Moreno-Vozmediano *et al.*
Journal of Cloud Computing (2023) 12:108
https://doi.org/10.1186/s13677-023-00485-9

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Latency and resource consumption analysis for serverless edge analytics

Rafael Moreno-Vozmediano[1*], Eduardo Huedo[1], Rubén S. Montero[1] and Ignacio M. Llorente[2]

Abstract

The serverless computing model, implemented by Function as a Service (FaaS) platforms, can offer several advantages for the deployment of data analytics solutions in IoT environments, such as agile and on-demand resource provisioning, automatic scaling, high elasticity, infrastructure management abstraction, and a fine-grained cost model. However, in the case of applications with strict latency requirements, the cold start problem in FaaS platforms can represent an important drawback. The most common techniques to alleviate this problem, mainly based on instance pre-warming and instance reusing mechanisms, are usually not well adapted to different application profiles and, in general, can entail an extra expense of resources. In this work, we analyze the effect of instance pre-warming and instance reusing on both application latency (response time) and resource consumption, for a typical data analytics use case (a machine learning application for image classification) with different input data patterns. Furthermore, we propose extending the classical centralized cloud-based serverless FaaS platform to a two-tier distributed edge-cloud platform to bring the platform closer to the data source and reduce network latencies.

Keywords Serverless computing, Function as a Service (FaaS), Edge computing, Cloud computing, Data analytics, Internet of Things (IoT)

Introduction

Stream or real-time data analytics in IoT environments [1] involves the analysis of large volumes of incoming data as soon as they are stored or created. The IoT applications that generate this kind of data streams can be of quite different nature [1, 2], such as e-health, manufacturing, traffic control systems, cameras and surveillance systems, energy management, smart transportation, smart cities, etc. To process these streams efficiently, a real-time data analytics platform should exhibit several key features [3], namely: low latency, high-availability, and horizontal scalability. Considering these features, the serverless computing model is a suitable candidate for supporting real-time data analytics [4, 5].

Serverless computing [6, 7] aims to abstract infrastructure management from end users and application developers. The cloud provider is responsible for allocating, deploying, and scaling the resources required to meet the needs of the user's applications, while users are charged only for the time their code is running. In a serverless environment, the application logic is commonly implemented as a set of stateless functions that are triggered by events (e.g., API calls, message queues or scheduled tasks), and are executed by containerized or micro-VM based runtime environments. The platforms that incarnate this serverless model are categorized as Function as a Service.

One of the major drawbacks of current FaaS platforms for supporting low-latency applications, including stream data analytics, is the cold start problem [8, 9]. Cold start arises when a function is invoked, but there is not any runtime environment ready to execute this function, so a new instance (usually, a container or a micro-VM) must be spun up, along with the

*Correspondence:
Rafael Moreno-Vozmediano
rmoreno@ucm.es
[1] Computer Science School, Complutense University, Madrid, Spain
[2] OpenNebula Systems, Madrid, Spain

appropriate execution environment and the function code. There are several techniques for reducing cold start times, mainly based on instance pre-warming and instance reusing. Despite the potential benefits of these techniques, many existing FaaS platforms employ simplistic approaches when implementing them. For instance, in the case of instance reusing, most commercial FaaS platforms rely on a pre-defined keep-warm interval that cannot be customized by the user [10]. Similarly, when it comes to instance pre-warming, many of these platforms place the responsibility of selecting the number of pre-warmed instances on the user [11]. These mechanisms are not optimized for varying application profiles and may result in unnecessary resource consumption, leading to increased costs and energy consumption of the underlying infrastructure.

Another problem for these applications is the raw network latency [12], which can be reduced by moving the FaaS platform to the edge of the network. Edge computing platforms can provide computational capacity near data generating devices or users. They consist of several geo-distributed micro-data centers, with limited resource capacity, located at the edge of the network, such as user facilities, telecommunications access networks, or ISPs. Serverless can complement the edge computing model, by providing on-demand resource provisioning for edge applications while minimizing resource requirements and lowering latency responses to event triggers.

The goal of this paper is twofold. On one hand, we carry out an in-depth analysis and fine tuning of the instance pre-warming and instance reusing mechanisms for reducing the cold start problem in FaaS platforms. Most existing works only focus on the effect of these mechanisms on the application latency (response time), but they ignore the extra consumption of resources that they can entail. In this work, we analyze the effect on both the response time and the resource usage when using a different number of pre-warmed instances and keep-alive intervals for a machine learning application for image recognition [13] with different input data profiles. On the second hand, we propose the extension of these mechanisms to a two-tier edge-cloud platform, where resources can be provisioned on-demand either in the edge node or in the cloud site, depending on the edge resource availability, and according to several placement policies. The experimental results presented in this work have been obtained using a trace-driven FaaS platform simulator that implements several cold-start and allocation policies.

The main contributions of this work are the following:

- In this work, we analyze the effect of instance pre-warming and instance reusing mechanisms, isolated or combined, on both response time and resource consumption in serverless applications.
- We propose to extend the classical centralized cloud-based serverless FaaS platform to a two-tier distributed edge-cloud platform to bring the platform closer to the data source and reduce network latencies.
- We propose and compare two different allocation heuristics for this two-tier edge-cloud serverless platform, called edge-first and warm-first policies, which try to reduce the network latency first or the instance initialization time first, respectively.
- We present FaaSim [14], a trace-driven FaaS platform simulator that implements the different cold-start reduction mechanisms and allocation policies analyzed in this paper.

The paper is organized as follows: *State of the art* section analyzes the existing serverless computing platforms, the main strategies proposed for cold-start reduction, and the extension of the serverless computing paradigm to edge computing platforms. *FaaS execution model and cold-start reduction mechanisms* section analyzes the various stages and time components of the FaaS execution model and presents the two main mechanisms for cold-start reduction: instance pre-warming and instance reuse. The extension of serverless model to a two-tier edge-cloud platform is studied in *Extending serverless platforms to two-tier edge/cloud environments* section. The use case used in this work and the FaaSim simulator are described in the *Experimental environment* section. The section of *Results* conducts different experiments for performance and efficiency evaluation. Finally, the paper is concluded in *Conclusions and future work* section.

State of the art

Most of the prominent cloud computing providers are currently offering serverless computing capabilities. Amazon's AWS Lambda [15] was the first serverless platform, which is integrated into the large AWS ecosystem of services. Lambda functions can be easily created in several programming languages, such as Node.js, Java, Python, or C#, and can be associated with a variety of trigger events including changes to the state of a storage account, web service invocations, stream events and even workflow events. Google Cloud Functions [16] also provide FaaS functionality to run serverless functions written in different languages (Node.js, Python, Java, and more) in response to HTTP calls or events from some Google Cloud services. Microsoft Azure Functions [17] provides HTTP webhooks and integration with Azure services to run user provided functions. The platform supports C#, F#, Node.js, Python, PHP, bash, or any executable.

There are also some open source serverless initiatives [18–20]. For example, OpenFaas [21] is a serverless function engine that is part of the Cloud Native Computing Foundation (CNCF), which enables developers to publish, run, and manage functions on Kubernetes clusters. The main component of the OpenFaas framework is the API gateway, which provides access to the functions from outside the Kubernetes cluster, collects metrics and provides scaling by interacting with the Kubernetes orchestration engine. Apache OpenWhisk [22] is another open-source platform that provides event-based serverless programming with the ability to chain serverless functions to create composite functions. It supports Node.js, Java, Swift, Python, as well as arbitrary binaries embedded in a Docker container. Another interesting project is OpenLambda [23], an open-source implementation of the Lambda model, which provides faster function startup time for heterogeneous language runtimes and across a load balanced pool of servers, and the ability to respond quickly and automatically to load surges by scaling the number of workers. Kubeless [24] is a Kubernetes-native [25] serverless framework with a programming model based on three primitives: functions, triggers, and runtime. A function is a representation of the code to be executed, and a trigger is an event source. A trigger can be associated to a single function or to a group of functions depending on the event source type. Knative [26] is another framework built on top of Kubernetes and Istio [27] that support the deployment of serverless applications and functions, by offering rapid deployment of serverless containers, automatic scaling up and down to zero, routing and network programming for Istio components, point-in-time snapshots of deployed code and configurations and serving resources.

Regarding the cold-start problem in serverless platforms, there are several solutions that help to reduce this problem. One simple solution is instance reuse, which consists of keeping the instance alive for a while after the end of the execution of a function call, so that it can be reused as a warm instance to execute a new invocation of the same function. Most commercial FaaS platforms (e.g., AWS Lambda, Microsoft Azure Functions, or Google Cloud Functions) use this technique to reduce the cold start time. However, the exact keep alive interval used by these providers is a parameter that is not well documented, and, in any case, is not configurable. There exist also some plugins, such as the Serverless WarmUp plugin [28] for AWS Lambda, which creates a scheduled lambda that invokes all the selected service's lambdas in a configured time interval (5 min, by default), forcing the lambda function instances to stay warm. Some other improvements to the basic keep-alive mechanism have been proposed, for example in [29] the keep-alive interval is adapted to each particular workload, according to its actual invocation frequency and pattern, [30] uses caching-based techniques to implement a greedy-dual keep-alive policy based on the memory footprint, access frequency, initialization cost, and execution latency of different functions, and [31] proposes the IceBreaker technique, which reduces the service time and the keep-alive cost by composing a system with heterogeneous nodes (costly and cheaper), by dynamically determining the cost-effective node type to warm up a function based on the function's time-varying probability of the next invocation. Another technique for reducing the cold start problem is instance pre-warming, which consists of starting in advance a given number of function instances that stay always alive during the serverless application lifecycle and can run different invocations of the same function. Many FaaS platforms implement this mechanism, for example, AWS Lambda offers the *provisioned concurrency* [32] feature to keep a number of containers initialized ready to execute lambda functions with minimum delay; similarly, Microsoft Azure offers the *Premium Plan* [33], which allow users to have their code pre-warmed on a specified number of instances; Apache OpenWhisk [34] also includes the possibility for users to pre-warm a given number of containers. There are other research proposals for mitigating the cold start problem, for example the prebaking functions proposed in [35] that implements a mechanism that restores snapshots of previously created functions processes, or the reinforcement learning approach proposed in [36], which analyzes some factors, such as function CPU utilization, to determine the function-invocation patterns and reduce the function cold start frequency by preparing the function instances in advance.

The extension of the serverless computing paradigm to edge computing platforms has been also addressed in some research works. For example, an analysis of the suitability of serverless model for implementing services in edge computing platforms handling IoT data is achieved in [37]. In this work, the performance of the cold and warm start modes offered by OpenFaas is compared, for different IoT use cases, and authors conclude that, if the application tolerates latencies of few seconds, the cold start paradigm can be suitable, with the consequent saving of computing resources which can be of paramount importance in edge computing environments. Another interesting analysis is achieved in [38], where authors examine the main advantages of bringing serverless to the edge, and identify the main obstacles for this accomplishment, such as long latencies caused by cold start; the adaptation from a cost-efficiency design to a performance-oriented design; the unsuitability of serverless platforms for dealing with

continuous workloads; or the lack of support for distributed networking, among others. Other works focus on proposing different frameworks for serverless function deployment on edge platforms, for example, in [12], authors propose a distributed architecture made of self-organizing edge platforms able to collaborate in the allocation of resources and provisioning of serverless functions. Each platform has access to a pool of virtualized resources, which are used to provide the FaaS functionality to latency-sensitive and data-intensive applications. An extension of this work is presented in [39], where they present a new framework, PAPS (Partitioning, Allocation, Placement, and Scaling), for the efficient, automated, and scalable management of large-scale edge topologies. Another interesting approach is presented in [40], where authors propose a WebAssembly-based framework for serverless execution at the edge. This technology provides an alternative method for running serverless applications at near-native speeds, while having a small memory footprint and optimized invocation time, so it is suitable for edge environments with limited resources. From the point of view of networking. An analysis of serverless edge computing from a networking perspective is presented in [41], where authors propose a network architecture and layered structure to meet the design principles required for a serverless edge computing network (heterogeneity, scalability, performance, and reliability), and they address the main technical challenges such as service deployment and lifecycle management, service discovery and resource awareness, service scheduling, or incentive mechanism design.

