Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda

Jaime Dantas, Hamzeh Khazaei and Marin Litoiu
jaimecjd,hkh,mlitoiu@yorku.ca
Department of Electrical Engineering & Computer Science
York University
Toronto, Ontario, Canada

***Abstract*—Serverless computing has emerged in recent years as the new computing paradigm adopted by key players in the industry for software development. This new paradigm has seen rapid growth in adoption due to its unique billing model and scaling characteristics. Public cloud providers such as Amazon Web Services (AWS) offer several configurations and language runtimes for their serverless functions. Although extensively explored by the research community, this field still lacks current studies that address the many challenges developers face when leveraging serverless functions for real-world applications. One of these challenges that are often overseen by many programmers is the cold start problem which is present in any serverless application. For this reason, we propose the first study to characterize the underlying cold start impacts caused by the choice of language runtime, application size, memory size and deployment type on AWS Lambda. In this paper, we analyze the performance of the *container-based deployment* and *ZIP-based deployment* of AWS Lambda using a variety of language runtimes and applications running with different function configurations; then we propose guidelines for developers and cloud managers to consider when deploying/managing the workloads on the cloud.**

***Index Terms*—AWS Lambda, Cold Start, Function as a Service, Serverless Computing, Performance Benchmark**

I. Introduction

Cloud computing has become the new standard for running applications and workloads in the industry. Most of the companies nowadays host their services on private or public clouds, and public cloud providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure (Azure) offer support to most of the services required by complex systems and applications. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Service as a Service (SaaS) are the most popular cloud computing service models used in the industry [1].

Serverless computing, often referred to as Function as a Service (FaaS), is the most recent cloud computing paradigm that originated from the SaaS model. With this new computing paradigm, developers deploy small pieces of code named functions and the cloud provider is in charge of provisioning all the infrastructure and resources required to run these functions. Unlike IaaS or PaaS, where cloud admins still need to provision their cluster, FaaS does not require resource provisioning or complex configurations. The cloud provider is responsible for managing the FaaS infrastructure, allowing customers to deploy and execute their functions in an efficient and faster way compared to the older computing models. With FaaS, cloud providers charge per request, not per resource used. The value of each execution varies according to the function configuration and the region it is executed.

Most of the public cloud providers have their own FaaS platforms and services. Cloud Functions are functions offered by GCP where customers can deploy and run their code with zero server management [2]. Azure also provides their FaaS service named Azure Functions [3], IBM Cloud Functions are the serverless functions offered by IBM [4] and AWS has the AWS Lambda functions [5]. AWS is currently the only public cloud provider to offer different deployment types for their FaaS services. Previously, all major FaaS platforms, including AWS Lambda, supported only one deployment type. Customers were required to upload their function source code into the cloud platform either manually or by compacting the code and deploying it automatically. However, in December 2020, AWS introduced the new deployment type named *container-based deployment* for AWS Lambda [6]. Now, customers can build their own container using the container images provided by the company. The size limitation that was once present on all serverless platforms is now eliminated on AWS Lambda since they allow the container to be as large as 10GB.

One of the most important challenges that all FaaS platforms share is the cold start delay. When a function is deployed in the cloud, a set of sequential processes is executed when this function is first invoked. First, the function source code is fetched from storage, deployed in a container, and then initialized. Once the first request is handled, the following requests are executed faster for a certain period. Then, since the container is already initialized, the function enters into idle mode, and all following requests are executed instantaneously. However, cloud providers can decommission the function container at any time, and whenever it does, clients will experience a cold start again for their requests. This issue is often addressed by researchers as ”the cold start problem”, and many works [8]–[10] try to mitigate this problem faced by FaaS. Still, most of these approaches used to reduce the cold start time of serverless functions often require additional software and architectures which are often seen as complex implementations by developers since it requires extra computational resources and maintenance. Additionally, the lack of knowledge about the benefits and drawbacks of each deployment type of AWS

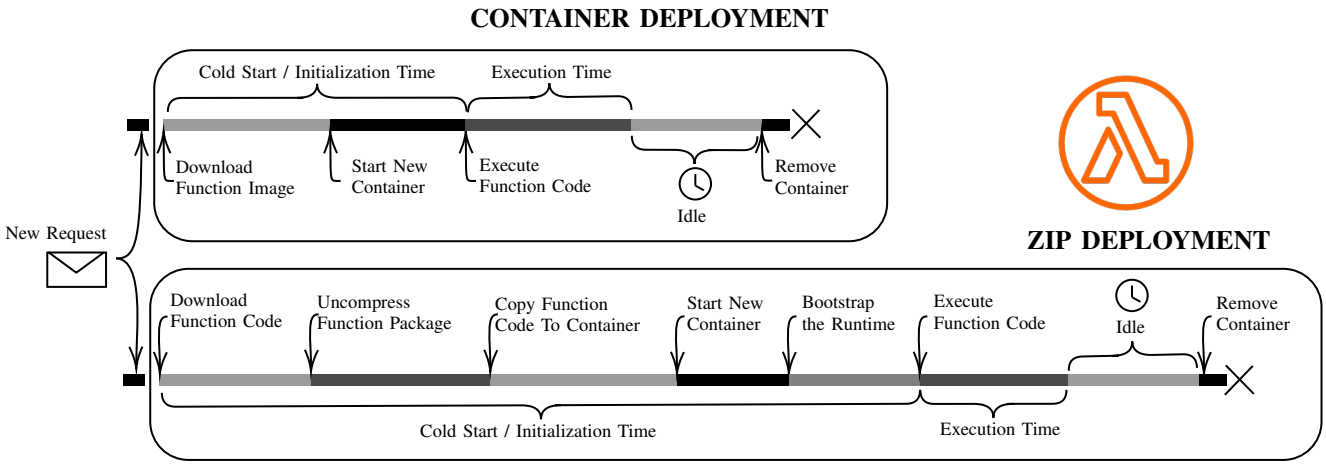

Fig. 1. AWS Lambda lifecycle (adapted from [7]).

