CLEI electronic journal, Volume 24, Number 2, Paper 6, July 2021

Cloud Function Performance: a component modeling approach

Martín Flores-González
Costa Rica Institute of Technology, Computer Science Department
Cartago, Costa Rica
cflores@tec.ac.cr

and

Ignacio Trejos-Zelaya
Costa Rica Institute of Technology. Computer Science Department,
Cartago, Costa Rica
itrejos@tec.ac.cr

Abstract

Cloud Functions are a trend in cloud computing in which developers are allowed to install code in a *Function-as-a-Service* (FaaS) platform able to manage provisioning, execution, monitoring and automatic scaling. The underlying infrastructure in FaaS platforms is hidden from the developers and designers and, since the influence of the infrastructure is unknown, this makes it difficult to apply software performance engineering approaches on cloud functions, which could lead to wrong or inaccurate performance estimations. In this study, we explore the use of component-based modeling and simulation in order to generate performance estimations of an exemplar cloud function which was exercised using a variety of workloads. A cloud function was both implemented and instrumented to record performance data in a log file, which was associated with its invocations. Providing the log file as an input, we extracted a component-based performance model in a format suitable for running simulations on the *Palladio Component Model*, to validate whether the generated model could explain the runtime behavior of the function. Using this approach and further tunings in the model, we were able to validate that the simulations could explain more than 95% of the function's behavior and that component-based modeling and simulation can be considered a serious option when trying to explain the behavior of a cloud function.

Keywords: Software Performance Engineering, Model-driven Software Engineering, Function-as-a-Service, Cloud Computing, Cloud Functions.

1 Introduction

Function-as-a-Service (*FaaS*) services are a recent trend in cloud computing [1]. They allow the deployment of code, in function form, on a cloud provider platform which is responsible for the execution, resource provisioning, monitoring and automatic scaling of the service at runtime. Charges are made depending on the use of the resources involved (which are usually measured in milliseconds).

In this context, code "in function form" is small in size, has no state, works on demand, and has only one functional responsibility. Since the developer doesn't need to worry about the operational details of code deployment, industry started to describe this as a situation in which it is not necessary to deal with servers for both application development and execution, coining the term *serverless* to refer to this kind of technology. It is worth mentioning that *FaaS* and *serverless* are not the same thing. *FaaS* is one of the use cases of *Serverless* technologies. Currently, there are *serverless* technologies related to message queues, databases, security, networking, and the list keeps growing [2].

Performance estimation of applications running in the cloud, such as the ones running on *FaaS* platforms or in microservices-based architectures, present interesting problems to the software performance engineering

community. Some authors argue that we still lack of performance engineering approaches which consider the peculiarities of microservices [3] and have identified at least six challenges for the performance of FaaS Cloud Architectures [4].

There are several open source platforms for *FaaS* (e.g. Apache OpenWhisk) where one can get in-depth details about their underlying infrastructure. This helps to better understand what to expect performance-wise, but, on the other hand, these platforms have also big and complex infrastructures, making performance estimation a very challenging task: There are a lot of moving parts, actors, and resources involved.

In this article, we explore the application of component-based software modeling [5] for functions running in *FaaS* environments, with the aim of obtainig meaninful performance estimations for systems based on this kind of software architecture. As a case study, we implemented an image resizing function. Image resizing (and other such on-the-fly image modifications) is a recurring problem when working with user interfaces in large Web applications such the ones intended for E-commerce or Content Management Systems. Work of this kind can be performed by an independent function within a larger software system.

We implemented a reference function named *Image Handler* (an on-the-fly image resizer). Then, we generated a series of workloads to be run against the function in order to gather performance data in runtime logs [6] and extract a base performance model from these data. Finally, we made both iterative analyses and simulations [7] over the model in order to evaluate how appropiate it would be for characterizing the function's performance behavior under representative workloads and futher experimentation.

Results acquired through this process were promising. The simulations we ran were able to explain in more than 95% the observed response times of the reference function when running in a production environment. The study revealed that component-based software modeling and simulation is a viable alternative for performance estimation of cloud functions and that this modeling approach opens new possibilties for the testing, refinement and predicting of the function's behavior under heavy workloads - prior to its actual deployment on a cloud provider's platform.

This article is organized as follows: Section 2 provides a *Background* about software performance engineering, software performance engineering in the cloud, microservices and *serverless* computing. Section 3 describes *Related work* in software performance engineering aimed at cloud applications. Section 4 introduces a *Case study: A Cloud-Based Image Handler* on the implementation details of the *Image Handler* function and the process around the performance model extraction. Section 5 presents the *Experimental design* proposed in order to validate and compare the results of the simulations performed on the model against real function invocations. Section 6 analyzes and discusses the *Results* obtained in the study. *Conclusions* are presented in section 7 and *Future work* is addressed in Section 8.

2 Background

2.1 Software Performance Engineering - SPE

According to Woodside et al. [8]: *Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements.*

According to these authors, software performance engineering approaches can be classified in two main categories: *measurement* based and *model* based. The first one is the most common, has had more relevance in industry [9] and is used to validate performance or to locate and repair *hotspots* in a system. In this approach, testing, diagnosis and tuning can only be performed when the software project is close to reach the end of the development process. In contrast, the model based approach focuses on the early stages of the development process: there, models are key for quantitative prediction and, through iterative simulation, they can help to validate how well a software architecture can meet its performance expectations. In practice, these two approaches are not 100% independent nor completely pure: measurements are needed to obtain inputs for most of the model based approaches (e.g. model-based simulators) and, on the other hand, the majority of the measurement based approaches use at least one model.

2.1.1 Model based performance engineering

Software performance modeling seeks to find performance issues and design alternatives early in the development process, to avoid risks and the cost and complexity of requirements changes or redesign (with subsequent reprogramming).

Performance modeling tools contribute to predict a system's behavior prior to building it, or to assess the potential impact of a change before it is implemented. Performance modeling can provide early warnings during the development process, helping to formulate more precise and detailed models throughout. When development is starting, a model cannot be validated against a real system. A model represents a designer's

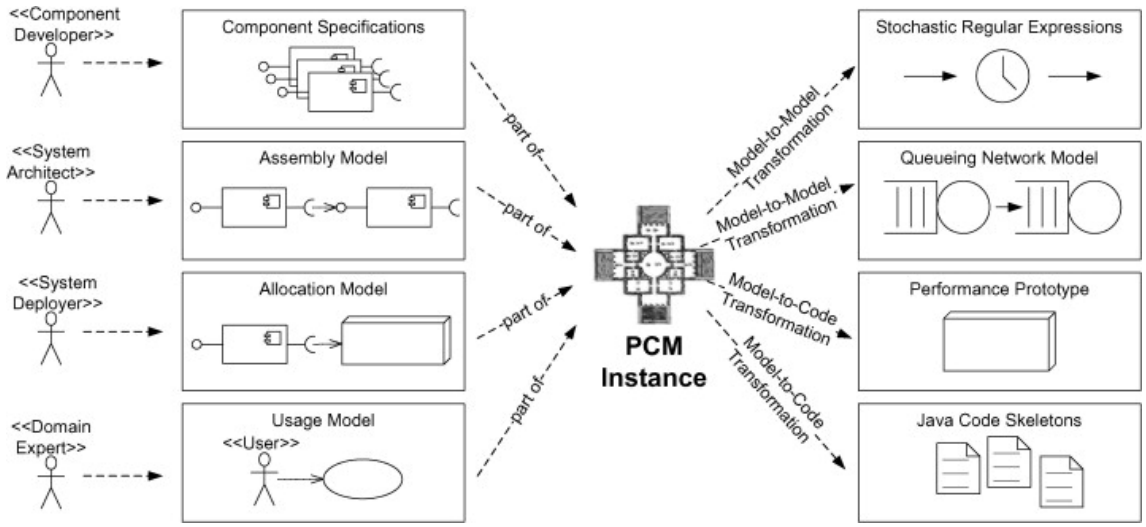

Figure 1: Instance of a PCM model. Source: [10]

uncertain knowledge and includes assumptions that will not necessarily hold in the real system; models are useful to abstract the (future) behavior of the system. A model can be validated against an architectural prototype of the software system, so as to reduce the risk of drawing erroneous conclusions due to its limited accuracy. The model can then be iteratively validated against measurements on refined architectural prototypes, up to the real system, in order to improve the precision of the model.

2.2 Component-Based Performance Modeling and Simulation

Component-based software engineering is considered a successor to object-oriented software development. In component based software development, each software element (component) is defined by a specification and an interface. From these, software architects build and integrate software components, thus creating more complex systems [5].

In component performance models, a challenge is that the performance of a software component within a running system depends on the context in which it is installed and also on its usage profile. This is usually unknown to the component developer as well as to the modelers of the system and its components.

2.2.1 Modeling Sotfware Architectures with the Palladio Component Model

The *Palladio Component Model* (PCM) is a modeling approach for component-based software architectures that enables model-based performance prediction. PCM provides modeling concepts to describe software components, software architecture, component deployment and usage profiles of component-based software systems in different sub-models (Figure 1).