FaaS execution model and cold-start reduction mechanisms

FaaS execution model

When deploying a serverless application, one of the main challenges is to minimize the application latency by reducing the overall response time of every function invocation. This issue is especially relevant when working with latency sensitive applications. The function response time depends on several latency components, such as the initialization time of the resource assigned to the function invocation (usually, a container or a micro-VM, also known as function instance), which can be warm or cold, the execution time of the function code, and the network latency between the function instance (i.e. the FaaS platform) and the source and destination recipients of the input and output data (e.g., an end user, an IoT device, a storage system, a database, etc.) Formally, given a FaaS function *f*, the response time of the function invocation *j* which executed by the instance *R(j)* can be computed as follows:

$$RT_f(j) = Init_f(j) + Exec_f(j) + Net_f(j) \quad (1)$$

where:

$Init_f(j)$ is the initialization time (cold or warm) of the instance *R(j)* to be ready for the execution of function invocation *j*.

$Exec_f(j)$ is the execution time of the function invocation *j* in the instance *R(j)*.

$Net_f(j)$ is the network latency for transmitting the input and output data between the instance *R(j)* and the data recipient corresponding to the function invocation *j*.

Therefore, to minimize the latency of a serverless application, it is necessary to minimize the average response time of every FaaS function *f* of this application, which can be expressed as:

$$Avg\_RT_f = \frac{\sum_{j=1}^{N} RT_f(j)}{N} \quad (2)$$

where N is the overall number of invocations of function *f* during the serverless application lifecycle.

The execution time depends on the computing capacity of the function instance, the computational nature of the function and the input parameters. These issues are out of the scope of this work. On the other hand, the network latency depends on the proximity between the FaaS platform where the function runs and the source/destination recipient of input/output function data. This point will be analyzed in a subsequent section by extending the serverless model to the edge. Finally, the initialization time of a function instance involves several stages, as shown in Fig. 1. First, there is some event that triggers the function call; then, the FaaS platform checks if there is a warm instance with the appropriate execution environment ready to run this function; in this case (warm start), the function handler is invoked and the function code executed; if there is not any ready warm instance (cold start), a new one is booted and allocated, the function code is downloaded to the instance and the corresponding execution environment is initialized. Finally, the function handler is invoked, and the function code executed.

The cold start delay will depend on many varied factors [9], such as the size of the code package (in general, the larger the code size, the longer the delay), the memory size of the instantiated resource (the more memory, the shorter the delay), or the runtime (usually scripting languages, like Python, Ruby or Javascript, perform better in startup time than compiled runtimes, like Java, NET, or C#). According to some recent empirical analysis [42, 43], cold start delays can vary from a few hundred

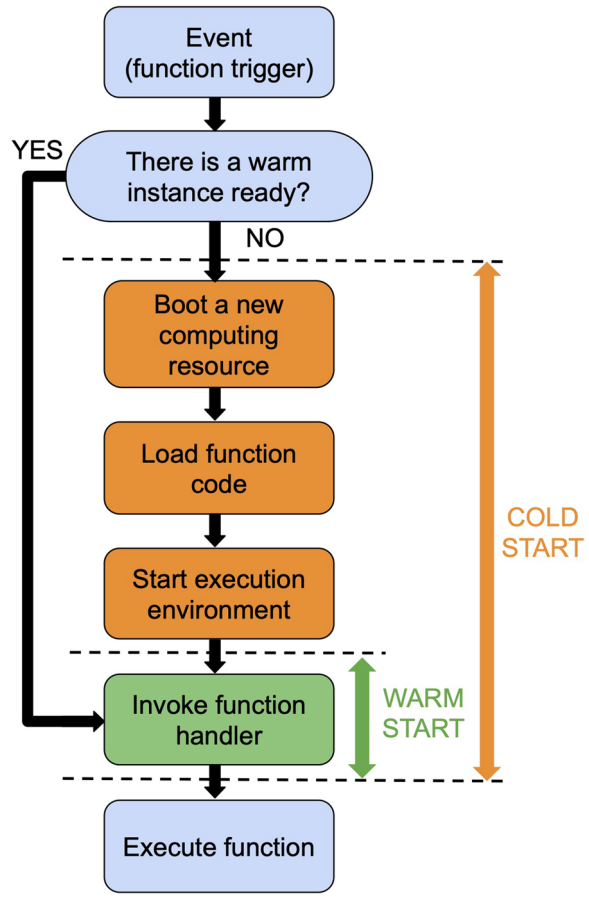

Fig. 1 FaaS execution stages

milliseconds to a few seconds, while warm start delays are usually about a few tens of milliseconds. There are different techniques and proposals for reducing cold start delay in serverless environments, as shown in section *State of the art*, however the two most common techniques implemented in existing FaaS platforms are instance pre-warming and instance reusing, that will be analyzed in detail later in this section.

Another important challenge of a FaaS platform is to minimize the total instance usage time to respond to the different invocations of a given function. Function instances in a FaaS platform can be in five different states, as shown in Fig. 2: Cold-start, Warm-start, Busy, Idle, and Terminated. When a new instance is spun up to execute a recently invoked function, it goes first to the Cold-start state. When the instance starts the execution of the function code, it goes to the Busy state. When a busy instance finishes the execution of the code, depending on whether the instance reuse mechanism is disabled or enabled, it can be shut down (Terminated state), or it can remain as a warm inactive instance (Idle state). When a warm instance in the Idle state is selected to execute a new invocation of the function it goes to the Warm-start state and then to the Busy state. Otherwise, when a warm instance in the Idle state is not used for a while (keep-alive interval), it is automatically shut down (Terminated state). Note that, when instance reuse mechanism is enabled, a given instance can execute several invocations of the function and go over the Busy, Idle, and Warm-start states multiple times.

Therefore, given an instance r that executes one or more invocations of a FaaS function f of a serverless application, the usage time, $UT_f(r)$, of this instance can be computed as follows:

$$UT_f(r) = ColdStart_f(r) + WarmStart_f(r) + Busy_f(r) + Idle_f(r) + Shutdown_f(r) \quad (3)$$

where:

$ColdStart_f(r)$ is the cold start time of the instance r.

$WarmStart_f(r)$ is the total warm start time of the instance r (it may include several invocations of the function).

$Busy_f(r)$ is the total busy time of the instance r (it may include several invocations of the function).

$Idle_f(r)$ is the total idle time of the instance r.

$Shutdown_f(r)$ is the shutdown time of the instance r.

Therefore, the total instance usage time of a FaaS function f is given by:

$$Total\_UT_f = \sum_{r=1}^{M} UT_f(r) \quad (4)$$

where M is the total number of instances used for executing the different invocations of function f during the serverless application lifecycle.

We can also compute the total start-up time, the total busy time, and the total idle time for all the instances used in the execution of a FaaS function f, as follows:

$$TotalStart_f = \sum_{r=1}^{M} (ColdStart_f(r) + WarmStart_f(r)) \quad (5)$$

$$TotalBusy_f = \sum_{r=1}^{M} Busy_f(r) \quad (6)$$

$$TotalIdle_f = \sum_{r=1}^{M} Idle_f(r) \quad (7)$$

$$TotalShutdown_f = \sum_{r=1}^{M} Shutdown_f(r) \quad (8)$$

To optimize the total instance usage time, and therefore the cost of the infrastructure, it is essential to minimize the total idle time of the different instances used. Pre-warming and reusing techniques used to minimize the cold-start problem can increment this idle time

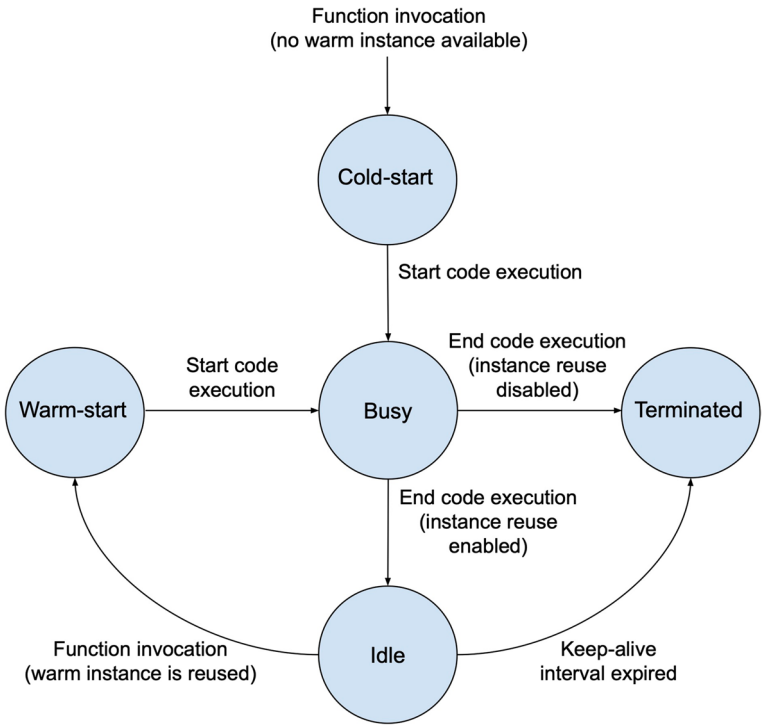

Fig. 2 Life-cycle state diagram of a function instance

and hence incur an extra expense of resources. Thus, it is important to adjust these techniques to reach a good trade-off between response time and resource consumption.

Pre-warming

Pre-warming techniques consist of starting in advance a given number of function instances (e.g., containers or micro-VMs) that stay always alive during the serverless application lifecycle and can run different invocations of the same function (several instance pre-warming mechanisms are discussed in section *State of the art*). Obviously, these pre-warming techniques can alleviate the cold start problem, however, it is important to be aware of the function workload profile (i.e., the number of simultaneous invocations of the function over the time), to choose the correct number of pre-warmed instances. If this number is insufficient, we will incur a deterioration of the average response time. On the other hand, if the number of pre-warmed instances is too high, we will cause an extra expense of resources, and consequently a higher cost without any significant reduction on the average execution time.

To probe this, we have conducted a real experiment in AWS Lambda by running a simple NodeJS-based function that takes 100 ms to execute. We launched 100 invocations of the function in a period of one second, and we used the *provisioned concurrency* feature of AWS Lambda to pre-warm various instances. Figure 3 displays the results of this experiment, showing the average response time, and the total number of instances used (both pre-warmed and on-demand) for different numbers of pre-warmed instances (between 0 and 50). As we can observe, between 0 and 30 pre-warmed instances, the average response time exhibits an important reduction, and the total number of instances is also improved. However, if we use a provisioned concurrency higher than 30, the average response time is not improved, incurring an unnecessary increase in resource consumption.

Instance reusing

Another complementary technique to reduce cold-start time is the reuse of function instances. When a FaaS function ends its execution, instead of shutting down the instance, it can be kept alive for a given interval, so that it can be reused as a warm instance to execute a new invocation of the same function (several instance reusing mechanisms are discussed in section *State of the art*). Obviously, this mechanism can alleviate the cold start problem, but at the expense of keeping a series of instances idle, thus causing extra consumption of resources on the FaaS platform. In most commercial serverless platforms, the cost of these extra resources is assumed by the provider and not charged to the user.

