of performance, cost, and init time during cold start were evaluated.

Table 1 compares the characteristics of related papers and highlights how this paper differs from the others. The compressed file and container image columns refer to the application deployment mode. The following columns present performance, cost and init time as metrics related to the function, and finally the programming languages used in the paper's experiments.

4 METHODOLOGY AND EXPERIMENTS

The performed experiments aim to help discover which deployment method on AWS Lambda is more advantageous and corroborate to the research questions (RQ-1 and RQ-2). To reach these conclusions, metrics relevant to the service were analysed, namely: cost, performance based on memory consumption, and init time during cold startup.

4.1 Function and Environment

First, an application was developed to be deployed as a serverless function. This application expects a geographic region as an input parameter (e.g.: *Europe/Berlin*). With this input, it obtains the current date and time of the region, and returns them as a response. The application was developed in the Java 21 programming language. This choice was made due to its popularity in the web environment and the ease of implementing the library interface required to implement the application as a function on AWS Lambda service.

Since the primary goal of this work is not to compare the use of computational resources (processing and memory, without disk usage) between functions, a relatively simple function was developed, which required low usage of computational resources. Therefore, it was possible to allocate the minimum available hardware resources (128 MB of memory) to the function in the AWS Lambda service, without impacting the results of the performed experiments.

The steps for deploying a function in the AWS Lambda service are similar, though have some particularities:

Compressed File. An executable file of the application is generated, which for the Java language is called JAR (Java ARchive). The folder containing the source code, along with the JAR file, must be compressed in ZIP format. When creating the function in the AWS Lambda service, this folder must be uploaded directly to the service.

Container Image. An extra file (usually named *Dockerfile*, for the Docker-type container) is added to the project containing the configurations and dependencies to generate the build of the container image. It contains the instructions to install the operating system and all environment dependencies. The image generated in the developer's local machine must be published to the AWS ECR service (image repository service) so that it becomes accessible within the cloud. When creating the function, the container image option must be chosen and the respective link

Table 1: Comparison of Related Works.

Paper	Compressed Folder	Container Image	Performance	Cost	Init Time	Languages
(Dantas et al., 2022)	Yes	Yes	No	Yes	Yes	NodeJS, Python and Java
(Elsakhawy and Bauer, 2021)	No	Yes	Yes	Yes	No	-
(Villamizar et al., 2017)	No	No	No	Yes	Yes	Java 7
(Rodríguez et al., 2024)	No	No	Yes	No	Yes	NodeJS, Python and Java
(Maharjan, 2022)	No	No	Yes	No	Yes	Python 3.x
(Bagai, 2024)	No	No	Yes	Yes	No	-
Current work	Yes	Yes	Yes	Yes	Yes	Java 21

must be provided. This link is a unique reference to the image available on AWS ECR service, and in this way, the serverless function is created based on this image.

Moreover, it is necessary to set up some computing resource configurations. In particular for this work, only the architecture and memory configurations were chosen. For the architecture, *arm64* was selected due to its lower cost (AWS, 2024c), as well as the application's low processing requirement. The memory chosen was the minimum available, 128 MB, since the functions do not require ephemeral storage or large computing memory to operate effectively. As for the CPU, it is allocated automatically and proportionally to the chosen memory, with no option to be altered.

To perform the experiments, the test console available within the AWS Lambda service through the AWS website was used. When viewing the function settings, there is a tab to test it, allowing to pass input parameters and view the output. On top of that, the test console provides the function's metrics for each invocation, which are collected for this work's analysis. As it is a service's internal tool, the actual metrics' values obtained at each execution are not affected by external factors, such as the user's internet connection, or the round-trip time of the request between the source and destination path.

Figure 1 shows a diagram with the test flow, as well as the configured environment. In this scenario, initially, the end user makes the request to a function (through the test console) by means of its browser, the function executes the application (either via a compressed folder or container image) and returns the result to the end user, as well as the metrics for that particular execution.

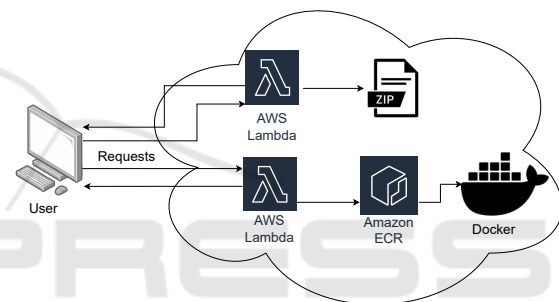


Figure 1: Test environment diagram.

To comprise the data garnered in the experiments, 40 calls were made to functions in cold start mode, a state in which the function needs to reallocate its resources, and consequently has a greater delay in the response time. The work of (Manner et al., 2018) empirically estimated 20 minutes for an idle serverless function to go from warm start-up mode to cold start. Adding an extra tolerance of 10 minutes, the experiment requests were made at 30-minute intervals between each invocation.

4.2 Cost

To run these experiments on AWS, including the services used, there was no charge from the provider as only services and configurations within the *Free Tier* level were used. This level allows customers to use services for free, as long as they meet each service's constraints (AWS, 2024a). Therefore, to compare the cost between the two deployment approaches, the AWS cloud provider's base pricing values are used.