In the Palladio Component Model, the sub-models are:

- **Component specifications**: abstract and parametric descriptions of software components. These describe the internal behavior of the component as well as the demands on its resources expressed in *Resource Demanding Service EFFect specifications* (RDSEFFs) using a notation similar to UML activity diagrams.
- **Assembly model**: specifies the types of components used in a modeled application instance. In addition, it defines how the component instances are connected, to represent the software architecture.
- **Allocation Model**: defines the runtime environment and resources, as well as the deployment of component instances for the resource containers.
- **Usage Model**: specifies the interaction of users with the system, with a syntax similar to an UML activity diagram, to describe the sequence and frequency with which users activate available operations on a system.

A Palladio model abstracts a software system at the architecture level; it can be annotated with previously measured resource consumptions or with estimates. The model can be used in model-to-model or model-to-text transformations for a particular analysis (e.g. queuing networks or code simulation), which can be

solved analytically or by simulation in order to obtain results on performance and predictions regarding the modeled system. These can be used to assess quality attributes of model-based software systems [11] and to improve their design.

2.2.2 The Declarative Performance Engineering Approach

To obtain a model from a running software system, Walter et al. [12] introduced the *Declarative performance engineering* approach, which seeks to automate several activities of the performance engineering process in order to explore, respond to, and visualize aspects of a system's performance. This is achieved thanks to a set of tools developed by the Software Engineering group of the University Würzburg (Germany), available at `http://descartes.tools`.

Figure 2 shows the tools involved in the declarative performance engineering process. The tools can be oriented to: determine service levels, interpret measurements obtained through the execution of a system, extract performance models, or visualize results.

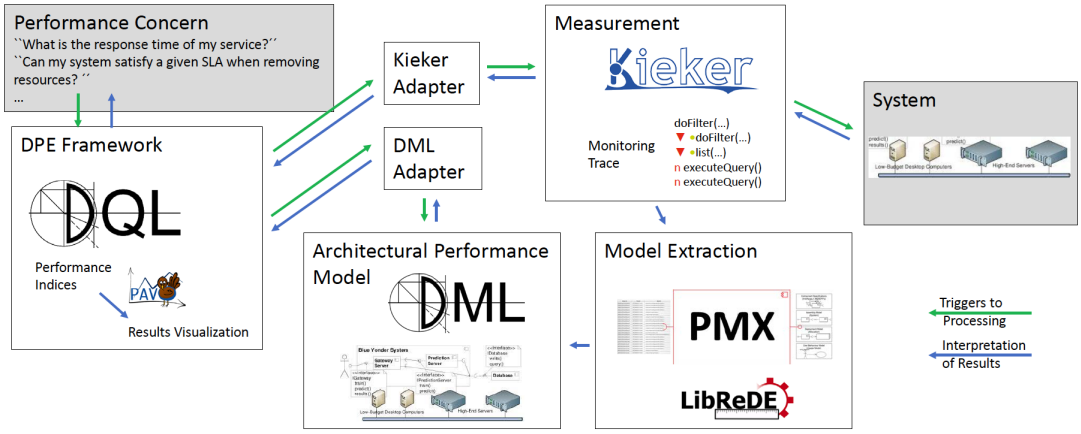

Figure 2: Tools used for Declarative performance engineering. Source: [12]

The Declarative Performance Engineering Approach's tools are:

- **DPE Framework (DPE)**: for determining the service levels of a system. This is achieved through the *Descartes Query Language* (DQL), a language for querying performance indicators. DPE also includes *Performance Analysis Visualization* (PAVO) for displaying performance indicators.
- **Kieker**: to monitor application performance.
- **Performance Model Extractor (PMX)**: to derive performance models from application performance monitoring data. It can take the data obtained by Kieker and generate performance models in the Palladio Component Model (PCM) or the Descartes Modeling Language (DML).

2.3 Function-as-a-Service & Serverless

FaaS is a special use case of what is known today as serverless computing. Amazon.com defines *serverless* as a technology that allows one to build and run applications and services without having to worry about servers. *Serverless* applications do not require provisioning, scaling, or managing any servers. The *serverless* style allows one to build almost any type of application or service; everything that is required to run and scale an application with high availability is managed by the service provider [2].

FaaS refers to a type of *serverless* application where the server-side logic is written by a developer but, unlike traditional architectures, it is executed in compute containers that do not maintain state, which are activated through events, are ephemeral (they last for only one invocation) and are fully managed by a third party's platform [2].

2.3.1 Function-as-a-Service *providers*

According to [13], main providers of FaaS services are AWS Lambda, Google Functions, Microsoft Azure Functions and IBM Apache OpenWhisk Functions.

AWS Lambda (`https://aws.amazon.com/lambda/`) AWS Lambda is offered by Amazon Web Services. AWS Lambda supports several languages, such as NodeJS, Java, C# and Python. Memory allocation ranges from 128MB to 1536MB. Disk capacity can go up to 512MB (space in `/tmp`). Each request must complete execution within 300 seconds. The pricing is US$0.20 per 1 million requests and US$0.0000166667 for each Gigabyte-second. AWS Lambda permits zero-cost usage per month: 1 million free requests per month and up to 400,000 GB-seconds of compute time per month.[1].

Google Functions (`https://cloud.google.com/functions`) Google Functions supports NodeJS only. Google provides a zero-cost segment for the first two million invocations per month, without regard to its duration. Beyond tthe first 2 M invocations, Google charges a per-unit rate of US$0.0000004 per invocation. Fees for compute time depend on the amount of memory (GB-seconds) and CPU (GHz-seconds) provisioned for the invoked function. Data transfer outside Google's infrastructure boundaries carries further charges. There are no specific charges for local disc (`/tmp`).

Microsoft Azure Functions (`https://azure.microsoft.com/en-us/services/functions`) Microsoft Azure Functions support languages such as JavaScript (NodeJS), C#, F#, Java, Python, PowerShell and TypeScript. Azure provides automatic scaling and the user only pays for compute resources when the functions are running. Function hosts are dynamically provisioned (allocated and deallocated), depending on incoming events. A Premium plan anticipates demand and pre-warms workers to run invoked functions. There is a free monthly allowance of 1 million requests and 400,000 GB-second of resource consumption per month; beyond that the fee for execution time is US$0.000016/GB-second and US$0.20 per million executions.

IBM Apache OpenWhisk Functions (`https://www.ibm.com/cloud/functions`) Supports languages such as JavaScript (NodeJS), Swift, Java, PHP, Go and Python. Any other language and be used as long as there is a Docker container for it. There is a free tier and, in general, the model may be pay-as-you-go (with flexible consumption) or with reserved instances. There is a basic pricing of US$0.000017 per execution second for each GB of allocated memory.

Other FaaS providers :

- Apache OpenWhisk: `http://openwhisk.incubator.apache.org/`
- Webtask: `https://webtask.io/`
- OpenFaas: `https://github.com/openfaas/faas`
- IronFunctions: `https://github.com/iron-io/functions`
- Kubeless: `http://kubeless.io/`
- Fn Project: `http://fnproject.io/`
- Spotinst Functions: `https://spotinst.com/products/spotinst-functions/`

3 Related work

3.1 Software performance engineering for cloud applications

The microservices architectural style is garnering wide adoption by the software development community, particularly for new applications. This architectural style involves the implementation and delivery of highly granular, independent, scalable, and collaborative software systems[14]. Software Performance Engineering of microservices faces several research challenges to be addressed.

According to [3], performance monitoring, testing, and modeling are the three areas where Software Performance Engineering of microservices is in need of more research and development. Aderaldo et al. [15] point out the need for more repeatable empirical research on the design, development and evaluation of microservices applications, which makes it difficult to evaluate them, since there are very few reference applications and architectures, as well as archetypical workloads to help characterize their behaviors.

The report presented in [16] provides a listing of Software Performance Engineering activities related to the integration of software development, roll-out and maintenance activities. The authors assert that, even

[1]Data correct as of April the 4th. 2021. Source: `https://aws.amazon.com/lambda/pricing/`

though practices such as continuous integration, continuous delivery and DevOps are being adopted by the Software Engineering community, they do not explicitly tackle the performance quality attribute.

Other studies, such as the one carried out in [17], indicate that the most investigated attributes are the scalability and maintainability of the microservices' code and their deployment.

3.2 *Serverless & Function-as-a-Service*

Researchers have already started to describe and analyze FaaS through a surveys, experiments [18, 19, 20], and also by cost-benefit reports [21, 22]. However, in those reports it is mentioned the need for Software Performance Engineering on FaaS in order to better understand the technology and take advantage of its service cost model.

In [3] it is mentioned that, for FaaS, new modeling strategies are required to characterize the behavior of the code over such platforms. Traditional performance models, based on the notion of independent machines, might be inadequate.

The report by van Eyk et al. [4] introduces topics such as FaaS platform comparison and evaluation, overhead reduction, scheduling policies, the cost-performance of a function, and performance prediction. The report notes that many of the evaluation and testing tasks for FaaS are complicated by the lack of reference applications, architectures, and workloads. It also introduces the concept of *information gap* where neither the FaaS engineer nor its users are aware of the hardware resources on which the functions are executed, while, on the other hand, the FaaS platform does not have information about the construction details of the function.