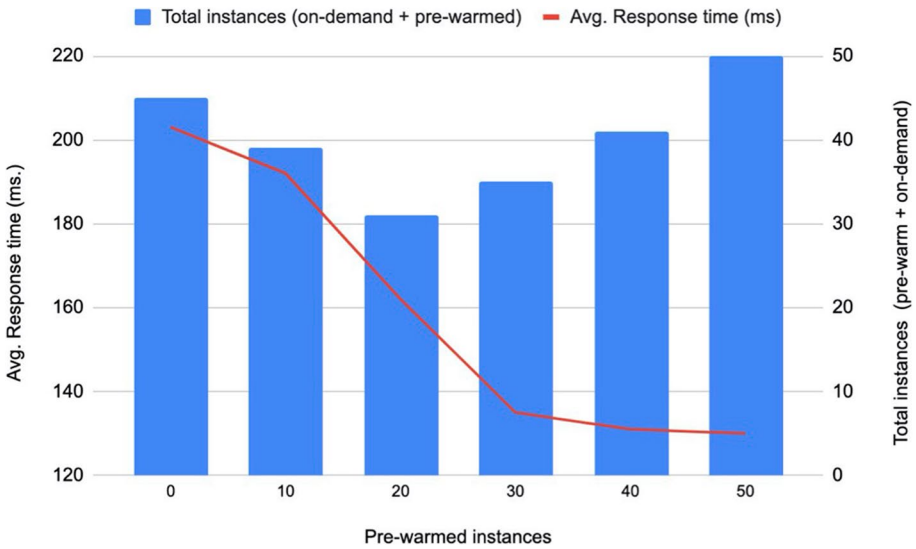

Fig. 3 Average response time and resource consumption for different numbers of pre-warmed instances in AWS Lambda

However, if resources are limited (e.g., in the case of edge micro-data centers), it is important to choose an appropriate keep alive interval that minimizes the average response time, while reducing the expense of resources. The optimal keep alive interval will depend on several factors, such as the execution time of the function code, the cold start delay, and the invocation frequency of the function. However, to the best of our knowledge, there does not exist any study about the optimal selection of this parameter and its impact on the response time and the resource consumption in a serverless environment. In this work (see section Results) we will analyze this impact.

Extending serverless platforms to two-tier edge/cloud environments

Most FaaS platforms are implemented by large cloud infrastructure providers with enough resources to support a large number of users deploying FaaS functions. Although the number of simultaneous executions for a specific function is often limited by the platform (for example, in AWS Lambda this parameter, called unserved concurrency, is limited to 1,000 instances per function), usually this limit can be negotiated. However, in order to support applications with low-latency requirements, such as real-time, IoT, or stream data analytics applications, these large centralized FaaS platforms exhibit an important drawback regarding the network latency between the devices where data is generated and the serverless computing infrastructure where this data is processed.

A solution to this problem is to move the FaaS platform to the edge [8, 35, 38] by means of serverless edge infrastructures that provide computational capacity in close proximity to data generation sources. There are some commercial solutions that fit this model, such as AWS IoT Greengrass or Azure IoT Edge. The main challenge of these solutions, when compared to cloud FaaS platforms, is that these edge platforms are usually resource constrained. To overcome this problem, in this work we propose a two-tier edge-cloud FaaS platform, as shown in Fig. 4, where instances used to execute the different invocations of a function can be provisioned either by the edge infrastructure or by the cloud site, depending on the resource availability at the edge infrastructure, and according to several placement policies. This provision model is like those used in hybrid clouds [44, 45] where the on-premises cloud infrastructure (usually a private cloud) can be complemented with remote resources from one or more public clouds to provide extra capacity to satisfy peak demand periods.

This edge-cloud serverless model supports several geographically distributed edge FaaS platforms to satisfy the demands of users/devices in different locations. Every edge FaaS platform can manage a limited local resource pool and an almost unlimited remote resource pool located on the cloud provider. The edge platform manages the function invocations of nearby users, so that instances to execute these functions can be deployed in the local edge infrastructure or in a remote cloud, according to the selected placement policy.

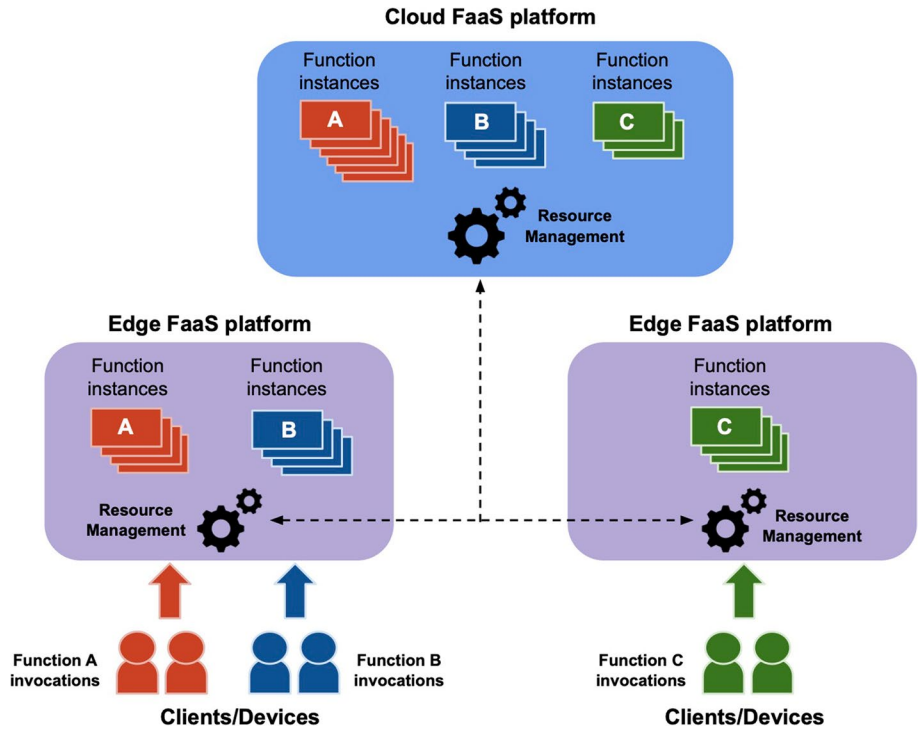

Fig. 4 Two-tier edge-cloud FaaS platform

In this work, we propose and evaluate two different placement policies:

- *Edge-first* policy*:* Try to execute the function in the edge platform first if enough resources are available. Otherwise, execute the function in the remote cloud. In both cases, try to reuse a warm instance first, if available.
- *Warm-first* policy: Try to execute the function where there is some warm instance available for this function first. If there are warm instances available in both locations, first use the edge instance to execute the function. If there are no warm instances, either on the edge or in the cloud, first try to deploy a new instance on the edge platform, if enough resources are available. Otherwise, deploy a new instance in the remote cloud.

The goal of the edge-first policy is to reduce the network latency first, and then the instance initialization time, while the goal of the warm-first policy is to reduce the instance initialization time first, and then the network latency. Table 1 summarizes the behavior of both policies when a new function invocation is executed, assuming sufficient resources in both cloud and edge to start a new cold instance if required. The placement policy determines whether the function should run on a warm or cold instance, either on the cloud or on the edge infrastructure. As we can see, the primary distinction between the two policies lies in the second scenario. In

Table 1 Behavior of the placement policies

Warm instance available at the edge	Warm instance available at the cloud	Edge-first policy placement decision	Warm-first policy placement decision
No	No	Use an edge cold instance	Use an edge cold instance
No	Yes	Use an edge cold instance	Use a cloud warm instance
Yes	Yes/No	Use an edge warm instance	Use an edge warm instance

the edge-first policy, an edge cold instance is utilized to execute the function, whereas the warm-first policy opts for a cloud warm instance.

Experimental environment

Use case description

The data analytics use case used for experimental purposes in this work is based on a deep learning model for image classification. In particular, we use a pre-trained model of the Inception-v3 [46] convolutional neural network [13] for assisting in image analysis and object detection, which uses the ImageNet Large Visual Recognition Challenge [47] as image classification training dataset. This Inception-v3 model can classify entire images into 1,000 different classes [48] with a 3.36% error rate. It is programmed in Python and is based on the TensorFlow-Slim image classification model library. According to its authors [13], the computational cost of Inception is much lower than other neural networks-based models for image classification, which makes it feasible to utilize in data analytics scenarios where a huge amount of different images is needed to be processed at a reasonable cost, or scenarios where memory or computational capacity is limited.

The experimental scenario we propose, see Fig. 5, is made up of several geographically distributed cameras that take pictures/images following a given pattern (e.g., images generated at fixed intervals, at random intervals, following a burst pattern, etc.) and send them for classification to a serverless infrastructure. Every time the FaaS platform receives a new image, it triggers the invocation of the FaaS function that runs the Inception-v3 model for image classification. The communication between the device and the FaaS platform can be done using an API gateway (e.g., a REST or a HTTP API), or using some intermediate data store (e.g., a bucket in Amazon S3).

The three major requirements for this scenario are: (i) to be highly scalable to support an increasing number of cameras from various locations (hundreds or even thousands); (ii) to offer the lowest possible response time for each function invocation; and (iii) to minimize the number of instances used to execute the different function invocations in order to reduce both the cost for the end users, and the resource expenses at the infrastructure provider.

The FaaSim simulator

The experimental results presented in this work are mostly based on simulations. Using a simulator offers several advantages for our research purposes compared to a real platform. First, it enables large-scale, multiple experiments to be run with significant savings in execution time and infrastructure costs. Second, a simulator allows to easily tune different platform parameters (e.g., number of pre-warmed instances, instance reuse interval, instance limits, etc.), even those that are not available on real platforms. Third, it enables the analysis of new resource management mechanisms or resource allocation policies in FaaS platforms. Finally, it allows us to extend the classical cloud-based Faas platforms to edge-based platforms.

For this purpose, we have developed a trace-driven simulator, called FaaSim, which reproduces the behavior of a FaaS platform executing a single FaaS function. FaaSim uses as input a list of events ordered by arrival time, and each event triggers a new invocation of the FaaS function. The simulator uses two different pools of resources, a pool of warm instances, and a pool of allocated instances. When a new function invocation is triggered, the system first looks for a free instance in the warm pool, if none is available, then a new (cold) instance is provisioned. In both cases, the selected instance passes to the pool of allocated instances. When the execution of

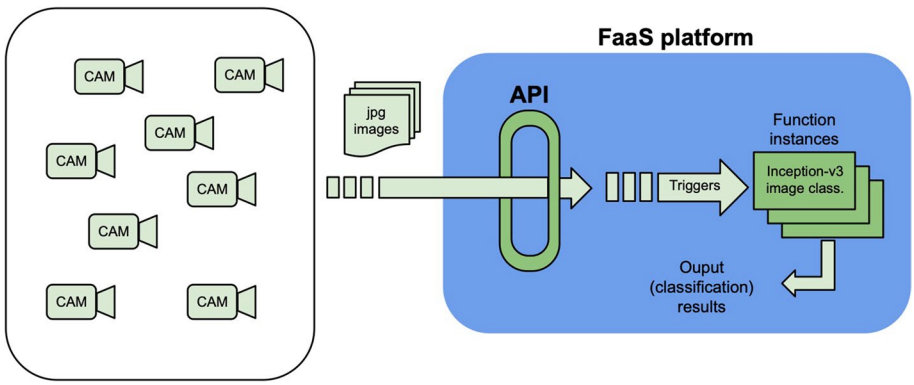

Fig. 5 Experimental scenario

a function finishes, if the instance reusing mechanism is disabled, the allocated instance is shut down. Otherwise, the instance is moved to the warm pool for a specific keep-alive interval, so if the instance is not reused during this interval, it is shut down.

FaaSim can be configured using the following parameters:

- *Simulation interval:* this parameter specifies the duration of the simulation time interval. By default, it will be equal to the arrival time of the last event in the input event list.
- *Pre-warmed instances:* this is the number of pre-warmed instances we use for the simulation (similar to the provisioned concurrency in AWS Lambda). If this parameter is equal to zero, then no pre-warmed instances are used.
- *Keep-alive interval:* this is to implement the reusing mechanism in the simulation. When the execution of a function ends, the allocated instance is kept warm for this interval. If this parameter is equal to zero, then the reusing mechanism is not used.
- *Maximum number of instances:* this parameter limits the number of simultaneous instances that can be used for executing the function (like unreserved concurrency in AWS Lambda).
- *Cold start time:* time that a cold instance takes to be ready to start the execution.
- *Warm start time:* time that a warm instance takes to be ready to start the execution.
- *Execution time:* time that the allocated instance takes to execute the function.
- *Network latency:* time it takes the input data to be transferred from the generating device to the FaaS platform (input bucket), and the output data to be sent back to the output data recipient.

The last four times are experimentally obtained from a real environment (AWS Lambda, as shown in the next sub-section). For each simulation run these values are randomly generated following a standard normal distribution.