Lambda makes this matter even worse.

Before the release of the new *container-based deployment* on AWS Lambda, developers had to compress their source code - including the external libraries and dependencies - in a ZIP file which had a size limit of 250MB (uncompressed). This file is usually referred to as the "Function package file", and when the Lambda is executed for the first time, it imports and uncompresses this function package which is stored in an S3 Bucket, then loads the function for execution in a container. This process is time-consuming, especially for large package sizes. The new *container-based deployment* eliminates the unzip and container building process, which in turn, could lower the initialization time. For this deployment type, a Docker image is built locally and then sent to be stored on the Amazon Elastic Container Registry (ECR) for use with AWS Lambda. AWS providers base container images for all supported language runtimes and system architectures. Although this new *container-based deployment* shows many advantages over its traditional counterpart, little research has been conducted using this new approach.

In this paper, we aim to fill in this knowledge gap by showing how to use application and platform knowledge to reduce the initialization time on AWS Lambda. We analyze the new *container-based deployment* and compared it against the traditional package deployment named *ZIP deployment* of AWS Lambda. We test the performance of several real-world applications under different language runtimes and architectures, and derive guidelines that developers and cloud admins can use to mitigate the cold start problem on AWS Lambda. We aim to answer three research questions:

- **RQ-1:** What is the impact of the AWS Lambda package size on the initialization time when using the *container-based deployment*?
- **RQ-2:** How does the memory allocated to the AWS Lambda function affect the initialization time when using the *ZIP deployment* compared to the *container-based deployment*?
- **RQ-3:** Does the machine learning model size have the same impact on both deployments of AWS Lambda functions?

The contributions of our research are:

- Presenting the first extensive analysis of AWS Lambda that takes into account the *ZIP* and *container-based deployment*, the language runtime, the memory and the package size of the function.
- Suggesting guidelines for reducing the cold start delay on AWS Lambda by choosing the ideal deployment type based on application knowledge.

This paper is structured as followed: We start by explaining the approach we used for analyzing the performance of the *container-based deployment* and the applications tested (section II), then we present the results for the execution time and cold start performance of the two deployment types under different configurations (section III). The package size impact on the initialization time (section III-B2), the memory size (section III-B3), the model size (section III-B4), and the language runtime choice (section III-B5) are discussed, and guidelines are suggested. Finally, we discuss the related work (section IV) before concluding this paper (section V).

II. Methodology

In this section, we go over the details of the methodology proposed in this paper. We answer **RQ-1** and **RQ-2** by analyzing the cold start and execution time of 13 serverless functions deployed with the *ZIP deployment* and *container-based deployment* and different memory sizes. Finally, we answer **RQ-3** by studying the response time of a machine learning function deployed with 5 distinct model sizes using the two deployment types and a wide range of memory configuration.

A. Language Runtimes and Libraries Used

The choice of libraries and language runtimes to use in our analysis was based on the most used libraries and language runtimes on AWS Lambda in the industry. We have selected a wide range of Python libraries that are used in many studies

with serverless functions [11], [12]. In particular, we focus on image processing and machine learning applications using the *TensorFlow*, *Pillow* and *Sklearn* libraries which are also analyzed on [13]. The most used language runtimes on AWS Lambda are Python, Node.js and Java, respectively [14], [15]. However, Node.js applications particularly have never been tested with the new *container-based deployment* on AWS Lambda.

B. Function Deployment Configuration

Fig. I shows the two deployment types of AWS Lambda in detail. The lifecycle of the *container-based deployment* of AWS Lambda is not documented by the company. However, we assumed that the AWS Lambda Manager starts the container as soon as it is fetched from storage in the ECR. This process could, in theory, speed up the start-up time for some language runtimes since it does not need to build the container image. We consider the cold start time - also referred as the initialization time - the period from the invocation of the function to the time the code is ready for execution. Although the objectives of *RQ-1* and *RQ-2* over the *ZIP deployment* have already been answered in previous studies [16], [17], the effects on the *container-based deployment* of AWS Lambda remain unknown to this date. Finally, the execution time is the time that the function takes to execute when its container is up and running.

C. Function Response Time Measurement

We invoke each Lambda using the AWS API Gateway through a RESTful API. In particular, we follow the same benchmark configuration adopted on [18] where AWS API Gateway is used for creating endpoints for invoking the Lambdas. AWS CloudWatch is used for storing the execution log of both the Lambda and the API Gateway call. We use the Locust[1] benchmark tool for our performance tests. Locust is an open-source, scriptable, and scalable performance testing tool that allows customized use test cases written in Python. Fig. 2 shows the architecture view used for each Lambda function. We use the fields *@billedDuration* and *@initDuration* from the original Lambda log file to calculate the execution time and initialization time of each execution, respectively. For the execution time analysis, we check the *@logStream* to extract the container identifier to validate the time by which the container is decommissioned and a new one is launched.

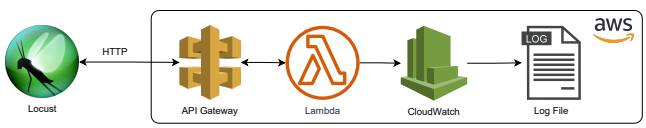

Fig. 2. Architecture built on AWS.

[1]https://locust.io

D. Function Workloads Used

We evaluate the performance of the two Lambda deployments with 13 serverless functions and 3 different language runtimes. These functions are coded in Python, Node.js and Java and tested on both arm64 and x86 architectures. A wide range of libraries and package sizes are used on each of these applications. All the 13 Lambdas are invoked by the Amazon API Gateway. Table I compiles all applications used.