The use and integration of *Serverless computing* technologies within software projects is a highly active and popular practice nowadays. In [23, 24] alternative *serverless computing* architectures were studied in order to explore performance design and compare cloud providers platforms. There is also research & development on *serverless* architectures for information retrieval [19] and chatbots [25]. There are reports of use cases in fields such as Machine Learning, Security, Internet of Things, Voice processing, and File systems, among others. Despite the new possibilities of this technology, it is also reported that not much is yet known about which tools, patterns, practices, and architectures are used or required to better produce, install, and execute functions.

In [20] factors influencing the performance of *serverless* computing platforms were put under analysis. There they identified four states of a *serverless* infrastructure: provider cold, *VM cold*, *container cold* and *warm*. They also show how the performance of services can vary from 1 up to 15 times a reference measure, depending on those states.

Positioning of our study *vs.* related work The work reported here is intended to contribute and expand the area of study of software performance engineering, particularly for applications running in the Cloud. The literature review revealed the need of performance modeling approaches able to consider details surrounding applications running on the cloud. For instance, we were unable to find a single performance *modeling* approach intended for microservices, one of the most popular architectural style nowadays, which is heavily based in the usage of Cloud technologies.

This is why we undertook our exploratory study, which is aimed at determining the factors impacting the performance of a cloud function. We seek to provide new understanting on how to apply known performance engineering approaches to novel kinds of applications. This work may provide an initial reference framework through which evaluation of competing, yet comparable, technological alternatives can be made in the early stages of a project - and provide insights on capacity planning, design and behavior of software systems intended to use of cloud functions. We adapt a proven modeling approach, component-based modeling, and make it work along with a cloud function. This enables measurement of performance of the cloud function under specific payloads. If software teams had previous experience with component-based modeling, they can transfer their practice to cloud functions; no need to learn something new.

4 Case study: A Cloud-Based Image Handler

Web sites with large images can experience long load times. Image processing is well suited to be handled by a function-as-a-service (FaaS), such as those provided by the Amazon Web Services (AWS) Lambda platform. In order to lower the cost of image processing and provide short response times, AWS proposes to incorporate a *serverless* Image Handler as a Lambda function, which can receive image manipulation or optimization jobs. The architecture of "Image Handler" is detailed in [26] and appears in Figure 3.

In this context, a modified version of an image will be one that presents some alteration in size, color, metadata, etc.

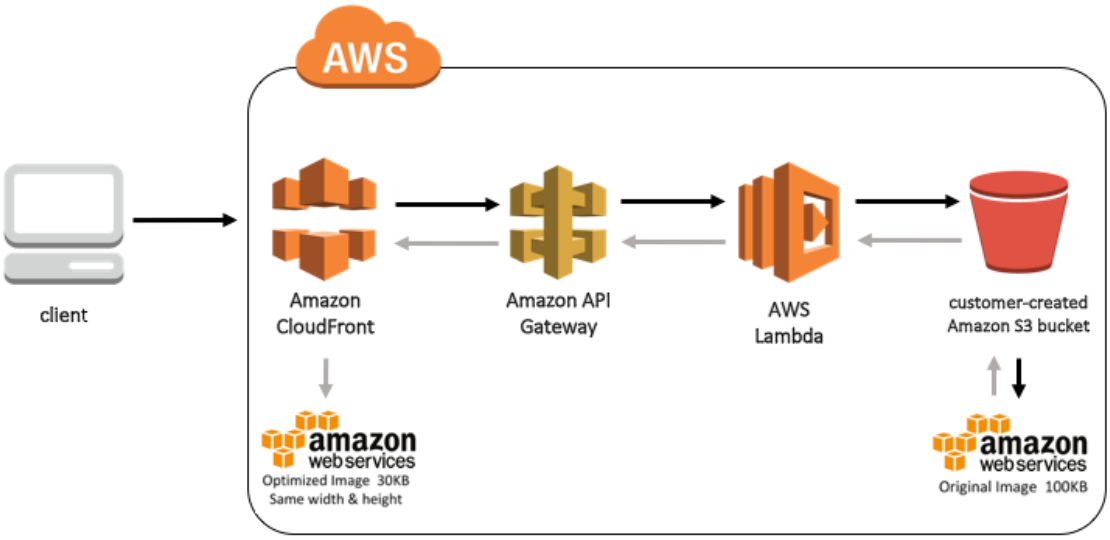

Figure 3: Architecture of the image handler. Source: [26]

4.1 Image Handler for Software performance engineering

For this study, we built an alternate version of the Image Handler architecture shown on Figure 3.

We intentionally left out AWS CloudFront and AWS API Gateway, as we looked to exercise the Lambda function directly. After a resizing invocation takes place, the Lambda function we implemented generates and returns ("on the fly") a new version of an image retrieved from an AWS S3 bucket. As an example, a request might be to resize a 500 x 500 pixels image into one with 100 x 100 pixels (width and height).

The activities involved in the image resizing process are shown in Figure 4:

1. A new image resizing invocation is sent to the Lambda function in `JSON` format. This invocation includes data about the image key (image location in AWS S3) and the new size.
2. The image resizing invocation reaches the Lambda function.
3. The Lambda function makes a call to the AWS S3 service, requesting to retrieve the original image.
4. AWS S3 returns the requested image to the Lambda function.
5. The Lambda function begins the image resizing process using the width and height parameters provided in the `JSON` invocation of step #1.
6. A new resized version of the original image is delivered to the client.

4.1.1 Image Handler Implementation

For this study, we developed a function written in the Java language, mainly to take advantage of the compatibility with the tools for application monitoring and performance model extraction approach exposed in Section 2.2.2. In what follows we describe the three versions of the Image Handler function:

IM-Simple The base Lambda function created for this study and whose name is "Image Handler". This function is responsible for performing three operations in order to process an image resizing invocation:

1. Process the `JSON` image resizing invocation. This invocation contains:
   - The name of the original image stored in the AWS S3 service.
   - The width and height parameters required for resizing the original image.
2. Obtain the image from the AWS S3 service and then apply to it the resize operation according to the height and width parameters specified in the resizing request.
3. Take the resized image, encode it in `Base64` and write the result to the output stream of the Lambda function.

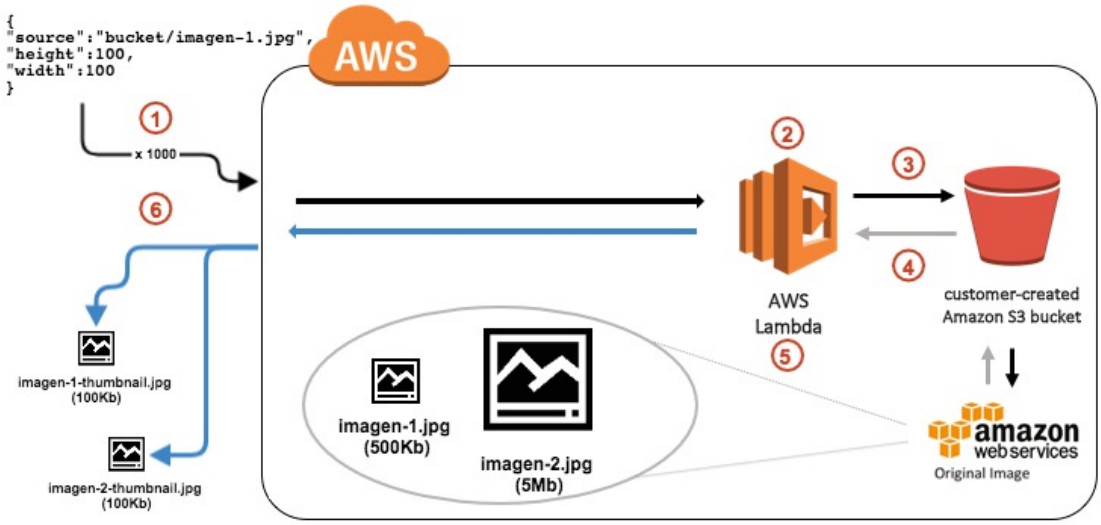

Figure 4: Worload suggested for the image handler. Source: the authors.

Instrumented version for Kieker and PMX support (IM-KP) We selected two tools, Kieker and the Performance Model Extractor (PMX), to collectively provide a framework for systematically obtaining performance measurements of an application and, from these, extract a component-based performance model (suitable for processing with the Palladio Component Model's simulation tools). We selected these tools and the performance model extraction and measurement approach after studying the *declarative performance engineering* approach proposed in [12].

In this version of Image Handler we modified the original code so as to generate records of the function's performance execution (using Kieker libraries). Additionally we configured the library to publish performance events in a Java Message Service (JMS) queue instead of a local log, which is the default configuration provided by Kieker.

Instrumented version with AWS X-Ray support (IM-XRay) AWS X-Ray[2] is an AWS tool that enables to perform behavioral analyses of applications running on AWS.

The main motivation behind this version of Image-Handler is to collect data about the invocations made to the functions, to have performance data of the Lambda function that could not be estimated during the model extraction process using the Performance Model eXtractor tool (PMX). Section 4.2 details what was observed during the PMX's model extraction process and where the results obtained by this version fit into the model.

For this version, a similar approach to IM-KP was followed, using the AWS SDK library (*Software Development Kit*) to create the traces and subsegments of traces, which are specific views of the application's behavior.