Once the simulation ends, FaaSim provides the following output:

- Response time per every function invocation (Eq. (1))
- Average response time (Eq. (2))
- Usage time per resource (Eq. (3))
- Total resource usage time (Eq. (4))
- Total resource start-up time (Eq. (5))
- Total resource busy time (Eq. (6))
- Total resource idle time (Eq. (7))
- Total resource shutdown time (Eq. (8))
- Average number of instances used per second
- Total number of cold starts

FaaSim parameters tuning and validation

To adjust the input time parameters of the simulator, we have measured the cold, warm and execution times for the image classification application in a real FaaS platform. In particular, the platform chosen is the AWS Lambda serverless compute service and the demo bundle provided by Amazon [49] for the Inception-v3 convolutional neural network model. This deployment uses AWS S3 to trigger the invocation of the Lambda function every time the user uploads an image file in a given S3 input bucket.

To feed this experiment, we emulate a single camera generating one image per second, for a period of 100 s. We use 100 JPG image files selected from Kaggle Animals-10 dataset [50] as inputs, with sizes between 16 and 183 KB and an average size of 80 KB. These image files are sent to the AWS S3 input data bucket, at a rate of one file per second. We run the experiment five times, with enough spacing between different runs to ensure that, at the beginning of each run, there are no warm instances kept alive from the previous one in the AWS Lambda platform. Instance pre-warming (*provisioned concurrency*) was not used in this experiment.

We use the AWS X-Ray monitoring tool to analyze the traces of the different runs of the serverless application in the AWS Lambda platform. Figure 6 shows, for one of the runs of the experiment, the duration (i.e., the response time) for the 100 function invocations obtained with AWS X-Ray. These response times do not include the network delays between the data origin (the camera) and the AWS S3 bucket. As we can observe, the first 9 invocations exhibit much higher response time (about 8 s) than the remaining invocations (about 2 s). This is because of the first invocations are using cold instances, while the rest are using warm instances, thanks to the instance reusing mechanism implemented in AWS Lambda.

From the data provided by AWS X-Ray we can obtain an estimation of the average values of the response time, the execution time, the cold start time, the warm start time, and the overall number of instances used for each of the five runs of this experiment, as shown in Table 2. The last column shows the average value (AVG) of the five runs along with the relative standard deviation (RSD).

To validate our FaaSim simulator, we tuned it with the average and standard deviation values for the execution time, the cold start time, and the warm start time, shown in the last column of Table 2. Then we performed several runs of FaaSim using a workload similar to the real AWS Lambda experiment described

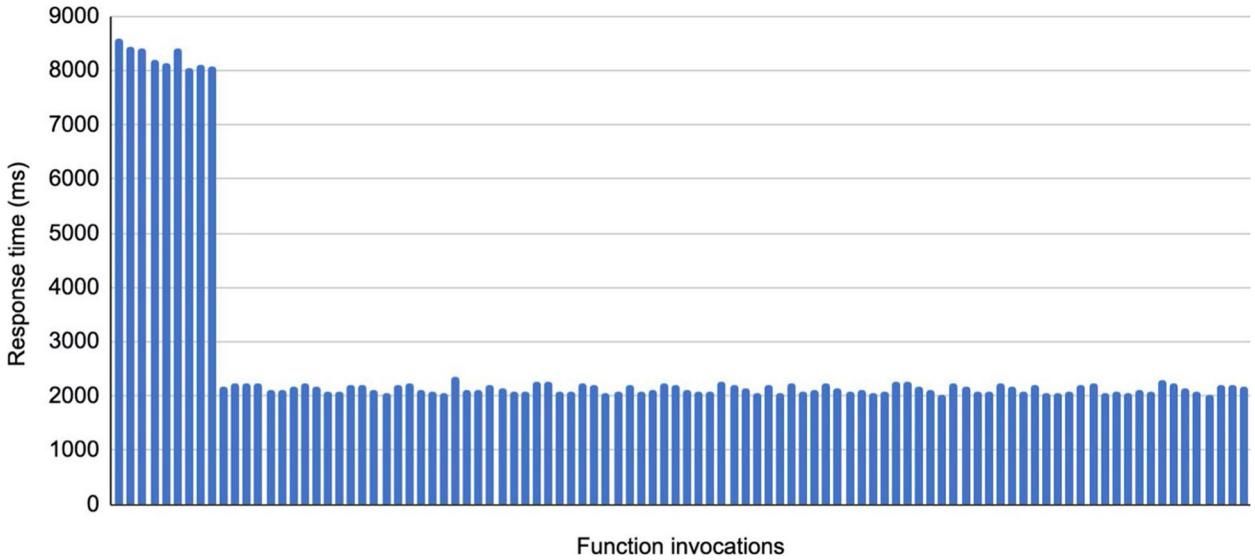

Fig. 6 Response time of the different function invocations for the run #1 of the AWS Lambda experiment

Table 2 Results of running the use case in AWS lambda

	Run #1	Run #2	Run #3	Run #4	Run #5	AVG (± RSD)
Response time (ms)	2,708.4	2,674.3	2,729,3	2,669.4	2,659.7	2,688.2 (± 1.1%)
Cold start time (ms)	6,190,8	6,072.0	6,115.1	6,042.0	6,028.0	6,089.5 (± 1.1%)
Warm start time (ms)	68.7	68.5	75.2	72.4	69.2	70.8 (± 4.1%)
Execution time (ms)	2,077.5	2,037.1	2,069.6	2,043.1	2,037.6	2,053.0 (± 0.9%)
No. instances used	9	9	9	9	9	9

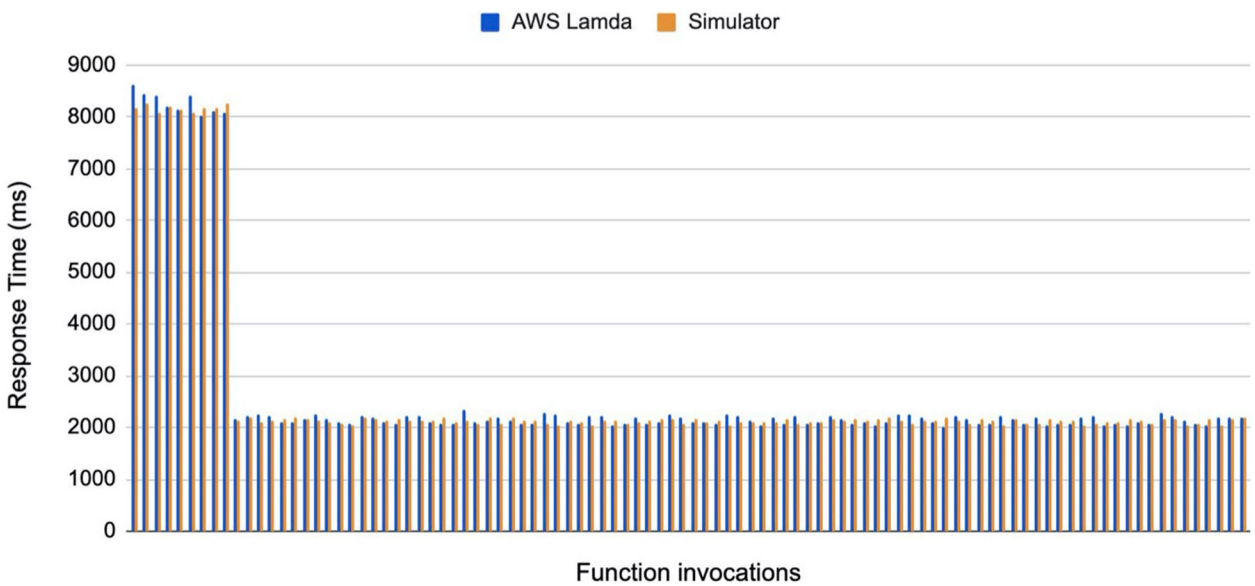

Fig. 7 Comparison of response time of the different function invocations for run #1 of the AWS Lambda experiment and the simulator

Fig. 8 Average response time obtained for the five runs of the AWS Lambda experiment and five runs of the simulator

above. We also activated the instance reusing mechanisms in the simulator, with a keep-alive interval equal to the overall simulation interval. Figure 7 shows the response time of the different function invocations for run #1 of the AWS Lambda experiment and one run of the simulator. We can observe that the profile of both runs is remarkably similar, and the number of instances used is the same (9 instances). Figure 8 shows the average response time of the serverless function for the 5 runs of the AWS Lambda experiment compared to 5 runs of the simulator. On average, the simulator gets a response time of 2,600 ms versus 2,688 ms for real experiments, with the same number of instances used. We can conclude that the results obtained with FaaSim closely reproduce the results obtained in a real environment, which proves the validity of the simulator. Observe, however, that the response times obtained by the simulator in the different runs show a more constant pattern than in the case of AWS Lambda. This is because, although standard deviations have been introduced in the simulator model, the data used, being statistically generated, present smoother fluctuations with respect to the mean than in real-world scenarios.

The previous experiments used for the validation of the FaaSim simulator do not consider the network latencies between the data source and the FaaS platform. However, in the next section we will demonstrate that these delays are very relevant when extending the FaaS platform to an edge environment. For this purpose, we have also performed several experiments on AWS Lambda to estimate the network latency in two different scenarios: a cloud scenario where a device sends the image files to an AWS Lambda function deployed in a remote Amazon region; and an edge scenario where the device sends the files from the same region where the Lambda function is deployed (for example, sending the files from an AWS EC2 instance located in the same Amazon region). In particular, we sent three files of different sizes—maximum (183 KB), minimum (16 KB), and average size (80 KB) -, and we repeated the experiment 10 times. Table 3 shows the results of these experiments.

Table 3 File transfer times for cloud and edge scenarios

	File transfer time for cloud scenario		File transfer time for edge scenario	
	AVG (ms)	RSD	AVG (ms)	RSD
Max. file size	1,510	2.9%	110	20.9%
Min. file size	422	5.0%	62	16.1%
Avg. file size	851	4.9%	73	15.1%

Results

As we stated before, most FaaS platforms use pre-warming and reusing techniques used to minimize the cold-start problem. However, these mechanisms should be adapted to the particular application profile, in order to reduce the response time of the serverless application, while minimizing the number of instances used during execution. It is not easy to obtain a mathematical model that fits with different platforms and application profiles, so here we propose to use FaaSim simulations for the optimal tuning of these mechanisms. First, we study the case of a centralized cloud FaaS platform, and then

we extend the study to a two-tier distributed edge/cloud FaaS platform.

The experiments in this section are based on the experimental scenario shown in Fig. 5, where several cameras send images to the FaaS platform, and a FaaS function classifies them using the Inception-3 convolutional neural network algorithm.

All the experiments use a testbed consisting of 100 cameras sending a total of 100 image files per camera, so that the serverless application must process 10,000 images. To emulate different input data streams, we consider three different image transmission patterns, represented in Fig. 9. In the first pattern (Fig. 9a), each camera transmits at a rate of one image per second, so the FaaS platform receives a continuous data stream of 100 images per second. In the second pattern, each camera transmits at a rate of one image every 30 seconds, resulting in a burst pattern (Fig. 9b) with 100 images per burst. Assuming that all cameras are in sync with an accuracy of about 1 second, and that transmission delays are, on average, about 1 second, we assume that the length of each burst of images is about 2 seconds, separated by 30 seconds between consecutive bursts. The third pattern considered is a random pattern (Fig. 9c) where each camera generates images at random intervals (e.g., triggered by motion detection), which can vary between 0 and a maximum of 1 minute between consecutive images. Table 4 summarizes the parameters of these three patterns. Note that the simulation interval varies for each transmission pattern because the number of images to be processed is constant (10,000 images in all cases), but the arrival rate of images differs for each pattern.