Image Classifier 230: Consists of a binary image classification function using one of the most popular and comprehensive open-source machine learning libraries, the *scikit-learn (sklearn)* [19]. It receives an image of any size, and it predicts which of the two classes this picture belongs to. In order to do the predictions, we first reduce the image to 59×59 pixels using the *Pillow* library, then we extract the Histogram of Oriented Gradients (HOG) and run a Support Vector Machines (SVMs) model. We train 5 SVMs models with different sizes (1MB, 7MB, 12MB, 16MB and 20MB) to evaluate the impact of the model size in the cold start of the application. These models are deployed on the Image Classifiers 230 to 249 from smallest to largest, respectively.

Linear Regression: This application uses the open-source scientific computing library for Python *SciPy*. Since its initial release in 2001, *SciPy* has become the most used library for scientific algorithms in Python [20]. We compute a simple and yet commonly used mathematical operation with this library, the least-squares regression for two sets of measurements.

Image Black and White: We convert an image to black and while using the OpenCV library. This library is widely used among developers for computer vision applications.

TF Image Classifier: It is a popular open-source serverless application that uses the *TensorFlow Lite* library. It consists of a multi-class image classifier using the on-device inference framework from TensorFlow and both the *ZIP* and *container-based deployments* are available on GitHub[2].

Resize and Feature: This function is part of the Image Classifier 230, and it performs the resize and feature extraction of an image of any size using the libraries *Pillow* and *Numpy*.

Resize: This application is a simple application used to resize an image, and it is also part of the Image Classifier 230. It uses only the *Pillow* library.

Factorial: Performs the factorial of a given number. It does not use any external libraries. There are three versions of this application, one for each language runtime.

Large Size Node.js: This application uses some of the most popular Node.js packages from the official NPM Registry website[3]. It performs image processing as well as natural language operations.

Medium Size Node.js: This function performs natural language operations using the well-known package *natural*.

Large Size Java: It is a popular open-source application that performs image processing and mathematical operations

[2]https://github.com/edeltech/tensorflow-lite-on-aws-Lambda
[3]https://www.npmjs.com

TABLE I
SERVERLESS APPLICATIONS.

App Name	ZIP Size (MB)	Image Size (MB)	Architecture	Runtime	Libraries
Python					
Image Classifier 230	230	480	arm64	Python 3.8	Pillow, Numpy, Pillow, Numpy Sklearn, Joblib, Scikit Image
Linear Regression	186	283	arm64	Python 3.8	Scipy, Numpy
Image Black and White	126	256	arm64	Python 3.8	OpenCV, Numpy, Pillow
TF Image Classifier	83	340	x86	Python 3.7	TensorFlow, Numpy, Pillow
Resize and Feature	64	210	arm64	Python 3.8	Pillow, Numpy
Resize	14	182	arm64	Python 3.8	Pillow
Factorial Python	0.004	176	arm64	Python 3.8	
Node.js					
Large Size Node	234	323	x86	Node 14	gulp-imagemin, jspdf, html-pdf, natural, text-extractor, jimp
Medium Size Node	77	204	x86	Node 14	natural, sharp
Factorial Node	0.006	148	x86	Node 14	
Java					
Large Size Java	133	227	x86	Java 11	OpenIMAJ
Medium Size Java	15	188	x86	Java 11	iTextPDF
Factorial Java	10	185	x86	Java 11	

using the award-winning library *OpenIMAJ*. This application is openly available on GitHub[4].

Medium Size Java: This function converts a text input to a PDF document using the popular library *ItextPDF*.

We classify all 13 applications into different application size groups according to the language runtime aiming to answer RQ-1. The Image Classifier 230 is used to answer RQ-2 and RQ-3, and finally, the three factorial applications are analyzed to visualize the impact of the language runtime on the two deployment types.

III. EXPERIMENTAL EVALUATIONS

We performed a combination of tests on both the AWS Lambda *ZIP* and *container-based deployments* and analyzed the execution time and initialization time of each configuration. The experiments were conducted over an extended period of time from December 28th, 2021, to January 18th, 2022. All tests were performed in the AWS region *us-east-1* where each Lambda handles one request per time. The benchmark tests were done using both the arm64 and x86 architectures. We compared the two deployments using several metrics that include the median, the 95th percentile and the average of the response time and the initialization time, as well as the cumulative distribution function (CDF) of all experiments. We first discuss the experimental setup of each test, and then present the results and discussion of the execution time followed by all three research questions and the impact of the language runtime. All the experiments are openly accessible on GitHub[5].

A. Experimental Setup and Data Collection

We first measured the execution time of all 13 applications for both Lambda deployments. Each application was invoked 400 times with requests sent every 1 second. This interval guarantees no queue is formed since the worst execution time is less than 1 second. During this period of approximately 7 minutes, none of the containers were decommissioned. All the logs from AWS CloudWatch, including the Lambda and API Gateway log files, were analyzed and validated. We studied the execution time for 128MB, 256MB, 320MB, 384MB, 448MB and 512MB memory configurations.

Fig. 3. Execution time for the Image Classifier 230.

Then, we evaluated the cold start performance of all applications with different memory configurations. We invoked the Lambdas 20 times with requests sent every 10 minutes. In total, each test lasted 200 minutes, and it was conducted with 128MB, 256MB, 320MB, 384MB, 448MB and 512MB memory configurations. According to our benchmark tests, the idle time of AWS Lambda was less than 10 minutes for all programming languages. We use 10 minutes interval between invocations to compute the cold start time of AWS Lambda. This interval guarantees that each call is executed by a new container.