4.2 Performance model extraction strategy for Image-Handler

Because Lambda functions run in containers that are both inaccessible and ephemeral, it is necessary to record the events associated with performance behavior on an external computer or service in order to access and process the results separately.

For the extraction of the Palladio component model from the Kieker logs, the following activities were carried out:

1. Creation of the IM-KP version: this version of Image-Handler contains the Kieker libraries to generate logs from the function's performance data.
2. Provision a new virtual machine on AWS, in which:
   - A JMS queue will run.
   - A Java application will act as a consumer by polling messages from the JMS queue.
   - Store the performance data from the Lambda function in a local log.

[2] `https://aws.amazon.com/xray/`

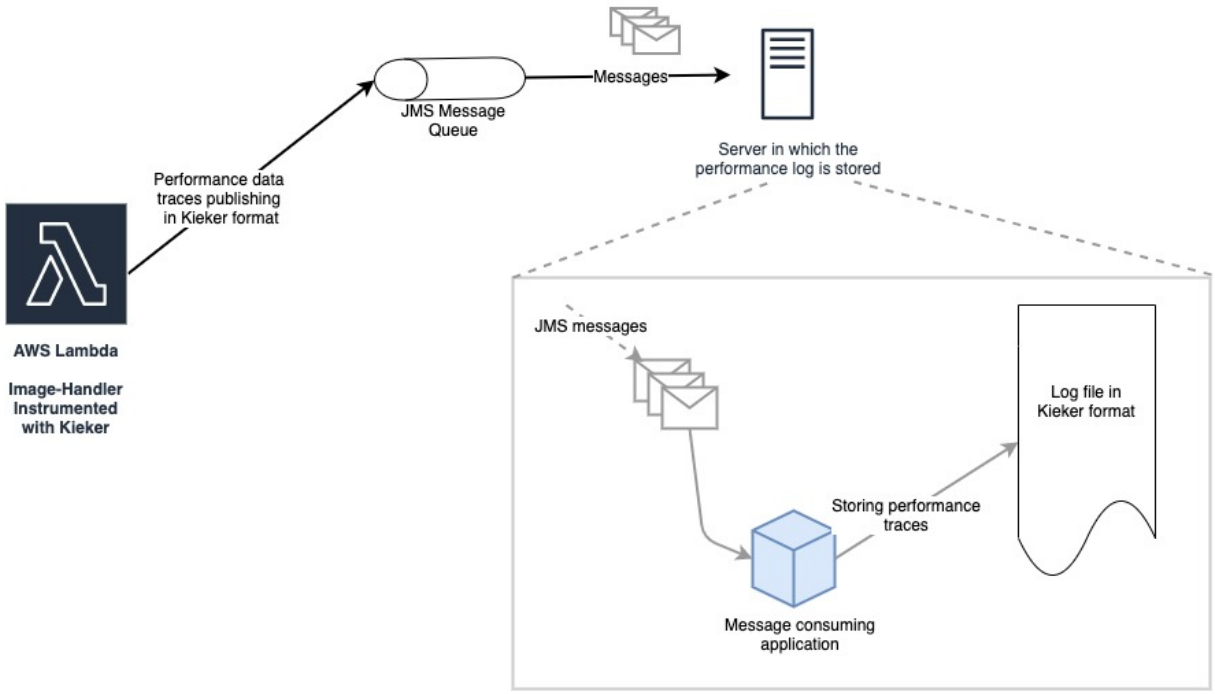

Figure 5: Image Handler performance measurements publishing process. Source: the authors.

3. Configure the Kieker library so that performance logs of the Lambda function will be published on the JMS queue in the virtual machine described in #2.

4. Build a message consuming application. The application will poll messages arriving on the JMS queue. It processes the messages from the queue and stores them in a log in the virtual machine created in #2. Figure 5 shows the participants in the process of publishing performance measurements from the Image-Handler code to an external log.

5. Once a log is obtained in Kieker format it can be used as input for the Performance Model Extractor (PMX). PMX then inspects the log, processes it and returns a `.zip` file with the corresponding files of a Palladio Component Model (PCM) instance.

4.3 Model obtained after the extraction process

PMX was able to identify 6 main components from the provided Kieker log:

1. `ImageHandlerKieker`: Function's entry point.

2. `ImageRequestParser`: Converts the resizing input invocation to an object that will be shared across other components.

3. `S3ImageService`: Holds the logic about how to resize an image.

4. `S3Dao`: Where communication with AWS S3 service takes place in order to get an image.

5. `AmazonS3Client`: Low level communication with AWS S3 SDK.

6. `HandlerResponseWriter`: Convert the newly resized image to a `Base64` representation and prepare the final function response.

As part of the refinement of the model, we identified and added a new component, and named it `AWS Lambda`. This new component's job was to estimate the impact of the AWS Lambda platform on the invocation of the function. Details about this component are discussed in Section 5.1.

During the tests performed on the model generated by PMX, we were able to verify that the generated components followed the structure of the source code and its interactions. We noticed that time consumption

estimations of each component wasn't accurate because a fixed number was always assigned to each of them. We were expecting PMX to generate some base time consumption estimation based of the function's performance log. For this reason we chose Amazon X-Ray for obtaining performance measurements. AWS X-Ray collects data from requests to application services and groups then into units called traces. Through the traces it is possible to see maps of the interaction of services, latencies and metadata to analyze behavior or to identify problems.

Traces collected with AWS X-Ray were very useful for refining performance estimations for each identified component. In experiment #1, on section 5.1, performance data for each component were collected, and frequency analyses were carried out in order to obtain empirical probability distributions. Those probabilities were introduced in the descriptors of each component, prior to executing the simulations.

5 Experimental design

In this section, we present the experiments undertaken to: - Validate whether a model and its simulation are able to characterize the behavior of the Image Handler cloud function in diverse scenarios, - Study the Lambda function's behavior when invoked with various workloads, and - Compare the results of the invocations to the Lambda function with those obtained with the Amazon Serverless Application Model (SAM) Command Line Interface (CLI), abbreviated as SAM CLI.

In all the experiments, three groups of images were used to invoke the Lambda function:

1. Images smaller than or equal to 500kB.
2. Images larger than 500kB and less than or equal to 1MB.
3. Images larger than 1MB and less than or equal to 2MB.

The image size ranges were selected based on current common recommendations regarding the handling of images targeted at the Web. Images within the first range ($x \leq 500kB$) represent general-purpose image sizes for well-performing websites. The other two ranges group images that can be used in a more specialized way within a website: images uploaded by users of social networks, higher quality images to enrich a design, or images associated with maps – among others.

5.1 Using Image-Handler to resize images of different sizes (EXP-1)

This experiment seeks to verify, via direct measurements and model simulations, the influence of different sizes of images on the function's response time. Intuitively, it is expected that requests for resizing larger images will take longer to process and the opposite will happen with smaller images. The results obtained provide an initial reference on the work performed by the software components associated with resizing and what improvements are possible.

The workloads for this experiment are *closed*: each request is executed after the previous one has finished (the invocations are strictly sequential). This is aimed at having better traceability of the function's performance.

To carry out this experiment, we obtained 1000 random images for each of the three groups of images mentioned above ($x \leq 500kB$, $500kB < x \leq 1MB$, $1MB < x \leq 2MB$) from the service *Lorem Picsum*[3]. As a sample of one of these image groups, in Figure 6 image size distribution of image group $\leq$ 500Kb are shown. Most of the image sizes in the sample are below 250Kb.

Base Measurement: 1000 invocations of image resizing on IM-Simple. We created a *script* to execute 1000 resizing invocations to the IM-Simple function on the three previously defined image groups. The *script* randomly selects an image from the group under test, then executes the resizing request with commonly used width and height for *thumbnails*. These parameters were selected from guides about *thumbnail* sizes on social networks such as Facebook, Twitter, LinkedIn, Pinterest, Tumblr, Instagram and Youtube [27, 28].

Measurements to obtain the performance model: 1000 invocations of image resizing in IM-KP and IM-XRay. To generate invocations to the Lambda function we reuse the *script* and the configuration described in the previous section.

First, 1000 invocations were executed on IM-KP to generate a Kieker log and, from it, extract a PCM performance model using PMX. 1000 IM-XRay invocations were also executed and, through a *script*, we

[3] `https://picsum.photos`.