Centralized cloud FaaS platforms

First, we analyze the instance reuse mechanism in order to adjust the optimal keep-alive interval for the application profile we are considering. If the keep-alive interval is too short, there will be more cold starts, so the average

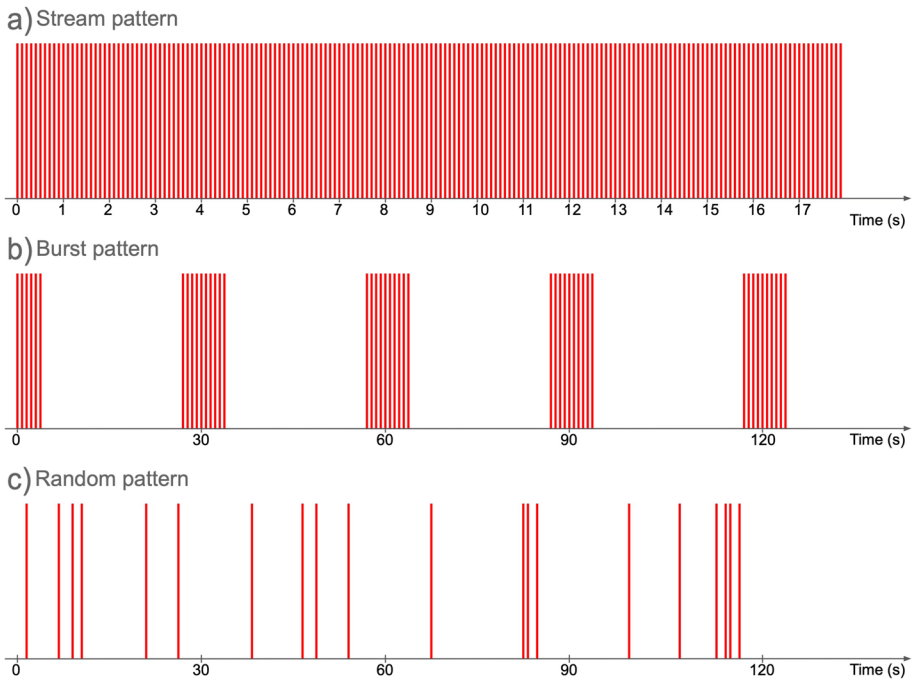

Fig. 9 Different image transmission patterns (the graphs represent the traffic from 10 cameras)

Table 4 Summary of image transmission patterns

Transmission pattern	Image transmission rate	No. Cams	Images per cam	Total images	Simul. interval
Stream pattern	1 image per second per cam	100	100	10 K	100 s
Burst pattern	1 image every 30 s per cam	100	100	10 K	2970 s
Random pattern	1 image at random intervals (between 0–1 min.) per cam	100	100	10 K	3600 s

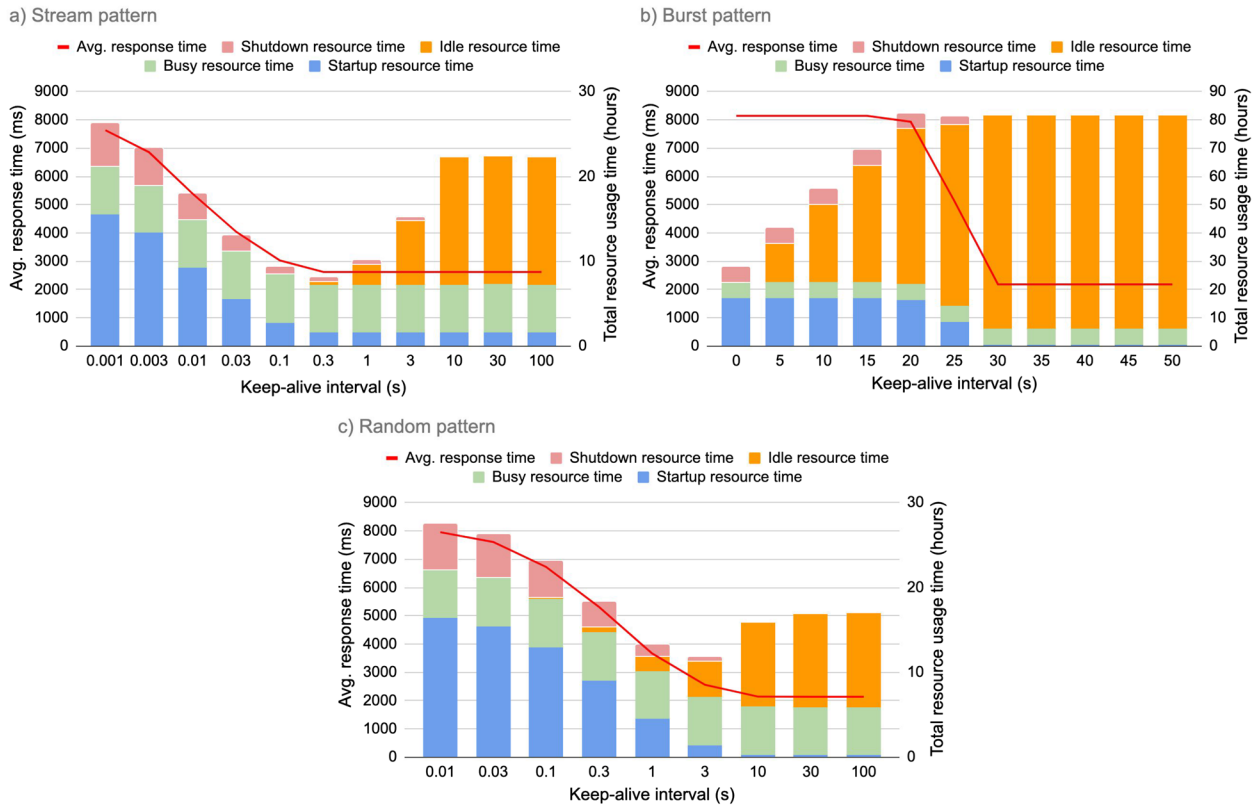

Fig. 10 Influence of keep-alive interval over average response time and resource usage time (charts a) and c) uses a logarithmic scale in the horizontal axis)

response time will increase. On the other hand, if the keep-alive interval is too long, the system can incur an extra expense of resources (instances), with the consequent increment of the infrastructure cost. To determine the optimal value of this parameter, we have analyzed three scenarios corresponding to the three transmission patterns explained above: stream, burst, and random patterns. For each scenario, we have achieved different simulations with different values of the keep-alive interval, and we have analyzed both the average response time and the total resource usage time, broken down in the four main components: startup, busy, idle and shutdown resource time. The FaaSim simulator is tuned with the input parameters (cold start, warm start, and execution time) obtained from the real AWS Lambda experiments shown in the previous section (see last column of Table 2), without considering the network delays, and with no pre-warmed instances (these two factors will be analyzed later). Additionally, the shutdown time for each terminated instance has been set to 2 seconds, which is the maximum duration of the entire shutdown phase for runtime environments in AWS Lambda [51].

Graphs in Fig. 10 show the average response time and total resource usage results for the three scenarios (stream, burst, and random transmission patterns, respectively). As we can observe, when the keep-alive interval is 0 (no instance reuse) or very low, the average response time is noticeably higher due to the high number of cold starts, which increase the total resource start-up time. The average response time decreases when the keep-alive interval increases, and the minimum average response time value is reached for keep-alive intervals of about 0.3 s in the first scenario (stream pattern), 30 s in the second scenario (burst pattern), and 10 s in the last scenario (random pattern), achieving an overall reduction in the average response time of 65.8%, 73.2% and 73.2%, respectively. Using longer keep-alive intervals does not improve the average response time, but in some cases (especially in the first scenario), it can increase the total resource usage time, due to the larger number of warm instances that remain idle.

Next, we analyze the instance pre-warming mechanism in order to adjust the optimal number of pre-warmed instances for the application profile we are considering. As we explained before, instance pre-warming can reduce the number of cold starts, and hence the average response time. However, if the number of pre-warmed instances is higher than needed, it will result in a useless

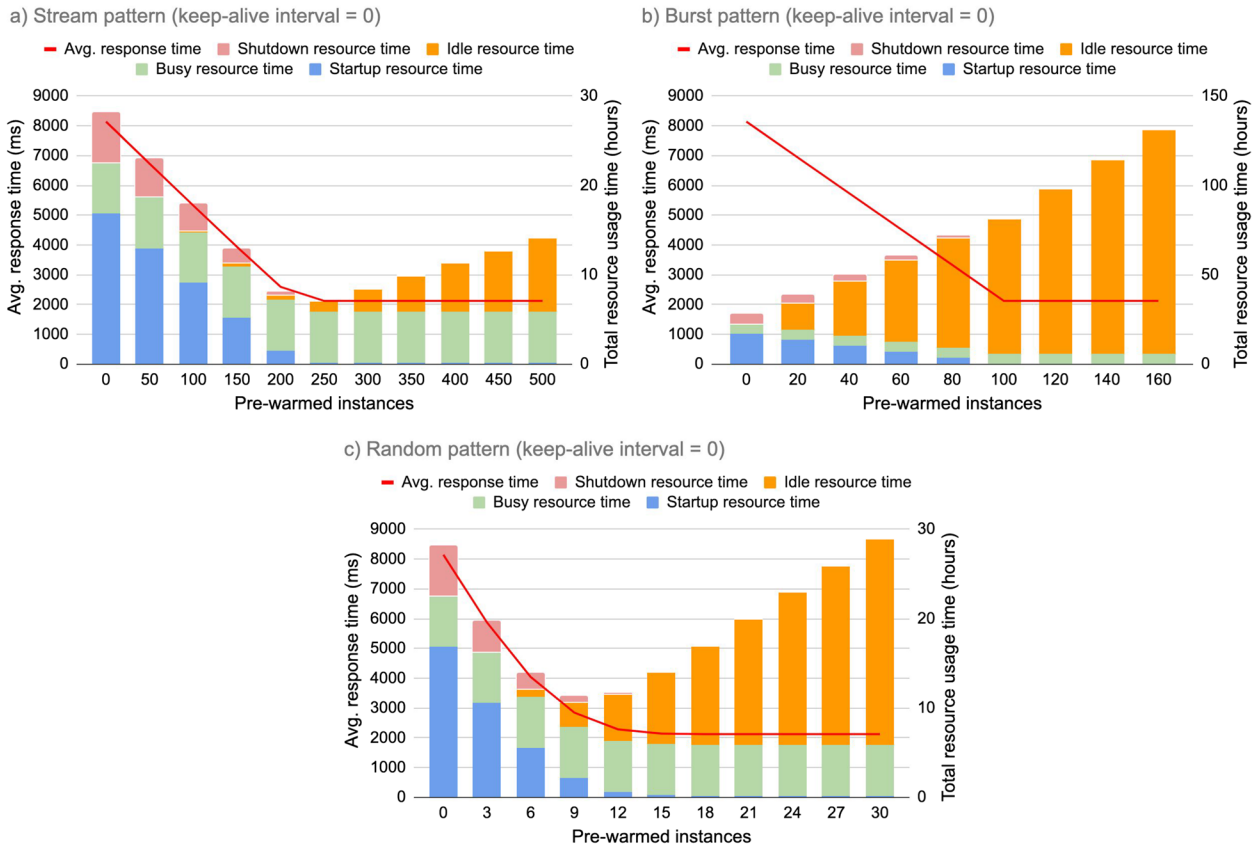

Fig. 11 Influence of pre-warmed instances (without instance reuse) over average response time and resource usage time

extra expense of resources, and consequently a higher cost. It is important to note that pre-warmed instances are assumed to be booted and ready before starting the execution of the function, so they do not contribute to the total start-up resource time. The instance pre-warming mechanism can be used alone or combined with the instance reuse mechanism, so we will analyze both cases. As in the previous experiments, we consider three different scenarios corresponding to the stream, burst, and random transmission patterns.

Graphs in Fig. 11 show the average response time and total resource usage results, for the three scenarios, for different numbers of pre-warmed instances and a keep-alive interval of 0 (no instance reuse). As we can observe, the optimal number of pre-warmed instances that minimizes the average response time is quite different depending on the transmission pattern: about 250 instances for the stream pattern, about 100 instances for the burst pattern, and about 14 instances for the random pattern. The overall reduction in the average response time is 73.9% in all the three cases. Increasing the number of pre-warmed instances over these values does not improve the average response time, but results in a useless increase in the total resource usage time.

Similarly, graphs in Fig. 12 show the average response time and total resource usage results, for the three scenarios, for different numbers of pre-warmed instances but now combined with the instance reuse mechanism. The keep-alive interval is set, in each case, to its optimal value according to the results shown in Fig. 10 (i.e., 0.3 s for the stream pattern, 30 s for the burst pattern, and 10 s for the random pattern). As we can observe, the instance reuse mechanism already reduces by itself the average response time significantly, so the improvement obtained with the pre-warming mechanism is not so noticeable (18.9%, 2.8%, and 0.9% reduction in the average response time for the stream pattern, the burst pattern, and the random pattern, respectively).