The second set of tests for the cold start were aiming to study the memory impact on the cold start time. We tested the Image Classifier 230 under 128MB, 256MB, 320MB, 384MB, 448MB, 512MB, 640MB, 704MB, 768MB, 832MB, 896MB, 960MB, 1024MB, 1088MB, 1152MB, 1216MB, 1280MB, 1344MB, 1408MB, 1472MB, 2048MB and 3008MB memory

[4]https://github.com/eugenp/tutorials/tree/master/image-processing
[5]https://github.com/pacslab/serverless-iot-deployment

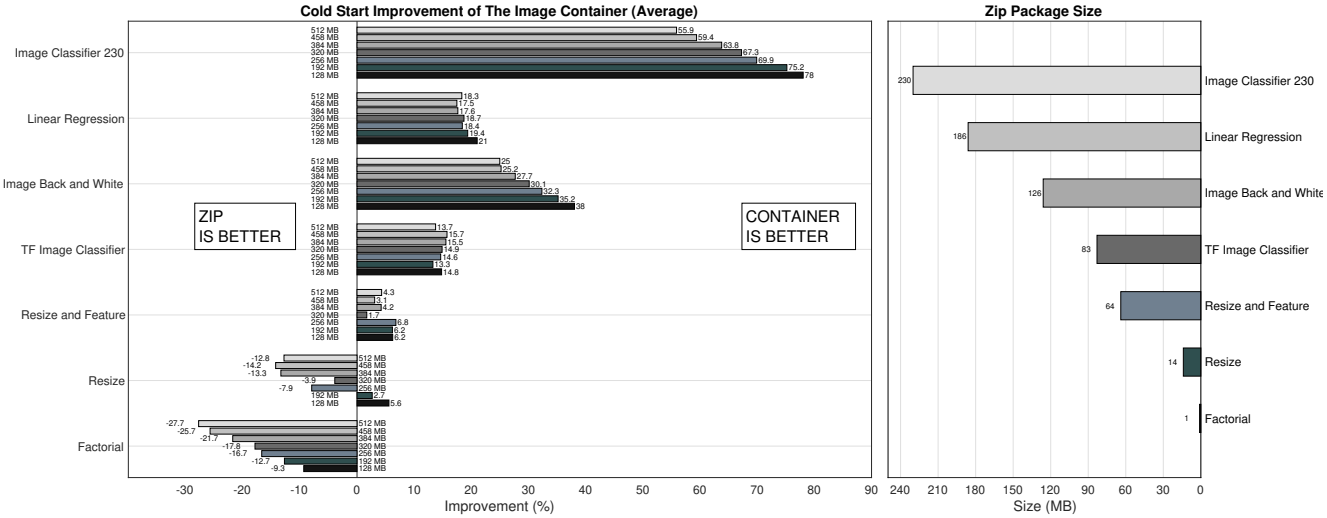

Fig. 4. Initialization time improvement of the *container-based deployment* over the *ZIP deployment* (left), and the ZIP package size (right) of all serverless applications.

configurations. Finally, we used 5 SVMs models from different sizes to study the relationship between the model size and cold start time on both deployments. In order to do this, we used different amounts of data to train these 5 models.

B. Results and Interpretations

We tested the execution time and cold start time of all 13 functions for both Lambda deployments, and present the results for each research question below.

1) Execution Time: In order to answer **RQ-1, RQ-2** and **RQ-3**, we need to first validate whether or not the execution time differ according to the deployment type. Therefore, we analyzed the execution time of all 13 applications. Fig. 3 shows the execution time distribution for two memory configurations of the Image Classifier 230 application. As we can see, both execution times are statistically equivalent. The same was seen for the Node.js and Java applications. Furthermore, the execution time of the AWS Lambda *ZIP* and *container-based deployments* are equal for all memory sizes and language runtimes. Although we have noticed a slight difference in the execution time in favour of the *container-based deployment*, both measurements are statically equivalent.

2) RQ-1: What is the impact of the AWS Lambda package size on the initialization time when using the container-based deployment?: On **RQ-1** we want to study the impact of the package size of the Lambda function on the *ZIP* and *container-based deployments*. We first studied this on Python functions, and then moved to the other programming languages. The right chart of Fig. 4 shows the different sizes of the *ZIP deployment*. We measured the initialization time of both deployments, and then calculated the percentage of improvement of the *container-based deployment* over the *ZIP deployment*. This data is presented on the right chart of the Fig. 4.

We observe two trends with regards to the package size of **Python** applications. The *container-based deployment* shows a better performance for large package sizes and small memory configurations. For instance, the initialization time of the Image Classifier 230 was 78% smaller than the *ZIP deployment* for 128MB memory. As we increase the memory size, this difference decreases and the two deployment types become similar in cold start time. For most of our tests, smaller memory configurations favoured the *container-based deployment* since this Lambda deployment type presented a smaller cold start time.

Our results contradict the academic work presented on [18] where the *container-based deployment* of Python applications showed worse initialization time than the *ZIP deployment*. Unlike our experiments, the authors on [18] used only one Python application - with a small package size - and tested it with one memory configuration only. Even though they only evaluated one specific scenario, they claimed that the *container-based deployment* was worse than the *ZIP deployment* for interpreted programming languages such as Python when it comes to cold start time. In contrast, our results showed that the size of the Lambda package - the ZIP archive or Docker container image - is, in fact, a key factor in the initialization time of these two deployments.

Another observation is that the package size impacts the cold start time of both deployments. Small package sizes had better initialization time with the *ZIP deployment*. For the Python Factorial function, which has only 4kB in size, the *container-based deployment* was worse than the *ZIP*'s for all memory configurations with figures varying from 9.3% to 27.7% worse initialization times for 128MB and 512MB, respectively. These findings are in line with the results presented on [18], and it may be due to the fact that Python functions are non-static binary programs, and consequently, all libraries and modules have to be imported dynamically which adds overhead in the initialization time of the *container-based deployment*.