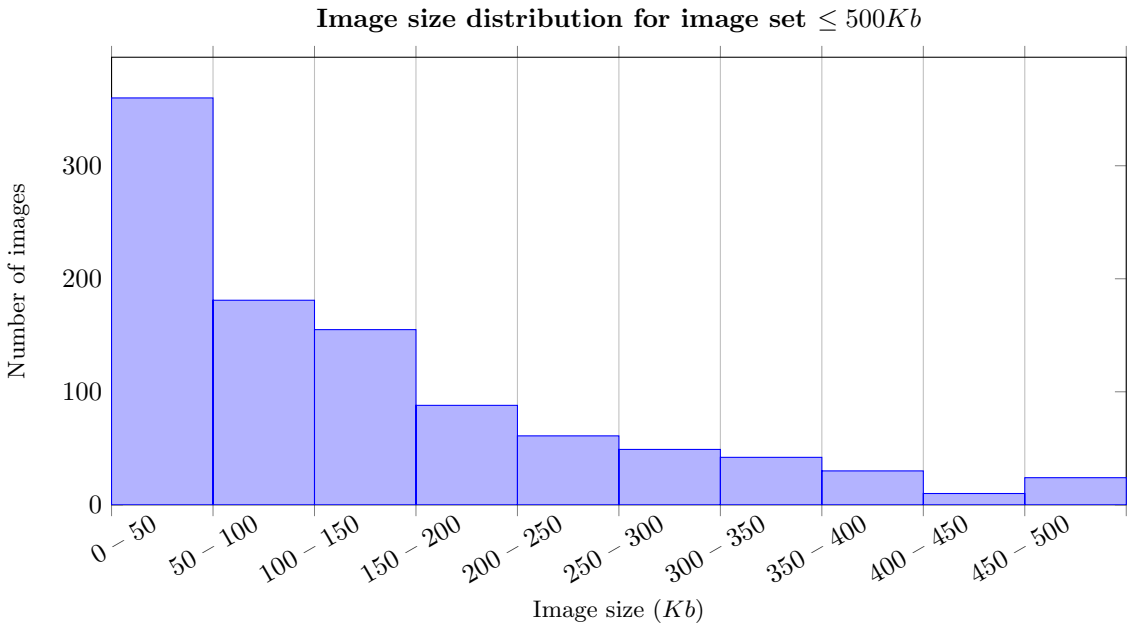

Figure 6: Image size distribution for image set $\leq 500Kb$. Source: the authors.

obtained the traces corresponding to the 1000 invocations. The new data was exported to `.csv` format and interpreted with the `R` language. In `R`, frequency distributions of the probability that a component processed a portion of the total workload were calculated. These data were included in the service level specifications for each component of the model.

Using the measurements made on IM-XRay, we observed that in the first few resizing invocations, "initialization" times were reported and that, when this occurred, the total duration of the resizing invocation was much larger than the rest . These initialization times led us to create a new component in the model (with name `AWS Lambda`). As in the other components, the initialization times reported by AWS X-Ray were taken, then exported in `.csv` format to `R` to obtain a frequency distribution of the probability and include it in the *specification* of the `AWS Lambda` component.

Finally, we ran a simulation on the *Palladio Workbench* with the following parameters:

- Generation of 1000 measurements.
- Workload: *closed*. A request on the model is executed when the previous one finishes.

5.2 Uninterrupted sequential execution of resizing requests (EXP-2)

One of the main concerns when executing functions in the cloud is to determine whether the speed with which a request is processed is low and shows a predictable trend. According to [20], a function can go through two major states: 'cold' and 'hot'. In the cold state, the platform that supports the function in the cloud must provision the necessary resources to execute the function and therefore longer response times may be observed. The hot state occurs when the platform supporting the function recognizes that it is being invoked continuously and that, due to such use, it needs to provide it with greater computational resources to obtain better response times. It is expected that, for a continuously invoked function, it will be in the cold state only a small percentage of the time and will keep itself mostly in the hot state (as long as it continues to be invoked).

Therefore, to carry out this experiment, it was proposed to execute:

- 1000 sequential invocations for resizing images with random dimensions, on each of the three defined image groups
- 1000 sequential invocations of single image resizing, for each of the three defined image groups.

5.3 Varying the intervals of resizing invocations (EXP-3)

This experiment is a variant of the one exposed in Section 5.2: we control the interval between invocations to the Lambda function in order to assess the effects on the function's behavior and whether the model

obtained in Section 5.1 helps to explain what was observed. In particular we are interested in assessing whether the intervals between invocations cause the performance of the function to reach a 'cold' or 'hot' state at some point.

5.3.1 Invocations interval strategy

To evaluate the effects of launch intervals on invocations to *Image Handler*, we proposed executing a number of sequential invocations - one emphburst -, followed by a controlled pause (timeout), and then, perform another burst of invocations.

The wait time between bursts is to be increased to twice the previous wait time. For example, if the first timeout is 2 minutes, the next will be 4 minutes, then 8, 16, 32 minutes, and so on. For this experiment we propose to use an initial waiting time of 10 minutes, and then increase it in subsequent bursts of invocations.

For this experiment we performed 100 sequential invocations of resizing images with random dimensions to IM-XRay, followed by timeouts of 10, 20 and 40 minutes.

6 Results

In this section we explain and analyze the results obtained by developing the experiments described in section 5. A sub-section is dedicated to each of the experiments described previously (*EXP-1*, *EXP-2*, and *EXP-3*).

6.1 EXP-1

These are the results for the experiment in Section 5.1. Figure 7 shows the results of the simulations for the images $\leq 500kB$.

There is a difference of 209.965ms in the average time of the response times of the simulations in PCM with respect to the times of IM-Simple. This is because the model for the simulations was built from data generated by instrumented code (which imposes an execution *overhead*).

The results show standard deviations of 460.569ms and 465.445ms for IM-Simple and PCM simulations, respectively. The standard deviation of the PCM simulations is only 4.782ms greater than that of IM-Simple, suggesting that the groupings of the data with respect to their arithmetic means will be very similar.

The results of the PCM simulations point to a 95% probability that the processing time of a resizing request for an image of size $\leq 500kB$ will take 1.6 seconds or less (Figure 8). In tests of IM-Simple, a 97.5% probability was obtained for the same case.

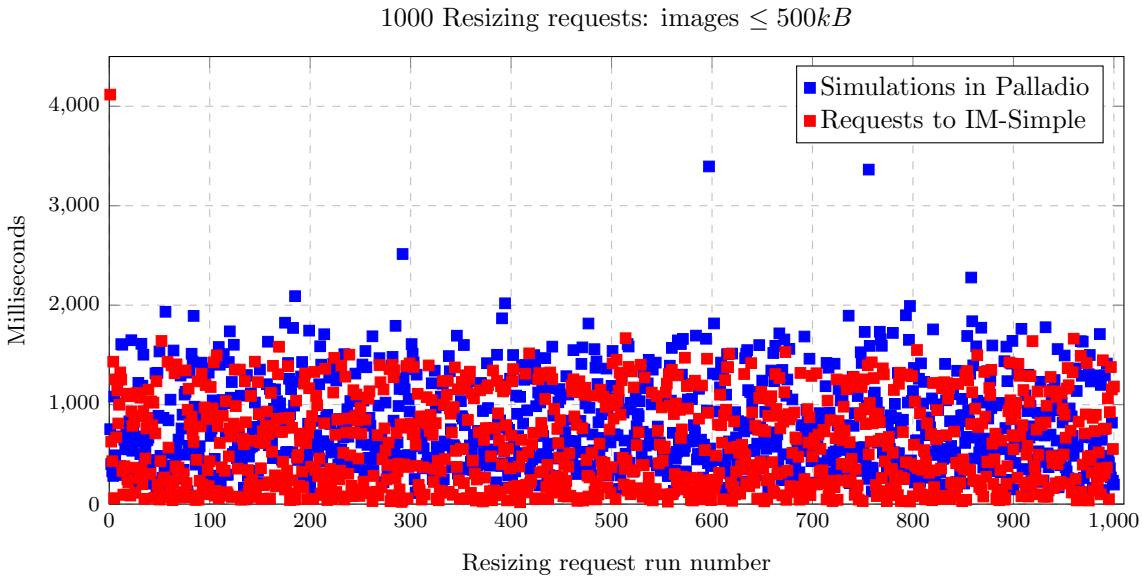

Figure 7: IM-Simple *vs* simulations in PCM: response times of 1000 requests for resizing images of size $\leq 500kB$. Authors' results.

For image sizes $500kB < x \leq 1Mb$, the comparison of the response times of IM-Simple and PCM simulations are shown in Figure 9. There is a difference of 274.428ms in the average time of the response times in the PCM simulations with respect to the times of IM-Simple. The results show standard deviations of 1731.974ms and 1844.893ms for IM-Simple and PCM simulations respectively. The standard deviation of

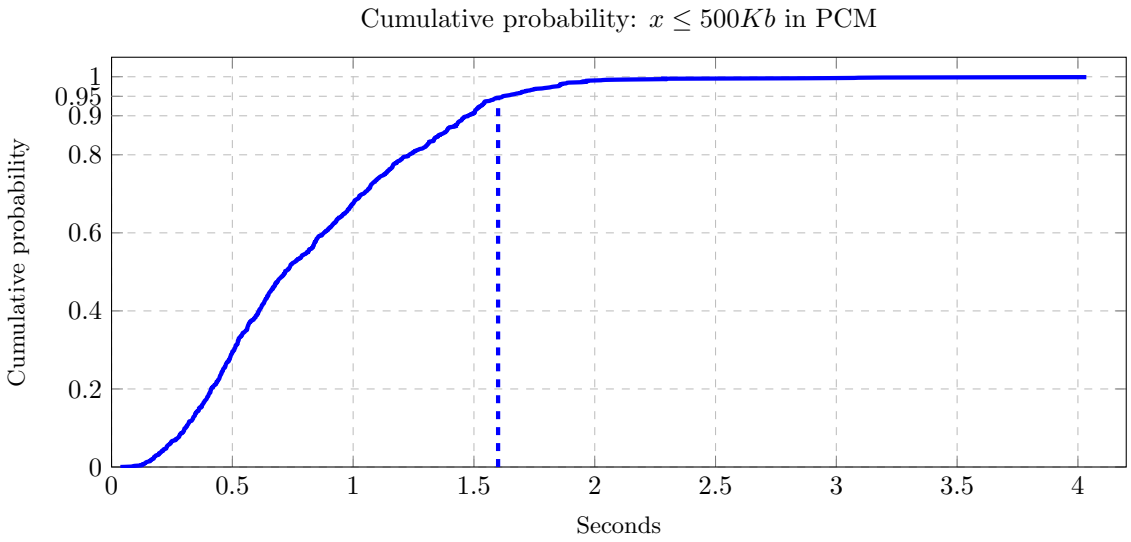

Figure 8: Cumulative probability in resizing requests for images sizes $\leq 500kB$ in *Palladio Workbench*. Authors' results.

the PCM simulations is 112.919ms greater than that of IM-Simple. This indicates that the grouping of the data with respect to its arithmetic mean is very similar between both samples.