Distributed edge/cloud FaaS platforms

In this subsection we analyze the results of deploying our experimental scenario in a two-tier distributed edge/cloud FaaS platform. For this purpose, we include the network latencies in the computation of the response time. In the following experiments, we assume that this

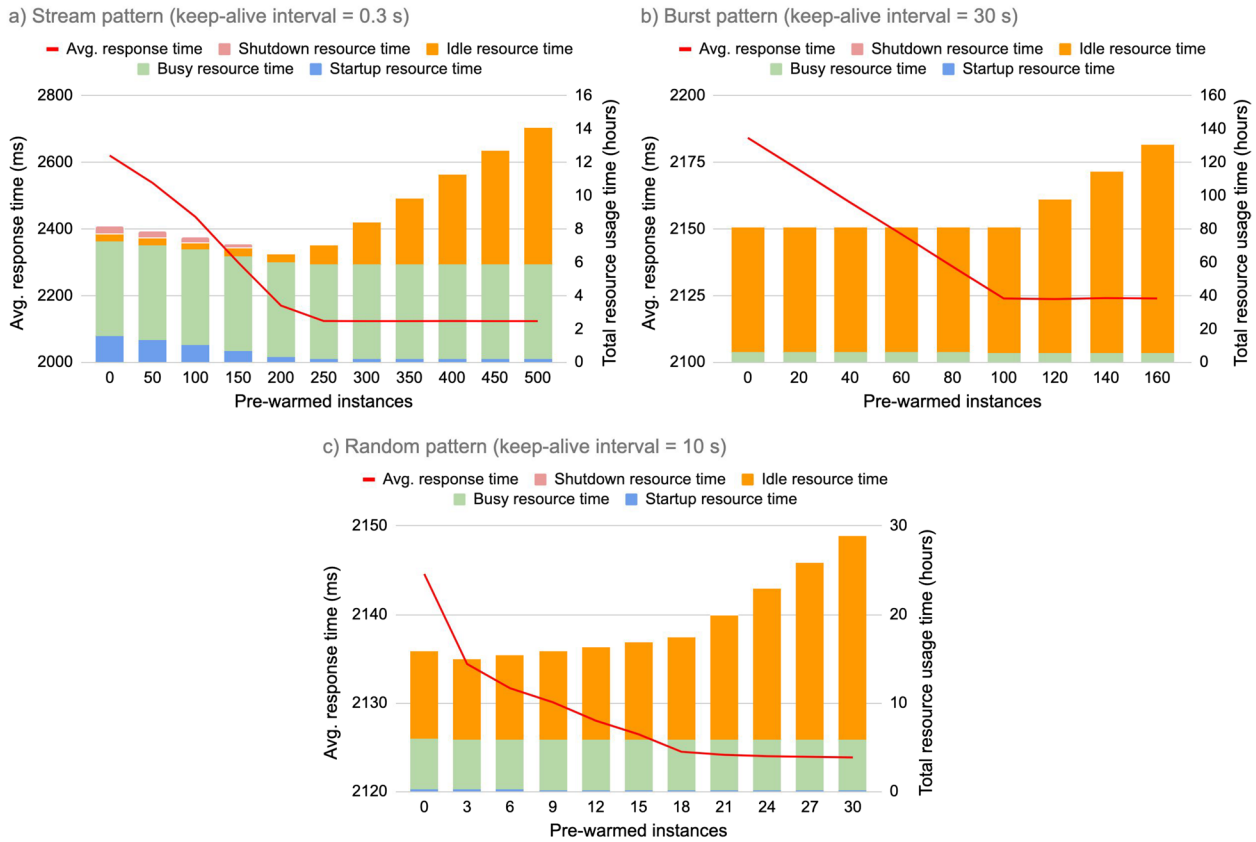

Fig. 12 Influence of pre-warmed instances (combined with instance reuse) over average response time and resource usage time

latency is equal to the transfer time of an image file of average size, see Table 3. The testbed for these experiments consists of 100 cameras distributed in five different geographical regions, with 20 cameras per region, as shown in Fig. 13. The edge FaaS platforms, one per region, are responsible for processing the image files generated by the cameras belonging to that region. The different function invocations on a given region can be allocated to a local edge instance, or to a remote cloud instance, depending on the instance availability on the edge FaaS platform, and the placement policy used, implemented by the scheduler. We will also consider the same three image transmission patterns (stream, burst, and random patterns) summarized in Table 4.

First, we will study the instance reuse mechanism by analyzing the average response time results obtained for different keep-alive intervals when deploying our serverless application in this distributed edge FaaS platform, compared to the results obtained in a centralized cloud FaaS platform, shown in Fig. 14. In these graphs, the average response time has been broken down into the three components as defined in equation (1): the initialization time (cold or warm) of the instances in the FaaS platform, the execution time of the different function invocations, and the network latency. In the case of the edge FaaS platform, we assume that all the function invocations of a given region are executed in the local edge platform, using the edge-first allocation policy. The comparison of different allocation policies will be discussed later in this section. In analyzing these results, we note that for certain patterns (stream and random), the cloud FaaS platform achieves better response times than the edge FaaS platform when keep-alive intervals are short. This is due to the inability of the edge platform to share warm instances across different regions, resulting in more frequent cold starts and, consequently, higher instance initialization times. However, with longer keep-alive intervals, the edge FaaS platform consistently outperforms the cloud FaaS platform due to its lower network latency.

Next, we analyze the influence of the pre-warming technique over the average response time, comparing the results obtained in a distributed edge FaaS platform and a centralized cloud FaaS platform. Figure 15 shows the results for different numbers of pre-warmed instances without instance reuse (keep-alive interval = 0, in graphs 15.a, 15.b, and 15.c) and combined with the instance reuse mechanism (keep-alive interval > 0, in graphs 15.d,

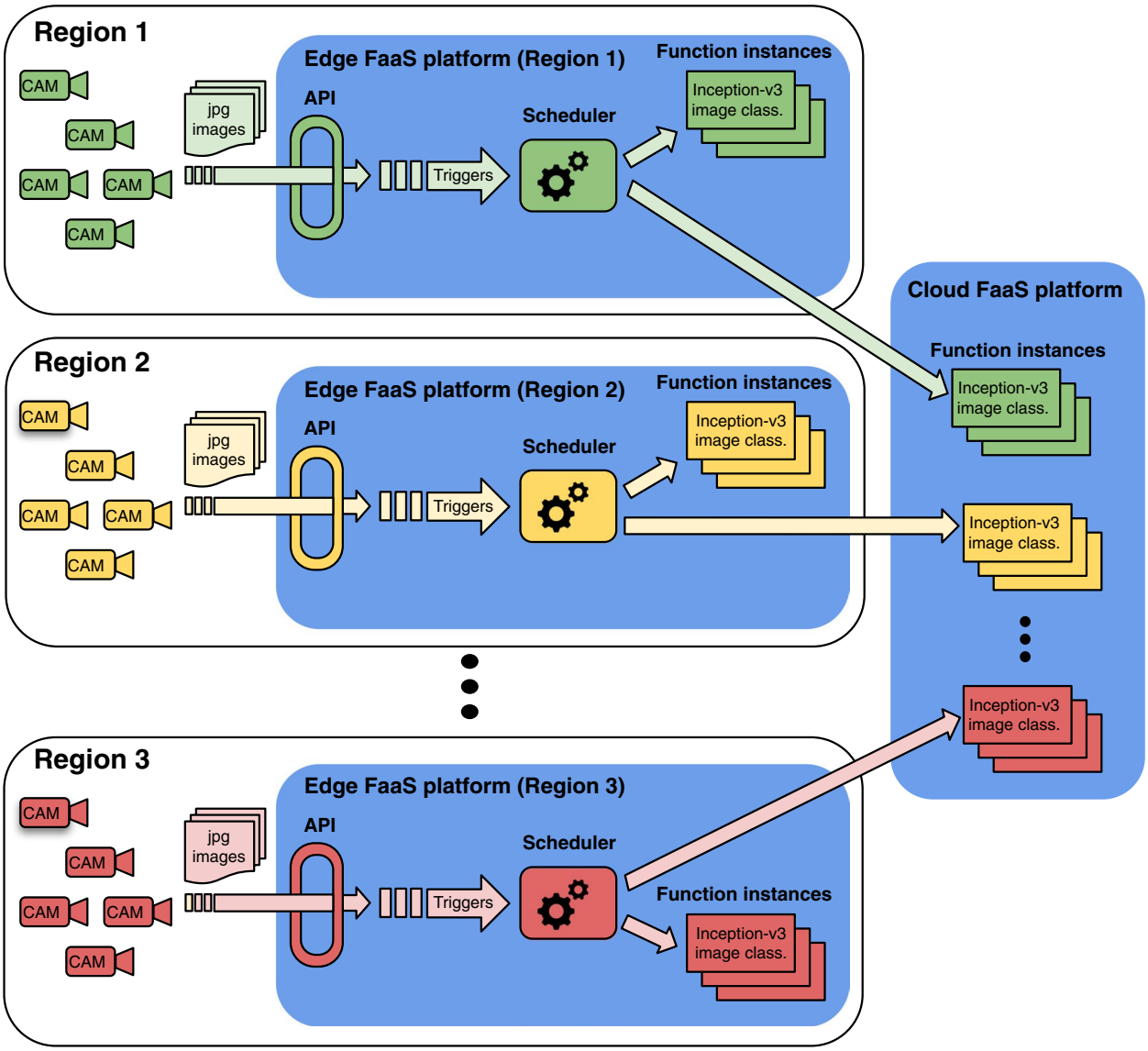

Fig. 13 Two tier distributed edge/cloud FaaS platform experimental scenario

15.e, and 15.f). As in the previous case, in the edge FaaS platform deployment we assume that all the function invocations of a given region are executed in the local edge platform, using the edge-first allocation policy. It is also important to notice that, in the case of the distributed edge platform, the pre-warmed instances shown in Fig. 15 are evenly distributed among the FaaS platforms in each of the regions. As we can observe, in most cases, the edge FaaS platform provides faster response times than the cloud platform, because of the lower network latencies supported at the edge.

Finally, we analyze the two allocation policies proposed for the two-tier distributed edge/cloud FaaS platform. These policies are the edge-first policy, which prioritizes executing functions on the edge platform when resources are sufficient, and the warm-first policy, which prioritizes executing functions on warm instances if they are available. For these experiments we assume that we have a different number of pre-warmed instances in the cloud platform, not in the edge platform. So, the edge-first policy will always execute all the function invocations in the corresponding edge platform, but the warm-first policy can execute some function invocations in the cloud platform if there are warm instances available. We analyze two different scenarios, shown in Fig. 16. In the