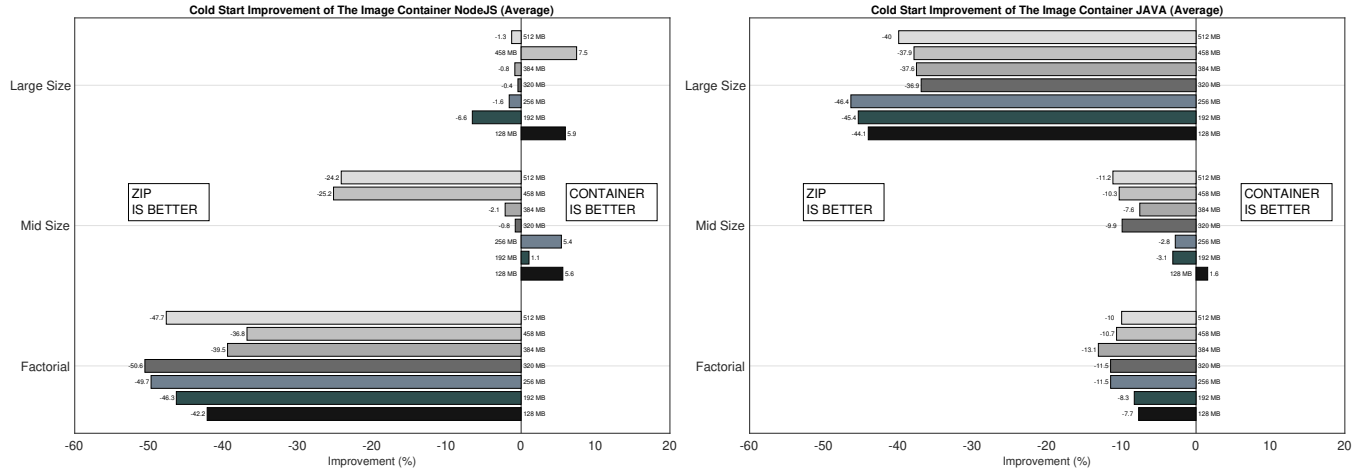

Fig. 5. Initialization time improvement of the *container-based deployment* over the *ZIP deployment* of the Node.js (left) and Java (right) functions.

One of the most important findings is the inflection point by which the *ZIP deployment* becomes better than the *container-based deployment*. This can be seen when we analyze the initialization time for the Resize application which has 14MB in size, and it uses a popular Python library for image manipulation - *Pillow*. From this data, we can see that the point of inflection is located between 192MB to 256MB where the *container-based deployment* becomes worse than the *ZIP*'s by 7.9%. This is because while for 192MB the former deployment was 2.7% better than the latter one, for 256MB the container-based was 7.9% worse than the *ZIP deployment*.

Therefore, we advocate that developers and cloud managers should choose the *container-based deployment* whenever they are using a large Python application with external libraries and dependencies. In our tests, Python applications with sizes equal to or larger than 64MB are better with the *container-based deployment* for memory configurations of up to 512MB. If, however, developers are deploying smaller Python applications of 14MB in size, for instance, further testing is necessary to find out the best deployment according to their memory needs. In our tests, Python applications with 14MB in size and up to 192MB memory are better with the *container-based deployment*. After this point, the *ZIP deployment* becomes a more suitable option. Finally, small Python applications of 1MB or less in size that have no libraries or external modules should be deployed using the *ZIP deployment* since it offers a better initialization time.

Even though both Python and Node.js are dynamically-typed languages, the performance of these languages under the *container-based deployment* is different. The left chart of Fig. 5 shows the percentage of improvement of the *container-based deployment* for Node.js functions. For large **Node.js** applications, the *container-based deployment* showed a similar cold start compared to the *ZIP deployment*. Both the average and 95th percentile were statically equivalent for most memory sizes. For instance, for 320MB memory, the average differs in only 0.43% in favour of the *ZIP deployment*. Although we observed a tiny improvement in the average of the *container-based deployment* for small memory sizes of 5.9 %, the 95th percentile of this deployment is in fact 10% worse than the *ZIP deployment*. Therefore, the choice of deployment has little to no impact on the initialization time of large Node.js applications.

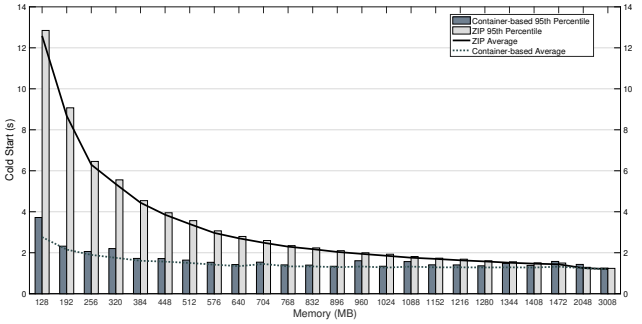

Fig. 6. Initialization time of the Image Classifier 230 under different memory configurations.

When it comes to small packages sizes, the same outcome found for Python applications also applies to Node.js functions. This is because the Node.js Factorial function was also faster with the *ZIP deployment*. Similar to the Python applications, as we increase the application package size, this difference becomes smaller and the two deployments perform similarly. This can be seen when we analyze the initialization time for medium size Node.js applications. However, unlike Python applications, Node.js applications performed better or equal with the *ZIP deployment* on all package sizes. Therefore, we recommend that cloud admins use this type of deployment for Node.js applications of any package size up to the 250MB limit.

Unlike the results seen with the dynamically-typed languages Python and Node.js, the performance of **Java** functions, which is a is a statically-typed language, is quite distinct. The right chart of Fig. 5 shows the percentage of improvement of the *container-based deployment* for Java applications.

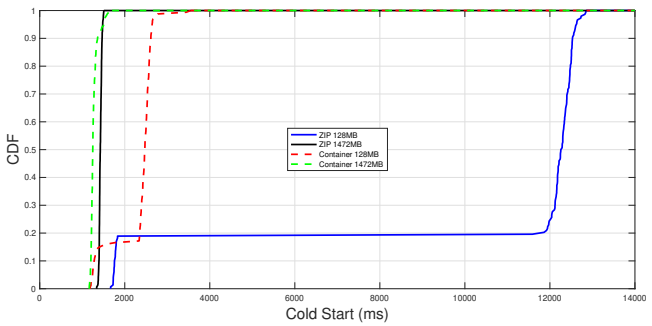

Fig. 7. CDF of Image Classifier 230 initialization time.