The factors that can affect the costs of the AWS Lambda service are: the choice of function deployment model, the environment configuration, the num-

ber of calls or requests to the function, the execution time per request, and the geographic region where the function is allocated. Among these factors, the number of requests and the execution time are disregarded in the analysis, as they are very specific to each implementation and business rules in place.

The cost analysis related to the function deployment model can be divided between applications with sizes of up to 10 MB and larger. The AWS Lambda service offers applications that fall into the first division (up to 10 MB) in compressed folder (ZIP) format, which can be uploaded directly to the service and are free of storage costs. Applications larger than 10 MB or generated through container images (regardless of size) must be stored in their respective storage service. For compressed folders, AWS offers S3 (Simple Storage Service); and AWS ECR for container images. Both AWS S3 and ECR services price their storage cost based on the size of the folder or container image. It is worth noting that container images tend to be larger in size, as they comprise the environment configurations required to execute the function, including the operating system, and as a consequence, may increase the final cost.

Functions in AWS Lambda are also charged according to the choice of computing resources allocated to the environment. The resources are: architecture (*arm64* or *x86_64*), available memory (between 128 MB and 10,240 MB), and ephemeral storage (between 512 MB and 10,240 MB). Among the architectures offered, *arm64* has a lower cost (AWS, 2024c). For memory allocation and ephemeral storage, the cost is proportional to the allocated size.

In general, the geographic zone selected for provisioning cloud services influences the total costs; however, it applies to all services offered by the provider (not just AWS Lambda), and since this work only used the *us-east-1* region (located in Virginia, USA), it cannot be used as a basis for comparison between both deployed models. Moreover, if they were available in different regions, it would also be unfair to make such a comparison, since the prices vary by region.

4.3 Performance

As a performance metric for the application, the work took into account the maximum RAM memory consumption in the environment tested when executing serverless functions that were in cold start mode.

As shown in Figure 2, the maximum memory usage with the container image approach remained constant, whereas the compressed folder had a larger standard deviation, which indicates that the function

in the container image presents a more stable memory usage management.



Figure 2: Maximum memory usage graph in serverless functions.

Figure 3 shows the average consolidation of memory usage for both approaches, corroborating the results of Figure 2. Even more, it demonstrates that the deployment with the container image obtained better results regarding memory consumption; that is, it consumes less memory to execute the function. However, the difference is too small to be considered relevant.

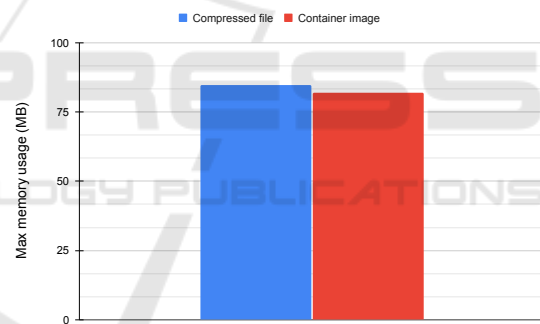


Figure 3: Graph of average maximum memory usage in serverless functions.

The difference in memory usage of the functions between the two deployment models, though relatively low, does not change the final cost, since pricing is based on the allocated memory, which in the experiments was 128 MB, and the memory used during execution period is not considered.

4.4 Init Time in Cold Start

Figure 4 shows the initialization time during cold start of the functions in each deployment model, considering 40 executions, called with an interval of 30 minutes between each. This ensures that the functions halted their active state and went into cold start. In particular, can observe the consistency in the results obtained for the functions in the compressed folder,

the opposite result obtained in Figure 2 for memory usage, which may indicate that the AWS provider performs optimizations in the environment for functions deployed in this model, providing greater stability and achieving shorter initialization times.



Figure 4: Init time graph in serverless functions.

Figure 5 shows the average init time for cold startup. This metric is directly linked to cost, since the execution time is one of the pricing factors for the AWS Lambda service, and the longer the init time, the longer the total execution time. Consequently, functions with shorter init times have a lower cost, as well as better response time to end users. Based on the experiments, the compressed folder deployment model presented the best init time.

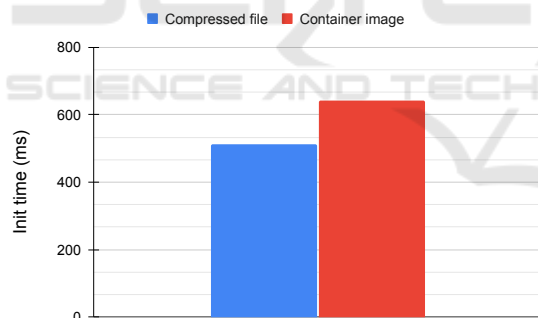


Figure 5: Graph of average init time in serverless functions.