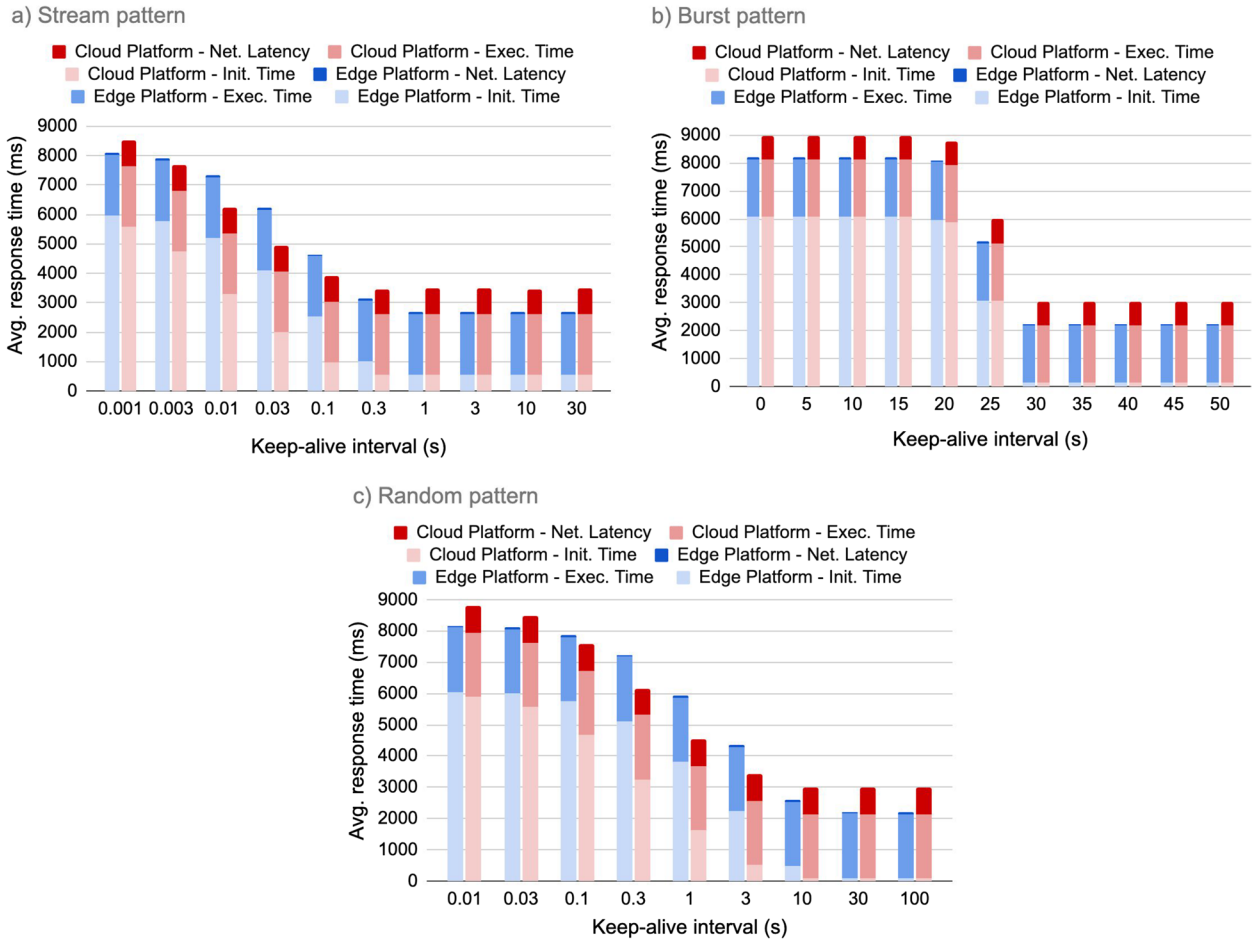

Fig. 14 Comparison of edge and cloud FaaS platforms: influence of keep-alive interval over average response time (charts a) and c) uses a logarithmic scale in the horizontal axis)

first scenario the instance reuse mechanism is not used (keep-alive interval = 0, in graphs 16.a, 16.b, and 16.c), and in the second one, the cloud instance pre-warming is combined with the instance reuse mechanism (keep-alive interval > 0, in graphs 16.d, 16.e, and 16.f).

As we can observe, if no instance reuse mechanism is used (graphs 16.a, 16.b, and 16.c) the warm-first policy always obtains better response time results than edge-first policy. This is because, with no instance reuse, the number of cold starts is remarkably high and the initialization time is the predominant term in the response time calculation, so as the number of pre-warmed instances in the cloud grows, the warm-first policy significantly reduces this initialization with the consequent improvement of the average response time. On the other hand, when the instance reuse mechanism is used, the number of cold starts is significantly lower, and then the network latency becomes a factor with greater weight in the calculation of the response time. Consequently, in two scenarios (burst and random patterns, as shown in graphs 16.e and 16.f), the edge-first policy delivers better average response time results than the warm-first policy. This is because, under the warm-first policy, when the number of pre-warmed cloud resources increases, more invocations are processed in the cloud platform, resulting in higher network latency. In the case of a stream pattern with keep-alive mechanism (graph 16.d) both policies yield very similar results in terms of average response time, with the warm-first policy being slightly better in almost all cases.

Conclusions and future work

The aim of this study is to investigate methods for reducing latency in serverless applications. Two primary approaches to mitigate the cold-start problem, instance reuse and instance pre-warming, were analyzed to assess their effect on application response time and resource consumption. The results show that instance reuse mechanisms can significantly reduce

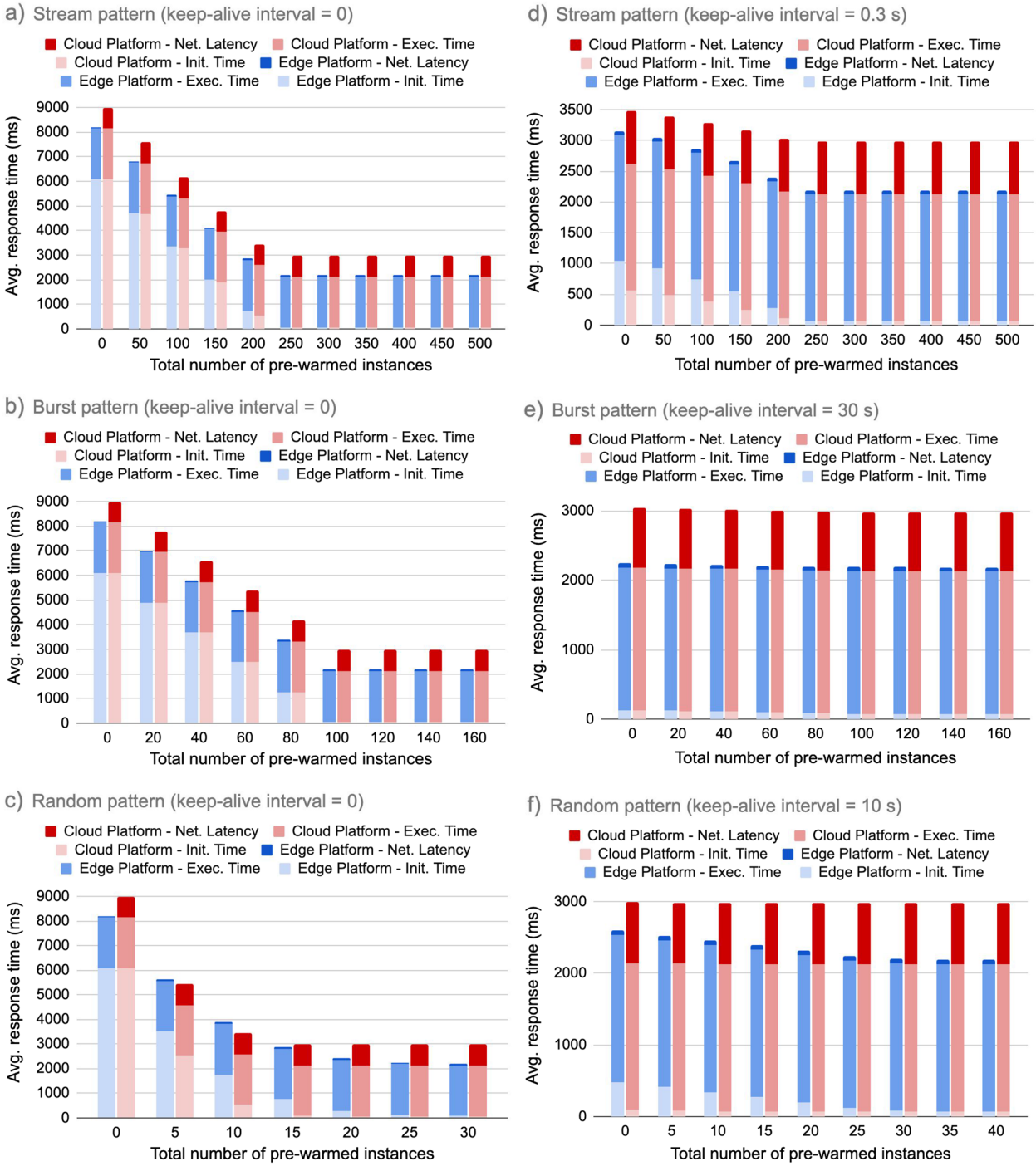

Fig. 15 Comparison of edge and cloud FaaS platforms: influence of pre-warming over average response time without instance reuse (left) and combined with instance reuse (right)

cold-starts and improve response time. However, selecting an appropriate keep-alive period specific to each application and input data profile is crucial to achieve optimal response time without increasing resource expenses. Similarly, while instance pre-warming can also reduce response time, exceeding a certain threshold of pre-warmed instances for a given application or input data profile may not improve response time further and may unnecessarily increase resource consumption. By combining both mechanisms

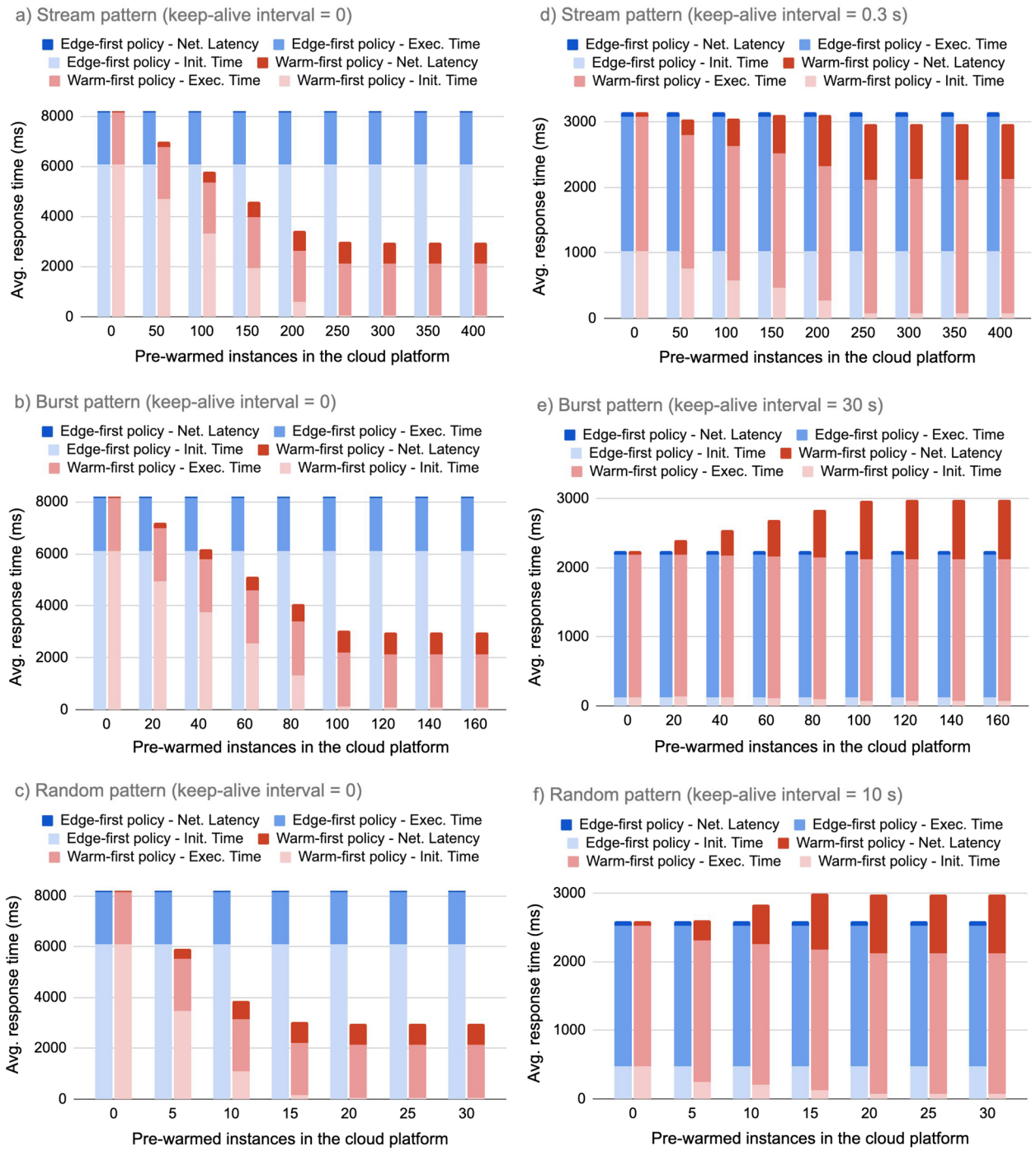

Fig. 16 Comparison of edge-first and warm-first policies without instance reuse (left) and with instance reuse (rigth)

and selecting the appropriate parameters, such as keep-alive interval and number of pre-warmed instances, an optimal response time can be achieved with limited resource usage. However, determining the suitable values for these parameters can be challenging as it is highly dependent on the application profile and workload pattern. Users may need to perform several iterations to adjust these values to find the optimal settings for their specific use case. Simulation tools, such as the one presented in this study, can be valuable in this context to adjust optimal parameters, such as

keep-alive interval, number of pre-warmed instances, and allocation policy, for each application or input data profile before deployment.