According to the simulations' results, there is a 95% probability that the processing time for a resizing request of an image of size $500kB < x \leq 1Mb$ will take 8 seconds or less. In IM-Simple, a 98% probability was obtained for the same case.

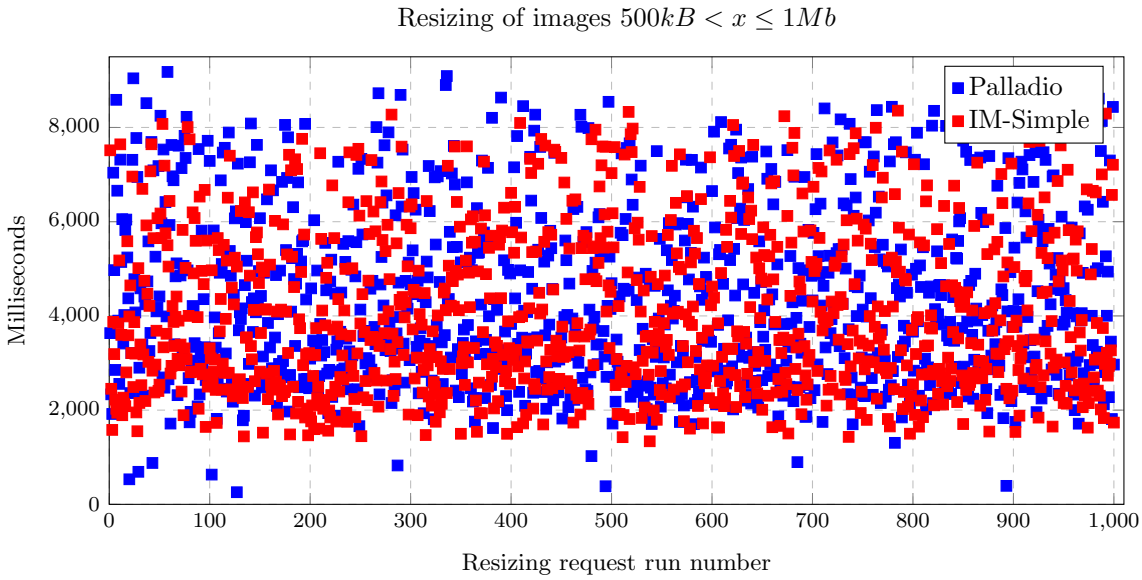

Figure 9: Response times for resizing requests on images of sizes $500kB < x \leq 1Mb$. Authors' results.

In the last group of images ($1Mb < x \leq 2Mb$) there is a difference of 257.773ms in the average time of the response times between PCM simulations IM-Simple.

The results show standard deviations of 1816.152ms and 1914.258ms for IM-Simple and PCM simulations respectively. The standard deviation of the PCM simulations is 98.106ms greater than that of IM-Simple. In the simulation results, there is a 95% probability that the processing time will take 10.14 seconds or less for a resizing request on image of sizes $1Mb < x \leq 2Mb$. In IM-Simple, processing 100% of the requests took 10 seconds or less.

In this case, the average processing time per resizing request could be used to characterize the behavior of the Lambda function, but, as in the two previous cases, we consider that the use of cumulative probabilities is suitable for providing accurate predictions. These results are synthesized visually in Figure 10.

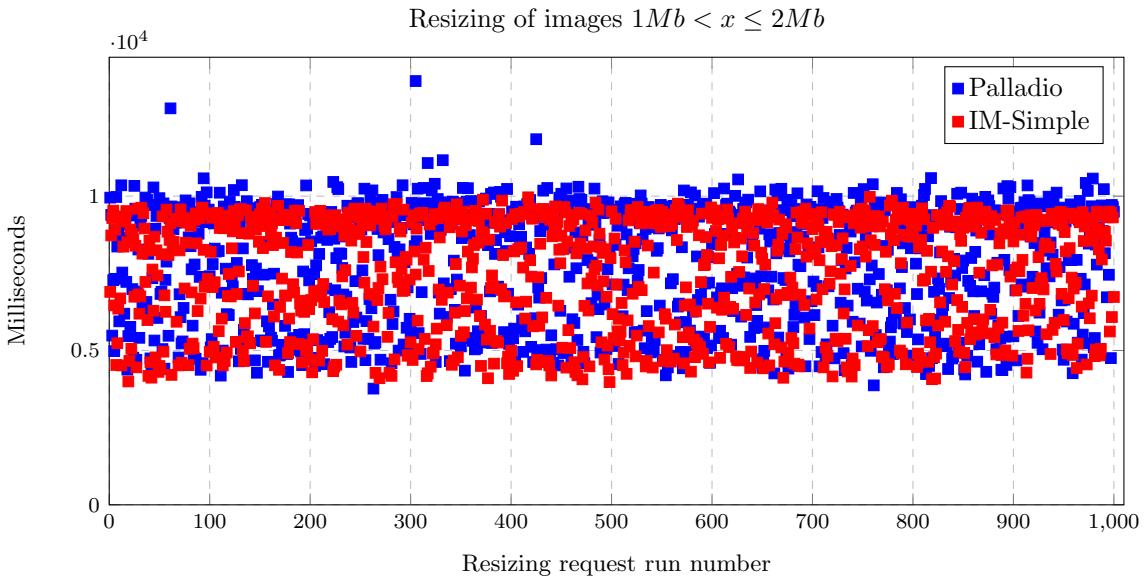

Figure 10: Response times for resizing requests on images of sizes $1Mb < x \leq 2Mb$. Authors' results.

6.2 EXP-2

After performing resizing invocations on both random images and a single image, it could be observed that, although the first invocations experienced higher response times, the response times delivered by the Lambda function dropped significantly and, in the cases of resizing invocations on a single image, such times were stable. An example of this behavior can be seen in Figure 11.

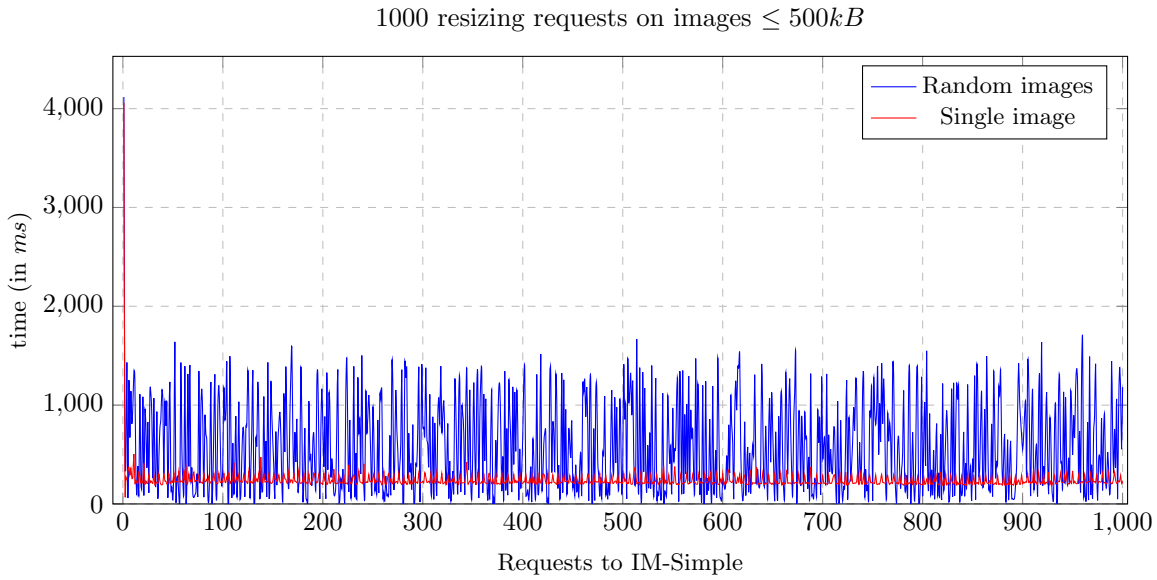

Figure 11: 1000 resizing requests on images $\leq 500kB$. The blue line represents requests made using random images. The red line, resquests on a single image. Authors' results.