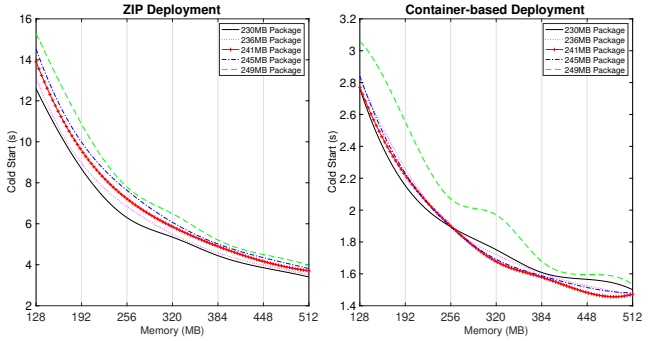

Fig. 8. Model size impact in the initialization time of the *ZIP deployment* (left) and *container-based deployment* (right).

Overall, the *ZIP deployment* outperformed the *container-based deployment* on all memory and package sizes. In fact, we have noticed the opposite trend where larger applications performed significantly better with the *ZIP deployment*. For instance, the average cold start time of the *container-based deployment* was 46.4% worse than the *ZIP*'s for the large size Java function with 256MB memory configuration. As we decrease the package size, this difference also decreases but for almost all configurations the *ZIP deployment* is preferable.

Our results contradict the findings on [18] where a Golang function - also a statically-typed language - is analyzed. The authors on [18] advocate that statically-typed languages such as Golang and Java have similar initialization times on both deployments. However, they only tested one package size and did not test Java applications. From our findings, we can also conclude that the *ZIP deployment* should be used for Java applications of any package size up to the 250MB limit.

Recommendations of RQ-1

- Python applications with large package sizes have faster initialization time with the *container-based deployment* while small applications are better with the *ZIP deployment*. Our experiments show that the inflection point is 64MB in size.
- Node.js applications with small package sizes have faster initialization time with the *ZIP deployment* while medium and large size ones have approximately equivalent cold start time to some extent.
- Java applications of any package sizes have faster initialization time with the *ZIP deployment*.

3) RQ-2: How does the memory allocated to the AWS Lambda function affect the initialization time when using the ZIP deployment compared to the container-based deployment?: Moving on to **RQ-2**, we wanted to see if the memory size had the same impact in the cold start time of the two different deployments for all three language runtimes. First, we analyzed only **Python** applications. Fig. 6 shows the Image Classifier 230 under a wide range of memory configurations. As expected, the *container-based deployment* presented a significantly smaller initialization time for memories from 128MB to 512MB. As we increase the memory size, this difference gets smaller and the two deployments become equivalent eventually.

This behaviour is seen when we analyze the average and 95th percentile initialization time for the 1472MB memory configuration. Even though the average initialization time of the *container-based deployment* is 8% better, the 95th percentile is 5% worse. Thus, we assumed this is the point where the two deployments reach the same minimum value. In order to further investigate the initialization time of the two deployments for very small and very large memories, we plotted the CDF of 200 executions for 128MB and 1472MB memory configurations on Fig. 7.

From the CDF on Fig. 7, we can see that the contained-based deployment with 128MB memory configuration had a cold start average $4.5\times$ smaller than the *ZIP deployment*. However, they have near-identical cold start CDFs for 1472MB memory configuration. Thus, we conclude these two deployments have the same initialization time for memory configurations larger or equal to 1472MB. Therefore, developers should consider the size of the application and the memory allocated to the Lambda to decide on what deployment type to choose. The analysis conducted on [16] and [17] also suggests that using large memory sizes can reduce the cold start effect in AWS Lambda when using the *ZIP deployment*. Thus, our results validate this behaviour for the *container-based deployment* as well.

However, function memory has little to almost no impact on the initialization time of both **Node.js** and **Java** applications. This is especially true for small size applications such as the Factorial one. In both these languages, the memory played no role in the performance of the cold start. The same was also seen for large-size applications where the performance of all memory configurations was similar. However, for medium-size applications, the memory size does affect the performance of the cold start. This is because both the Node.js and Java applications showed a slightly better performance in favour of the *container-based deployment* for small memory configuration. As we increase the memory size of the medium size Node.js function, for example, this difference becomes

smaller and after 256MB the *ZIP deployment* becomes better. This behaviour was the same observed for medium Python applications.

As a result, the impact of the Lambda memory on both Node.js and Java functions depends on the package size. We recommend that developers and cloud admins should choose the *container-based deployment* only if the application package size is between 15-77MB for Node.js applications under small memory configurations. For both small and large functions, Node.js and Java functions should use the *ZIP deployment* instead. Finally, we tested both the arm64 and x86 architectures on all applications, and they are equally in performance. Thus, the architecture does not influence the cold start time of Lambda applications.

Recommendations of RQ-2

- Small memory sizes, e.g., 128MB, make the initialization time of the *container-based deployment* faster when deploying medium and large size Node.js and Python applications compared to the *ZIP deployment*.
- Very large memory sizes, e.g., 1472MB for Python applications, make the initialization time of both deployment types approximately equivalent.

4) RQ-3: Does the machine learning model size have the same impact on both deployments of AWS Lambda functions?: Finally, **RQ-3** investigates the effect of the model size in the cold start of the functions. We tested the 5 SVMs models and included them inside the original application Image Classifier 230. The 5 applications were named Image Classifier 230MB, 236MB, 241MB, 245MB and 249MB from smallest to largest, respectively. Fig. 8 shows the effect the model size has on the two deployments studied. As we expected, the larger the model size, the higher the cold start time for the *ZIP deployment*. Also, the curve concave decreases as we increase the memory size for all model sizes under the *ZIP deployment*. The *container-based deployment*, however, was slightly less susceptible to the model size changes. While the variation of the model size caused a 22% change in the cold start time of the *ZIP deployment*, this figure was only 11% for the *container-based deployment* under 128MB memory configuration.