Expanding the serverless platform to the edge can also improve response time by reducing network latency. Two allocation policies, the edge-first and warm-first policies, have been analyzed to reduce either network latency or instance initialization time first. The warm-first policy is generally more effective without instance reuse mechanisms; however, with instance reuse enabled, the winning allocation policy depends on the input data pattern.

As future work, we plan to integrate all the analyzed mechanisms into an actual FaaS platform. We also plan to explore and implement advanced instance pre-warming and reuse mechanisms, such as predictive mechanisms that use historical application profiles and machine-learning based forecasting to predict optimal deployment parameters.

Abbreviations

API	Application Programming Interface
AVG	Average
AWS	Amazon Web Services
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
ISP	Internet Service Provider
REST	Representational State Transfer
RSD	Relative Standard Deviation
UT	Usage Time
VM	Virtual Machine

Acknowledgements
Not applicable.

Authors' contributions
RMV conceived the study, coordinated the research, defined the FaaS execution model, developed the FaaSim simulator, conducted the experimental section, and drafted the manuscript. EH participated in the definition of the FaaS execution model and the use case and helped to refine the manuscript. RSM and IML participated in the definition of the experimental scenarios and helped to refine the manuscript. All authors read and approved the final manuscript.

Authors' information
Not applicable.

Funding
This work was supported by Ministerio de Ciencia, Innovación y Universidades under the research project RTI2018-096465-B-I00 (EdgeCloud), by Comunidad de Madrid under the research program P2018/TCS4499 (EdgeData), and by the European Union under grant agreements 880412 (ONEedge) and 101092711 (SovereignEdge.Cognit).

Availability of data and materials
An experimental prototype of the FaaSim simulator, and the input files of the three data patterns used in this work (stream, burst and random patterns) are available at https://github.com/rmorvoz/FaaSim.
Images used in the experiments were selected from Kaggle Animals-10 dataset, available at https://www.kaggle.com/datasets/alessiocorrado99/animals10.

Declarations

Competing interests
The authors declare no competing interests.

Received: 16 March 2022 Accepted: 10 July 2023
Published online: 19 July 2023

References

1. Yu T, Wang X (2020) Real-time data analytics in internet of things systems. In: Tian Y, Levy D (eds) Handbook of real-time computing. Springer, Singapore. https://doi.org/10.1007/978-981-4585-87-3_38-1
2. Atitallah S, Driss M, Boulila W, Ghézala H (2020) Leveraging deep learning and IoT big data analytics to support the smart cities development: review and future directions. Comput Sci Rev 38. https://doi.org/10.1016/j.cosrev.2020.100303
3. Ellis B (2014) Real-time analytics: techniques to analyze and visualize streaming data. Indianapolis: Wiley; 2014.
4. Nastic S et al (2017) A serverless real-time data analytics platform for edge computing. IEEE Internet Comput 21(4):64–71. https://doi.org/10.1109/MIC.2017.2911430
5. López P et al. (2019) ServerMix: tradeoffs and challenges of serverless data analytics. arXiv: 1907.11465v1. doi:https://doi.org/10.48550/arXiv.1907.11465
6. Castro P, Ishakian V, Muthusamy V, Slominski A (2019) The rise of serverless computing. Comm of the ACM 62(12):44–54. https://doi.org/10.1145/3368454
7. Jonas E et al. (2019) Cloud programming simplified: a Berkeley view on serverless computing. arXiv:1902.03383v1. https://doi.org/10.48550/arXiv.1902.03383
8. Baldini I et al (2017) Serverless computing: current trends and open problems. In: Chaudhary S, Somani G, Buyya R (eds) Research advances in cloud computing. Springer, Singapore. https://doi.org/10.1007/978-981-10-5026-8_1
9. Manner J, Endreß M, Heckel T, Wirtz G (2018) Cold Start Influencing Factors in Function as a Service. In: IEEE/ACM Int. Conf. on Utility and Cloud Computing Companion 2018 (UCC Companion), 181–188. doi:https://doi.org/10.1109/UCC-Companion.2018.00054
10. Bajpai A (2021) Serverless Cold Starts - Mitigation Techniques. https://www.techtalksbyanvita.com/post/serverless-cold-starts-can-we-mitigate-these. Accessed 1 Mar 2023
11. Raza A, Matta I, Akhtar N, Kalavri V, Isahagian V (2021) SoK: Function-As-A-Service: From An Application Developer's Perspective. J Syst Res. 1(1). https://doi.org/10.5070/SR31154815.
12. Baresi L and Filgueira Mendonça D (2019) Towards a Serverless Platform for Edge Computing. In: IEEE Int. Conf. on Fog Computing 2019 (ICFC'19), 1–10. https://doi.org/10.1109/ICFC.2019.00008
13. Szegedy C, Vanhoucke V, Ioffe S, Shlens J and Wojna Z (2016) Rethinking the Inception Architecture for Computer Vision. In: IEEE Conf. on Computer Vision and Pattern Recognition 2016, 2818–2826. https://doi.org/10.1109/CVPR.2016.308
14. Moreno-Vozmediano R. https://github.com/rmorvoz/FaaSim. Accessed 1 Mar 2023
15. Amazon Web Services, Inc. https://aws.amazon.com/lambda. Accessed 1 Mar 2023
16. Google. https://cloud.google.com/functions. Accessed 1 Mar 2023
17. Microsoft. https://azure.microsoft.com/services/functions/. Accessed 1 Mar 2023
18. Palade A, Kazmi A, Clarke S (2019) An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In: IEEE World Congress on Services 2019, 206-211. https://doi.org/10.1109/SERVICES.2019.00057
19. Li J, Kulkarni S, Ramakrishnan K, Li D (2019) Understanding Open Source Serverless Platforms: Design Considerations and Performance. In: 5th Int. Workshop on Serverless Computing 2019 (WOSC '19), 37–42. https://doi.org/10.1145/3366623.3368139
20. Mohanty S, Premsankar G, di Francesco M (2018) An Evaluation of Open Source Serverless Computing Frameworks. In: IEEE Int. Conf. on

Cloud Computing Technology and Science 2018 (CloudCom), 115–120. doi:https://doi.org/10.1109/CloudCom2018.2018.00033
21. Ellis A. https://www.openfaas.com/. Accessed 1 Mar 2023
22. The Apache Software Foundation. https://openwhisk.apache.org/. Accessed 1 Mar 2023
23. Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau A, Arpaci-Dusseau R (2016) Serverless computation with openLambda. In: 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16), USENIX Association, 33–39. https://dl.acm.org/doi/https://doi.org/10.5555/3027041.3027047
24. Kubeless (VMware Archive). https://github.com/vmware-archive/kubeless. Accessed 1 Mar 2023
25. The Kubernetes Authors. https://kubernetes.io/. Accessed 1 Mar 2023
26. The Knative Authors. https://knative.dev/. Accessed 1 Mar 2023
27. The Istio Authors. https://istio.io/. Accessed 1 Mar 2023
28. Díaz J. https://www.npmjs.com/package/serverless-plugin-warmup. Accessed 1 Mar 2023
29. Shahrad M et al (2020) Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In: USENIX Annual Technical Conference 2020, 205–128. https://doi.org/10.5555/3489146.3489160
30. Fuerst A, Sharma P (2021) FaasCache: keeping serverless computing alive with greedy-dual caching. In: 26th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems 2021(ASPLOS'21), 386–400. doi:https://doi.org/10.1145/3445814.3446757
31. Roy R, Patel T, Tiwari D (2022) IceBreaker: warming serverless functions better with heterogeneity. In: 27th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems 2022 (ASPLOS 2022), 753–767. doi:https://doi.org/10.1145/3503222.3507750
32. Amazon Web Services, Inc, "Configuring provisioned concurrency," https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html. Accessed 1 Mar 2023
33. Microsoft Azure, "Azure Functions Premium Plan", https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan. Accessed 1 Mar 2023
34. The Apache Software Foundation, "OpenWhisk Actions", https://apache.googlesource.com/openwhisk/+/HEAD/docs/actions.md. Accessed 1 Mar 2023
35. Silva P, Fireman D, Emmanuel Pereira T (2020) Prebaking Functions to Warm the Serverless Cold Start. In: 21st Int. Middleware Conference 2020, pp. 1–13. doi:https://doi.org/10.1145/3423211.3425682
36. Agarwal S, Rodriguez M, Buyya R (2021) A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. I: 21st International Symposium on Cluster, Cloud and Internet Computing 2021 (CCGrid), 797–803. doi:https://doi.org/10.1109/CCGrid51090.2021.00097
37. Benedetti P, Femminella M, Reali G, Steenhaut K (2021) Experimental Analysis of the Application of Serverless Computing to IoT Platforms. Sensors, 21(3). https://doi.org/10.3390/s21030928
38. Aslanpour M et al. (2021) Serverless Edge Computing: Vision and Challenges. In: Australasian Computer Science Week Multiconference 2021 (ACSW '21), 1–10. https://doi.org/10.1145/3437378.3444367
39. Baresi L, Quattrocchi G (2021) PAPS: A Serverless Platform for Edge Computing Infrastructures. Frontiers in Sustainable Cities 3. https://doi.org/10.3389/frsc.2021.690660
40. Gadepalli P, Peach G, Cherkasova L, Aitken R, Parmer G (2019) Challenges and Opportunities for Efficient Serverless Computing at the Edge. In: 38th Symposium on Reliable Distributed Systems 2019 (SRDS), 261–266. https://doi.org/10.1109/SRDS47363.2019.00036
41. Xie Q, Tang S, Qiao H, Zhu F, Yu R, Huang T (2021) When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues. IEEE Wireless Commun 28(5):126–133. https://doi.org/10.1109/MWC.001.2000466
42. Malishev N (2019) AWS Lambda Cold Start Language Comparisons, 2019 edition. https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244. Accessed 1 Mar 2023
43. Roberts M (2020) Analyzing Cold Start latency of AWS Lambda. https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda. Accessed 1 Mar 2023
44. Moreno-Vozmediano R, Montero R, Llorente I (2012) IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. Computer 45(12):65–72. https://doi.org/10.1109/MC.2012.76
45. Moreno-Vozmediano R et al (2016) BEACON: A Cloud Network Federation Framework. In: Celesti A, Leitner P (eds) Advances in Service-Oriented and Cloud Computing. ESOCC 2015. Communications in Computer and Information Science, vol 567. Springer, Cham. https://doi.org/10.1007/978-3-319-33313-7_25
46. Silberman N, Guadarrama S (2016) TensorFlow-Slim image classification model library. https://github.com/tensorflow/models/tree/master/research/slim. Accessed 1 Mar 2023
47. Russakovsky O et al (2015) ImageNet Large Scale Visual Recognition Challenge. Int J Comput Vision 115:211–252. https://doi.org/10.1007/s11263-015-0816-y
48. WekaDeeplearning4j (2019) IMAGENET 1000 Class List. https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/. Accessed 1 Mar 2023
49. Ivanovic B, Ivanovic Z (2017) How to Deploy Deep Learning Models with AWS Lambda and Tensorflow. https://aws.amazon.com/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow. Accessed 1 Mar 2023
50. Corrado A (2020) Kaggle Animals-10 dataset. https://www.kaggle.com/datasets/alessiocorrado99/animals10. Accessed 1 Mar 2023
51. Amazon Web Services, Inc, Lambda execution environment. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html. Accessed 1 Mar 2023

Publisher's Note