This behavior suggests that when invoking a Lambda function on AWS, the underlying platform first needs to perform provisioning tasks to make the function available. Once this is done, the function gradually goes into a "hot" state according to the arrival of resizing invocations. Although the architecture of the AWS Lambda service is not publicly available, in the documentation for parallel projects such as `SAM CLI` and *AWS Lambda container image converter tool*[4] it is suggested to use Docker containers that are provisioned with the code of the function and take care of processing the incoming invocations.

[4] `https://github.com/awslabs/aws-lambda-container-image-converter`.

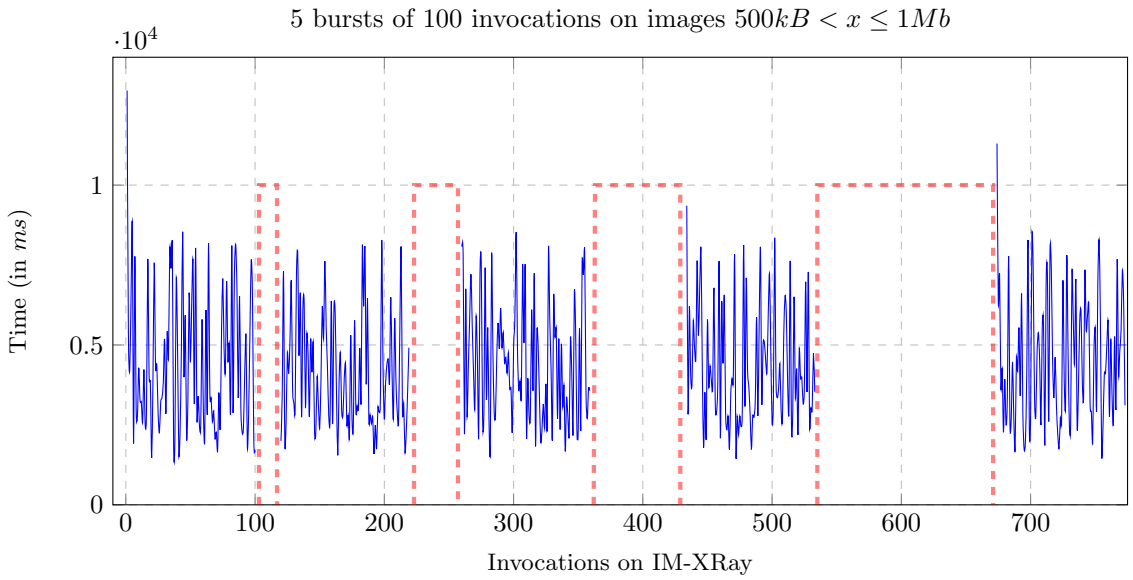

Figure 12: The space delimited by the red dotted line represents 10 minutes of inactivity between the first burst and the second. Thereafter, the space delimited by red dotted lines corresponds to 20, 40 and 80 minutes of inactivity between bursts. Authors' results.

6.3 EXP-3

Bursts of a hundred resizing invocations were executed and we intentionally introduced 10, 20, 40 and 80 minute programmed downtimes between each burst. The same three groups of images from experiment # 1 were used in the resizing invocations. As the idle time between bursts increased, we observed an increase also in the possibility that the function fell into a "cold" state and, although this behavior was not explicitly introduced in the performance model, it was possible to notice a correspondence between the response times of the Lambda function in "cold" and "hot" states in the individual bursts with the results obtained via the simulations.

Our main observation regarding this experiment is that, as idle times in the Lambda function increase, so do the response times delivered by the first invocation after a period of inactivity. An example of this behavior is illustrated in Figure 12.

According to [29], once a function has been installed on the platform, the first invocation must go through the provisioning process. After processing the first invocation, the function goes into an active or "hot" state and the container that supports it can be reused for subsequent invocations. When it is detected that the function becomes inactive or "cold", the container that supports it becomes a candidate to be "recycled" to be used by another function that needs it.

Although it is not possible to know with certainty whether there exists a defined time limit to decide when the container is going to be recycled or not, the results of this experiment and those exposed in [29] and [30] show that the longer the idle times of a function, the greater the probability that the container that supports it will be recycled and that when the function is invoked again, a higher than average response time will be experienced - due to the provisioning process.

According to the measurements in [29], the lifetime of a container does not appear to be deterministic but is estimated to be between 25 to 65 minutes. An inactive container almost always stays alive for 25 minutes, after that the probability of it being discarded grows slowly and reaches 100% after an elapsed time of about 1 hour since the last invocation. Those measurements coincide with the ones obtained in our experiments.

6.4 Discussion

Table 1 summarizes the quantitative results of our experiments. There are differences between the mean execution times of IM-Simple and the mean times predicted by Palladio (PCM) simulations. These differences can be explained in terms of the instrumentation required to obtain basic performance data captured by Kieker. This instrumentation causes unavoidable overhead, which was measured to be between 209.965ms and 274.428ms depending on the original images' sizes. Once the instrumentation is removed, we observe that the simulation results agree with 95% probability or higher. Furthermore, standard deviation can be attributed to platform provisioning for FaaS execution moving from the "cold" to the "hot" state. When

the state is "cold", the underlying platform must provision resources to run the function, and this takes time. As demand grows, more resources have to be provisioned, but these will be "recycled" to execute new invocations that activate instances of the function being invoked. Our experiments confirmed that when function invocations decrease in number or frequency the execution platform's resources start to be freed, and the state gradually passes from the "hot" to the "cold" state, and that this happens between 40 to 80 minutes of inactivity.

Summary of statistical data from the experiments			
$Image\ size < 500kB$			
Resizing request	IM-Simple	Palladio	Difference
Average time	583.842ms	793.808ms	209.965ms
Standard deviation	460.659ms	465.441ms	4.782ms
Variance	212206.961	216635.000	–
Median	466.715ms	680.482ms	–
Coefficient of variation	0.987	0.683	–
$500Kb < Image\ size \leq 1Mb$			
Average Time	4073.600ms	4348.029ms	274.428ms
Standard deviation	1731.974ms	1844.893ms	112.919ms
Variance	2999736.844	3403633.840	–
Median	3658.825ms	3989.406ms	–
Coefficient of variation	0.473	0.462	–
$1Mb < Image\ size \leq 2Mb$			
Average time	7539.139ms	7796.913ms	257.773ms
Standard deviation	1816.152ms	1914.258ms	98.106ms
Variance	3298410.017	3664385.000	–
Median	8200.875ms	8310.293ms	–
Coefficient of variation	0.221	0.230	–

Table 1: Summary of statistical data from the performed experiments. Authors' results.

Experiments have to be designed carefully in order to obtain the base data required for feeding the component models that are used in simulations via the Palladio approach.

7 Conclusions

In this article, we explored component-based modeling and simulation to understand the performance of functions running in the cloud (*FaaS*). We sought to evaluate whether this methodological approach offers relevant information regarding their behavior, with a view to reasoning about the design and construction of software architectures, as well as predicting the performance of software systems that use cloud functions.

The main challenge of this research was to face a *black box*. There were no prior performance models for characterizing the behavior of cloud functions with different workloads. Previous models of the AWS Lambda platform were not available for consultation, nor open documentation on its implementation. It was necessary to explore, test and measure the software system to identify its components and, iteratively, build a plausible model.

It was necessary to develop a function for resizing web images and install it as a service on top of AWS Lambda. Additionally, we adapted the declarative performance engineering approach described in [12], which enabled automating much of the process.

The main finding of this research is that component-based modeling and simulation *can* accurately characterize the performance behavior of a function in the cloud. After publishing a function as a service on a cloud platform, and subsequently invoking it thousands of times, and also running thousands of simulations on the performance model, it was possible to corroborate that the sets of results obtained in both scenarios presented similar trends, which was statistically established.

The instrumentation made in the source code to support the monitoring of events inside the function caused the simulations to report an average difference of 247ms with respect to the average response times of the invocations made to the Lambda function (without instrumentation). The simulations carried out on the obtained model were able to characterize more than 95% of the *Image Handler* function's real performance.

The simulations with the highest response times (< 1 % of the total), coincide with the executions of the *Image Handler* function in a "cold" state.

The methodological approach that supports the study reported in this article is summarized in Appendix A and is explained in detail in [31].

8 Future work

The modeling and simulation of cloud functions performance can be studied at various levels, in order to improve the understanding of FaaS platforms and their main components. It is also important to model and simulate cloud functions performance that work collaboratively, for this pattern of software development is growing in adoption. It is necessary to investigate tools for prototyping functions in the cloud that consider the particularities of production environments, as current tools provide limited support and this complicates testing.

Having worked with *Open Source* modeling tools, such as the Palladio Workbench, invites the development of components specifically oriented to the modeling and simulation of applications in cloud environments. It is also worth studying the performance of FaaS *Open Source* platforms. The methodological approach outlined in [31] may be transferred to other commercial cloud functions-as-a-service platforms, beyond AWS Lambda.

The authors are working on an iterative design process that incorporates early modeling and performance measurement of architectural prototypes of systems aimed at cloud platforms. This will enable rational comparisons of design alternatives prior to committing to building one of them.

A Appendix: Methodological guidance

We summarize the five more relevant activities for extracting a performance model from a Lambda Function. Those activities are:

1. Use case selection.
2. Instrument the code with Kieker.
3. Model extraction with the Performance Model eXtractor tool (PMX).
4. Import files generated by PMX into the *Palladio Workbench*.
5. Refinements or calibration on the model.