Therefore, changes in the model size of machine learning applications are less likely to cause variations in the cold start time of *container-based deployment* compared with the *ZIP*'s. This is especially true when we analyze the cold start of the Image Classifiers 241 and 245. The two curves for these applications are approximately equivalent for the *container-based deployment*. For example, for 256MB memory, these two applications had only 0.33% difference in the initialization time of the *container-based deployment* whereas for the *ZIP* version this figure was 5.39%, which is roughly 17 times more than the former deployment.

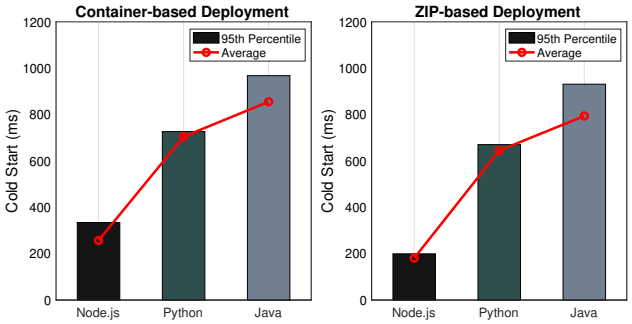

Fig. 9. Factorial function under different language runtimes and 128MB memory configuration.

Recommendations of RQ-3

- Variations in the model size are less likely to cause impact in the initialization time of the *container-based deployment* compared to the *ZIP deployment*.

5) Language Runtime Impact: Previous works have shown that the choice of the language runtime affects the cold start time of AWS Lambda [21], [22]. We extend **RQ-1** and **RQ-2** by analyzing the impacts of the language runtime in the cold start of serverless applications. We analyzed the average and 95th percentile initialization time of the Factorial function in Python, Node.js and Java. Fig. 9 compiles the cold start time of these applications on both the *ZIP* and *container-based deployments* for 128MB memory configurations.

Our findings show a noticeable impact of the language runtime in the cold start of applications. As can be seen, both deployments had the same trend where Node.js functions were the fastest followed by Python and Java, respectively. Java applications have the worse initialization time among all three languages, with figures approximately 4 times higher than the Node.js ones for the *ZIP deployment*. For the *container-based deployment*, this figure is around 3 times which is still representative. Our results follow the work present on [21], [22] where is shown that Java functions have a high trail latency compared to Python applications. However, our work showed that Node.js applications have the best performance which contradicts the works [21], [22] since they claim that Node.js functions are in fact worse than Python applications. One of the reasons may be due to the fact that these academic benchmark tests were performed using an older version of all language runtimes analyzed.

Our result is surprising since it also contradicts the work on [18] where their authors state that the language runtime has negligible implications for cold-start delays for the *ZIP deployment*. As seen on the right chart of Fig. 9, the impact of the programming language is significant for the 128MB memory configuration. One of the possible reasons that can explain this opposite outcome is the memory configuration allocated to the Lambda. While we tested a wide range of memory sizes - from 128MB to 3GB - the authors on [18] only

tested it with 2GB which may bias their findings. However, the analysis performed on [16] and [17] also suggests that using dynamically-typed languages such as Node.js and Python can reduce the cold start effect in AWS Lambda when using the *ZIP deployment*. Our study shows that this impact is also seen with the *container-based deployment* in a corresponding degree.

In conclusion, the *container-based deployment* had a better initialization time for larger Python applications and small memory configurations. Also, it is more suited to deploy machine learning applications with embedded models. Python applications larger or equal to 64MB in size should be deployed as containers whenever they are running with 512MB or less memory. Additionally, very large Python applications of 230MB or more in size are faster when using container-based with up to 1472MB of memory. Finally, both Node.js and Java applications have, in general, faster or equal initialization time when using the *ZIP deployment*, especially for large memory configurations.

IV. Related Work

Prior work includes a number of frameworks and FaaS architectures developed to mitigate the cold start time present in most of the public cloud providers nowadays. WLEC [23] and Pigeon [24] are alternative approaches to AWS Lambda that reduce the initialization time, and can be integrated with other serverless providers. Application knowledge is used on [25] for reducing the duration of cold start by implementing a lightweight choreography middleware for FaaS. However, all these works are complex architectures that require extra computational resources and maintenance in order to be used with Lambda functions. Cloud admins and developers could use our insights instead, and change the Lambda deployment type according to their applications to significantly reduce the initialization time.

Many benchmark tests addressed the impact of the memory size and package size on the cold start of serverless functions. Daniel et al. [26] studied the cold start time of a wide range of applications on AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions. This type of performance analysis was also investigated on [26]–[28], however, since no cloud provider supported container-based function deployment at the time, no prior work was developed using this new development type. Additionally, the open-source benchmark suite for characterizing serverless platforms ServerlessBench developed on [29] evaluates the cold start time of functions with different sizes on AWS Lambda using the *ZIP deployment*. Similar to our results, the authors on [29] advocate that large-size functions suffer from longer initialization time due to larger data transmission and package import overhead. However, only Python applications are studied on [29], and our work is the first to show that these impacts could be mitigated by changing the deployment type for some language runtimes.

The size limit of 250MB of AWS Lambda is challenging for most of the machine learning applications since they use large libraries and models. AMPS-Inf [30] is a framework that solves this problem by partitioning customized libraries and modules across a number of Lambda functions. Although this solution can be used for large Lambda functions, AWS Lambda now supports Docker container images of up 10GB in size, and as demonstrated on our benchmark tests, this deployment is ideal for large workloads. The choice of language runtime is also discussed on [12], [21], [22], however, our study is the first of its kind to address the *container-based deployment* with Node.js and other programming languages. In addition to the size limitation of the *ZIP deployment*, many Internet of Things (IoT) and latency-critical applications often face performance issues when using AWS Lambda due to the cold start problem. The survey on [31] shows that despite the many challenges the cold start delay brings to these applications, they still use serverless functions platforms such as AWS Lambda. Now, developers and companies can use the insights we present in this paper to mitigate the cold start problem these functions face.