5 CONCLUSIONS AND FUTURE WORK

This paper assessed two models for deploying serverless functions on AWS Lambda. In the compressed folder (ZIP) model, the deployment model is simplified, as the application folder is uploaded directly to the service. For a container-image-base deployment, additional steps are included, such as generating a build image for the application and publishing it to AWS ECR. Furthermore, two research questions were initially introduced as the motivation for this study,

and are now answered in this section.

RQ-1. Sections 4.3 and 4.4 presented the performance results, respectively, analyzing memory usage and init time. The average performance between the models showed an insignificant difference to be considered as meaningful. The init time, however, showed that the function deployed through a compressed folder obtained better results, contributing to a faster response time in executions. According to the evaluation conducted, the compressed folder function proved to be the most advantageous option.

RQ-2. In Section 4.2, the pricing of the AWS Lambda service was evaluated based on the values reported on the platform. Thus, it was possible to infer that the compressed folder model presented advantages, as no storage service is needed for functions whose maximum size is 10 MB, and ZIP folders tend to have a smaller size. In Section 4.4, it was discussed how the init time impacts the final cost of executing the functions.

For the experiments conducted on AWS Lambda, an application in Java was developed and deployed in both models studied. The application had plain requirements, with the main purpose of running a serverless function. With the function deployed, requests were made within the service testing section, and its execution metrics were collected for the current paper's analysis.

Those experiments corroborate with the research questions, regarding the metrics of performance, init time, and cost. Based on the scenarios tested and data obtained in each deployment model, it can be inferred that deployment via compressed folder (ZIP) presents the main advantages, which are: lower deployment cost, and shorter init time during cold start state.

As future work, the comparison can be extended to other programming languages supported by the AWS Lambda service, as well as comparing the deployment models in different cloud providers, such as Google Cloud and Microsoft Azure. Another aspect to be evaluated is the architecture in which the serverless function is executed, which can be *x86_64* or *arm64*. In this work, only the *arm64* architecture was evaluated, with room to explore the *x86_64* architecture. The scope of the application can also be extended to larger or more complex applications that demand greater computational resources, and which presumably impact the memory consumption and init time. Larger sizes can also impact the final cost, and can be compared with applications larger than 10 MB in ZIP format, which have to be stored on the AWS S3 service.

ACKNOWLEDGMENTS

This work received financial support from the Coordination for the Improvement of Higher Education Personnel - CAPES - Brazil (PROAP/AUXPE).

REFERENCES

- AWS (2024a). AWS Free Tier. <https://aws.amazon.com/free/?all-free-tier>. Accessed: 2024-11-08.
- AWS (2024b). AWS Pricing Calculator. <https://calculator.aws>. Accessed: 2024-11-08.
- AWS (2024c). Lambda instruction set architectures (arm/x86). <https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html>. Accessed: 2024-08-17.
- AWS (2024d). Pushing a Docker image to an Amazon ECR private repository. <https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html>. Accessed: 2024-09-12.
- AWS (2024e). Using Podman with Amazon ECR - Amazon ECR. <https://docs.aws.amazon.com/AmazonECR/latest/userguide/Podman.html>. Accessed: 2024-09-12.
- AWS (2024f). What is Amazon Elastic Container Registry? <https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html>. Accessed: 2024-08-17.
- AWS (2024g). What is AWS Lambda? <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. Accessed: 2024-08-17.
- Bagai, R. (2024). Comparative Analysis of AWS Model Deployment Services. *International Journal of Computer Trends and Technology*, 75(5):102–110.
- Dantas, J., Khazaei, H., and Litoiu, M. (2022). Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 1–10.
- Elsakhawy, M. and Bauer, M. (2021). Performance analysis of serverless execution environments. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–6.
- Kumari, A., Sahoo, B., and Behera, R. K. (2022). Mitigating cold-start delay using warm-start containers in serverless platform. In *2022 IEEE 19th India Council International Conference (INDICON)*, pages 1–6, Barcelona, Spain.
- Maharjan, C. (2022). Evaluating Serverless Computing. Master's thesis, Louisiana State University, Baton Rouge, LA.
- Manner, J., Endreß, M., Heckel, T., and Wirtz, G. (2018). Cold Start Influencing Factors in Function as a Service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188.
- Nupponen, J. and Taibi, D. (2020). Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50.
- Rodríguez, N., Murazzo, M., Martín, A., and Rodríguez, M. (2024). Evaluation of programming languages for memory usage, scalability, and cold start, on aws lambda serverless platform as a case study. In *Computer Science–CACIC 2023: 29th Argentine Congress of Computer Science, Lujan, Argentina, October 9-12, 2023, Revised Selected Papers*, volume 2123, page 33. Springer Nature.
- Scheepers, M. J. (2014). Virtualization and containerization of application infrastructure: A comparison. In *21st twente student conference on IT*, volume 21, pages 1–7.
- Siddiqui, T., Siddiqui, S. A., and Khan, N. A. (2019). Comprehensive analysis of container technology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 218–223.
- Vahidinia, P., Farahani, B., and Aliee, F. S. (2020). Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7.
- Vahidinia, P., Farahani, B., and Aliee, F. S. (2023). Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet of Things Journal*, 10(5):3917–3927.
- Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano Merino, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., and Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, 11:233–247.