A more detailed methodological guide can be found in [31].

A.1 Use case selection

```
1  /* Expert users may skip this instrumentation and extraction process          */
2  if there is already an existing PCM model then
3  |  use that model for simulations and testing;
4  else
5  |  if function is written in Java then
6  |  |  Kieker SDK may be used ;
7  |  else
8  |  |  integration with Kieker has to be made explicitly;
9  |  end
10 |  if FaaS platform is AWS Lambda then
11 |  |  Kieker integration can be done through JMS;
12 |  |  monitoring tools provided by AWS can be used;
13 |  else
14 |  |  /* Kieker integration may be similar to the method described above or could
   |  |     require additional work                                              */
15 |  |  integration with Kieker has to be made explicitly;
16 |  |  alternate measurement tools need to be used;
17 |  end
18 end
```

Algorithm 1: Use case selection: Lambda Function and initial Kieker integration

Despite the specific use case of the Lambda Function itself, in this study we decided to use Java as the primary language because the tools we used for model extraction and estimations (namely Kieker, PMX and Palladio Workbench) are also written in Java. This allowed us to make an easier integration of the Lambda's code withe the Kieker SDK: Kieker-compliant performance traces were generated using the Kieker SDK within our Lambda's code.

Since we developed our use case in AWS Lambda, we employed AWS monitoring tools such as AWS X-Ray and CloudWatch to refine and calibrate our models.

Attention: if there is a performance model at hand, the steps described above can be skipped and monitoring tools may be used to improve the accuracy of the model.

A.2 Instrument code with Kieker

```
/* We assume the Lambda Function is written in Java and deployed in AWS Lambda   */
Provision a new virtual machine in EC2;
Install, configure, and run Kieker according to [6];
Install, configure, and run a JMS queue;
Configure Kieker as the queue's message consumer;
Intrument the Lambda's code with the Kieker's SDK;
Generate OperationExecutionRecord events from the Lambda's code;
Make sure OperationExecutionRecord events are published to the queue of step 4;
while incoming resizing invocations do
    (λ Function) Publish performance events to the JMS queue;
    (JMS) Manages the incoming messages;
    (Kieker) Consumes messages from the queue and creates new traces in a local log;
end
```

Algorithm 2: Steps to instrument the Lambda's code with Kieker

The Kieker SDK can be used to instrument the code of a Lambda Function written in Java. This can be made by publishing *OperationExecutionRecord* events during the execution of an operation of interest (i.e. when the code initiates and finishes relevant computation). For this study we decided to publish those events through a JMS queue. The Kieker documentation in [6] offeres alternate Kieker configurations for a variety of projects.

A.3 Model extraction with PMX

```
Download and install PMX;
Run PMX to generate a performance model (PCM) from a Kieker log;
if No PCM model is generated after PMX then
    Review the Kieker instrumentation in the Lambda's code;
    Review Kieker's user manual;
end
```

Algorithm 3: Performance model extraction with PMX

The Performance Model eXtractor (PMX) tool can be downloaded from `https://se.informatik.uni-wuerzburg.de/software-engineering-group/tools/pmx/`. Once downloaded, PMX can be run either by executing `pcm-pmx-server` or `pcmConsole.jar`. Running PMX with a Kieker log as an input generates a set of XML files, which represent the *extracted Palladio Component Model* model.

A.4 Import files generated by PMX into the *Palladio Workbench*

```
/* We assume a PCM model was obtained through PMX                                 */
Create new project in Palladio Workbench;
Import file generated by PMX;
if files are compatible with Palladio Workbench then
    Simulation running can start;
else
    Evaluate incompatibilities and correct;
end
```

Algorithm 4: Import files generated by PMX into the *Palladio Workbench*

If the previous processes have executed sucessfully, then PMX will generate 5 XML files. Each of them will represent one of the models that compose an instance of *Palladio Component Model*. Those files have to be imported into the *Palladio Workbench* so that simulations can start running over the model. For the study reported here, we used the *SimuBench* simulation engine.

There is a chance that files generated by PMX to be incompatible with the installed version of *Palladio Workbench*. A valid XML file from an existing PCM project has to be compared with the ones generated by PMX in order to fix them.

A.5 Refinements and calibrations of the model

```
1 /* Assumption:  There is a valid PCM model                                  */
2 Execute simulation with SimuBench;
3 repeat
4 |   Modify submodels;
5 |   Evaluate new results;
6 |   Get new estimations from alternate measurement tools;
7 until results are good enough for the current study;
```

Algorithm 5: Simulation execution and model refinement.

Once a valid PCM model is available, simulations can be started. *Palladio Workbench* provides several visualizations of simulation results; what is most important is that, once simulations are available, analysis is possible and improvements to the model can be made - if needed. The main idea here is to start a run-then-analysis-then-calibration cycle to improve the accuracy of the model.

References

[1] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, ser. WoSC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–4. [Online]. Available: https://doi.org/10.1145/3154847.3154848

[2] Amazon Web Services. (2021) Serverless computing - Amazon Web Services. [Online]. Available: https://aws.amazon.com/serverless

[3] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: Research challenges and directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: ACM, 2017, pp. 223–226. [Online]. Available: https://doi.org/10.1145/3053600.3053653

[4] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG Cloud Group's vision on the performance challenges of faas cloud architectures," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 21–24. [Online]. Available: https://dl.acm.org/doi/10.1145/3185768.3186308

[5] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2009.07.007

[6] K. Project, "Kieker 1.13 user guide," Oct 2017, http://kieker-monitoring.net/documentation/, obtenido el 15 de Abril del 2019. [Online]. Available: http://kieker-monitoring.net/documentation/

[7] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016.

[8] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 171–187. [Online]. Available: https://doi.org/10.1109/FOSE.2007.32

[9] T. de Gooijer, "Performance modeling of `ASP.Net` web service applications : an industrial case study," 2011. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A432058&dswid=-1479

[10] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009. [Online]. Available: https://doi.org/10.1016/j.jss.2008.03.066

[11] O.-Q. Noorshams, "Modeling and prediction of i/o performance in virtualized environments," Ph.D. dissertation, 2015. [Online]. Available: http://dx.doi.org/10.5445/KSP/1000046300

[12] J. Walter, S. Eismann, J. Grohmann, D. Okanovic, and S. Kounev, "Tools for declarative performance engineering," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 53–56. [Online]. Available: http://doi.acm.org/10.1145/3185768.3185777

[13] N. Novkovic, "Top function as a service (FaaS) providers," May 2018, https://dashbird.io/blog/top-function-as-a-service-faas-providers/, obtenido el 22 de Noviembre del 2018. [Online]. Available: https://dashbird.io/blog/top-function-as-a-service-faas-providers/

[14] M. Cavallari and F. Tornieri, "Information systems architecture and organization in the era of microservices," in *Network, Smart and Open*, R. Lamboglia, A. Cardoni, R. P. Dameri, and D. Mancini, Eds. Cham: Springer International Publishing, 2018, pp. 165–177.

[15] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, May 2017, pp. 8–13. [Online]. Available: https://dl.acm.org/doi/10.5555/3101282.3101285

[16] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented DevOps: A research agenda," *CoRR*, vol. abs/1508.04752, 2015. [Online]. Available: http://arxiv.org/abs/1508.04752

[17] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, April 2017, pp. 21–30. [Online]. Available: https://doi.org/10.1109/ICSA.2017.24

[18] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *CoRR*, vol. abs/1706.03178, 2017. [Online]. Available: http://arxiv.org/abs/1706.03178

[19] M. Crane and J. Lin, "An exploration of serverless architectures for information retrieval," in *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, ser. ICTIR '17. New York, NY, USA: ACM, 2017, pp. 241–244. [Online]. Available: http://ezproxy.itcr.ac.cr:3837/10.1145/3121050.3121086

[20] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, April 2018, pp. 159–169. [Online]. Available: https://ieeexplore.ieee.org/document/8360324

[21] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 179–182.

[22] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, Oct 2017, pp. 1–6.

[23] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 405–410.

[24] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[25] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, ser. MOTA '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:4. [Online]. Available: http://ezproxy.itcr.ac.cr:3837/10.1145/3007203.3007217

[26] Amazon Web Services, "Serverless Image Handler – AWS answers." [Online]. Available: https://aws.amazon.com/solutions/implementations/serverless-image-handler/

[27] N. Zarzycki, "Social media image sizes: A quick reference guide for each network," Jul 2019. [Online]. Available: https://blog.hootsuite.com/social-media-image-sizes-guide/

[28] J. Spencer, "2020 Social Media Image Sizes Cheat Sheet," Dec 2019. [Online]. Available: https://makeawebsitehub.com/social-media-image-sizes-cheat-sheet/

[29] M. Shilkov, "Cold starts in AWS Lambda," Jun 2019. [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/

[30] ——, "When does cold start happen on AWS Lambda?" Jun 2019. [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/intervals/

[31] M. Flores-González, "Modelado y simulación de funciones en la nube en plataformas *Function-as-a-Service* (*Modeling and simulating Cloud functions in* Function-as-a-Service *platforms*)," Master's thesis, Programa de Maestría en Computación, Tecnológico de Costa Rica, 2019.