The only study found that partially investigates the use of container-based Lambdas is the STeLLAR benchmarking framework [18]. STeLLAR is an open-source serverless benchmarking framework that enables an accurate performance characterization of serverless deployments. It evaluates the cold start time of the *ZIP* and *container-based deployments* of Lambda applications with two types of applications. However, unlike our benchmark tests, the authors on [18] only tested their applications with a 2GB memory configuration. Their findings show that for small Python applications, the *ZIP deployment* is recommended based on the initialization time. Contrary to the aforementioned work, our findings suggest that memory, package size and language runtime are pivotal when choosing the best deployment type for better cold start performance. Additionally, unlike the work presented on [29] and our study where real-world applications were used and all functions imports and uses external libraries and dependencies, a random file is used for simulating the different application sizes on the analysis performed by STeLLAR.

As demonstrated in this paper, the impact of choosing a compiled or interpreted language runtime on the cold start is tremendous. The work on [8] came with one hypnotizes why Java functions have the worst initialization time of all three language runtimes tested. They claim it is because Java applications need more resource-intensive environments for starting the JVM, which overcharges the already busy CPU. This may explain why large Java functions performed badly with the *container-based deployment*. The authors on [8] also stated that this effect is smaller for higher memory settings. Our work is additional to this analysis since we demonstrate that medium-size Java applications when running with small memory configurations may present a lower cold start time when deployed as a container instead of a ZIP package.

V. Conclusion and Future Work

Over the past five years, serverless computing has grown exponentially in popularity, and it is now one of the most

used software architectures in the industry. However, not much focus was given to the underlying problems this model brings to light. The cold start problem is one of the biggest challenges that comes when using serverless functions. Additionally, the many choices around language runtime, memory configuration and deployment types one can have for deploying their workloads on the cloud make this problem even more complex. To address this issue, we presented the first extensive study about the impacts caused by the language runtime, memory allocation and function package size in the initialization time of AWS Lambda when using the two deployment types available, the *ZIP deployment* and the recently introduced *container-based deployment*.

We proposed guidelines for Python, Node.js and Java serverless functions under different memory configurations and application sizes according to which deployment type presents the best cold start performance. Developers can use these insights to achieve lower initialization times when deploying their applications on AWS Lambda by using application and platform knowledge. In the future, we want to launch an open-source benchmark framework to allow the automatic performance characterization of workloads for AWS Lambda.

REFERENCES

[1] I. Cloud. (2021) Iaas vs. paas vs. saas. [Online]. Available: https://www.ibm.com/cloud/learn/iaas-paas-saas

[2] G. Cloud. (2022) Cloud functions. [Online]. Available: https://cloud.google.com/functions

[3] M. Azure. (2022) Azure functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[4] I. Cloud. (2022) Ibm cloud functions. [Online]. Available: https://cloud.ibm.com/functions/

[5] A. W. Services. (2022) Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[6] ——. (2020) Aws lambda - container image support. [Online]. Available: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support

[7] ——. (2018) Become a serverless black belt - optimizing your serverless applications. [Online]. Available: https://pages.awscloud.com/Become-a-Serverless-Black-Belt—Optimizing-Your-Serverless-Applications_0205-SR_OD.html

[8] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[9] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 395–400.

[10] H. Puripunpinyo and M. Samadzadeh, "Effect of optimizing java deployment artifacts on aws lambda," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2017, pp. 438–443.

[11] I. Stancin and A. Jovic, "An overview and comparison of free python libraries for data mining and big data analysis," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 977–982.

[12] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 30–44. [Online]. Available: https://doi.org/10.1145/3419111.3421280

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

[14] R. Ribenzaft. (2019) What aws lambda's performance stats reveal. [Online]. Available: https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/

[15] Datadog. (2021) The state of serverless. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[16] E. Şamdan. (2018) Dealing with cold starts in aws lambda. [Online]. Available: https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532

[17] E. Samdan. (2017) A cloud guru news. [Online]. Available: https://acloudguru.com/blog/engineering/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda

[18] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," in *2021 IEEE International Symposium on Workload Characterization (IISWC'21)*. United States: Institute of Electrical and Electronics Engineers (IEEE), Sep. 2021, 2021 IEEE International Symposium on Workload Characterization, IISWC 2021 ; Conference date: 07-11-2021 Through 09-11-2021. [Online]. Available: http://www.iiswc.org/iiswc2021/index.html

[19] S. Raschka and V. Mirjalili, "Python machine learning: Machine learning and deep learning with python," *Scikit-Learn, and TensorFlow. Second edition ed*, 2017.

[20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[21] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 Usenix Annual Technical Conference USENIX ATC 18)*, 2018, pp. 923–935.

[22] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, *Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting*. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[23] K. Solaiman and M. A. Adnan, "Wlec: A not so cold architecture to mitigate cold start problem in serverless computing," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*, 2020, pp. 144–153.

[24] W. Ling, L. Ma, C. Tian, and Z. Hu, "Pigeon: A dynamic and efficient serverless and faas framework for private cloud," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1416–1421.

[25] D. Bermbach, A.-S. Karakaya, and S. Buchholz, *Using Application Knowledge to Reduce Cold Starts in FaaS Services*. New York, NY, USA: Association for Computing Machinery, 2020, p. 134–143. [Online]. Available: https://doi.org/10.1145/3341105.3373909

[26] D. Kelly, F. G. Glavin, and E. Barrett, "Serverless computing: Behind the scenes of major platforms," 2020.

[27] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 311–327. [Online]. Available: https://doi.org/10.1145/3419111.3421306

[28] S. Horovitz, R. Amos, O. Baruch, T. Cohen, T. Oyar, and A. Deri, *FaaStest - Machine Learning Based Cost and Performance FaaS Optimization: 15th International Conference, GECON 2018, Pisa, Italy, September 18–20, 2018, Proceedings*, 01 2019, pp. 171–186.

[29] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 30–44. [Online]. Available: https://doi.org/10.1145/3419111.3421280

[30] J. Jarachanthan, L. Chen, F. Xu, and B. Li, *AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472501

[31] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.