# A Programming Model and Middleware for High Throughput Serverless Computing Applications

Alfonso Pérez
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Valencia, Valencia
alpegon3@upv.es

Germán Moltó
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Valencia, Valencia

Miguel Caballer
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Valencia, Valencia

Amanda Calatrava
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Valencia, Valencia

## ABSTRACT

Serverless computing has introduced unprecedented levels of scalability and parallelism for the execution of High Throughput Computing tasks. This represents a challenge and an opportunity for different scientific workloads to be adapted to upcoming programming models that simplify the usage of such platforms. In this paper we introduce a serverless model for highly-parallel file-processing applications. We also describe a middleware implementation that supports the execution of customized execution environments based on Docker images on AWS Lambda, the leading serverless computing platform. Moreover, this middleware offers tools to manage the input/output of the serverless infrastructure and the creation of HTTP endpoints in a transparent way to the user. To test the programming model proposed and the middleware, this paper describes two case studies. The first one analyzes medical images with a high degree of parallelism. The second one presents an architecture to analyze video keyframes. The results from both case studies are analyzed and a cost analysis of the medical image architecture comparing different cloud options is carried out. The results show that the combination of a high-level programming model with the scalable capabilities of AWS Lambda makes it easy for end users to efficiently exploit serverless computing for the optimized and cost-effective execution of loosely-coupled tasks.

## CCS CONCEPTS

• **Architectures** → **Cloud computing**; • **Applied computing** → *Event-driven architectures*;

**ACM Reference Format:**
Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. 2019. A Programming Model and Middleware for High Throughput Serverless Computing Applications. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3297292

## 1 INTRODUCTION

Over the last years the offering made by large enterprises of renting computing, storage and network capacity on a pay-per-use basis has resulted in a tremendous revolution that democratized the access to large-scale enterprise-ready computing infrastructures without large upfront investments. The main public Cloud computing providers are Amazon Web Services [2], Microsoft Azure [27], and Google Cloud Platform [18]. On the other hand, Cloud Management Platforms such as OpenStack [30] and OpenNebula [29] enable system administrators to create on-premises Cloud infrastructures.

In parallel, the evolution of container-based technologies exemplified by Docker [9, 25], LXC [23] and rkt [7] introduced significant advantages with respect to Virtual Machines. There are on-premises Container Orchestration Platforms such as Swarm mode in Docker Engine [10], Nomad [19] or Kubernetes [22], and managed services provided by the leading public Cloud providers. Examples of the latter are Amazon ECS [5], Azure Container Service [26] and Google Container Engine [16]. The main drawback with these services is that they are typically oriented to advanced users, in order to deal with the capacity planning required to deploy the clusters in advance and optimize the allocation of resources to containers.

In the recent years, the term Serverless computing [6] has been coined to embrace an event-driven Functions-as-a-Service (FaaS) approach to computing with a fine-grained cost model. Pioneer services in this area, such as AWS Lambda [1], allow functions to be invoked in response to events such as uploading a file to a *bucket* in Amazon S3 (Simple Storage Service) or also in response to HTTP calls made to a predefined endpoint created with the AWS API Gateway service.

The programming model introduced by AWS Lambda can be effectively exploited for scientific applications and there are few examples in the literature using it for distributed computing such as Bulk Synchronous Processing (e.g. PyWren [21]), or fine-grained video processing (e.g. ExCamera [11]). It is important to note that current serverless platforms are typically focused on Functions-as-a-Service, where applications need to be redesigned as a set of event-triggered functions coded in a supported programming language. However, many applications cannot be easily redesigned as a set of functions. Indeed, the interface between the user and the serverless platform should not only be based on functions, which suffer from the inherent restrictions of the programming languages chosen. Instead, containers provide users with the ability to run virtually any kind of application without having to introduce changes. Supporting applications defined via container images in a serverless platform would allow the user to: i) easily bring their own applications which may already be packaged as a Docker image, ii) use applications that depend on libraries not available in the runtime environment of the functions, and iii) use programming languages not currently supported by the serverless provider.

To this aim, this paper introduces a High Throughput Computing Programming Model that allows to create highly-parallel event-driven file-processing serverless applications. The programming model is used in combination with a middleware (i.e. SCAR [32]) to simplify and automate the application deployment process and permit the users to execute customized runtime environments in the serverless platforms, thus bypassing some limitations imposed by the Cloud providers. This demonstrates how serverless computing can be effectively employed for many applications to achieve unprecedented scalability for loosely-coupled tasks with almost no configuration provided by the user side.

The remainder of the paper is structured as follows. First, section 2 describes the related work in the area. Next, section 3 introduces the programming model. Then, section 4 describes the updates done to the middleware that provides an implementation of the programming model. Section 5 describes two case studies to assess the usefulness of the proposed programming model and the middleware. Finally, sections 6 and 7 summarize the main achievements and point to future work, respectively.

## 2 RELATED WORK

The serverless computing model aims to revolutionize the design and development of modern scalable applications, allowing developers to run ephemeral, event-driven code without provisioning or managing servers. This new paradigm is experimenting an industry momentum around the cloud event abstraction [13]. In fact, over the last three years, several event-driven services such as AWS Lambda [1], offered by Amazon Web Services, Google Cloud Functions [17], Microsoft Azure Functions [28], and the open-source Apache OpenWhisk [12], have arisen. These services allow the users to take advantage of the improvements offered by this new computing model. The works presented by McGrath et al. [24] and Gannon [14] performed a review of these services, discussing the recent state of the art in this field.

Indeed, the serverless technology is starting to be employed in several scenarios. For example, in Web environments reduces the infrastructure costs more than 70% achieving a similar level of performance than traditional server-based solutions [35]. There are tools in the literature like Up [4], that facilitates the deployment of vanilla HTTP servers on serverless platforms, and developments like OpenLambda [20] an open-source platform for building web services applications using the serverless computing model. Another area where serverless computing is significantly being adopted is Big Data. Case studies of data analytics over serverless platforms, like [15], where the authors perform data processing with Spark over Apache OpenWhisk, are getting attention of researchers and developers. Some examples of recent works using serverless computing are open-source tools like Ooso [31], a Java library designed to execute MapReduce tasks based on Apache Hadoop and Spark on AWS Lambda, or enterprise solutions like Databricks Serverless [8], a serverless computing platform for complex data science and Apache Spark workloads. Moreover, projects like AWS Serverless Application Model (AWS SAM) [3] attempt to provide the means to define serverless functions for AWS Lambda.

A major requirement for writing serverless code, however, is to express the logic as functions that are instantiated to process a single event triggered by a service. The work by Baldini et al. [6] regarding the open problems of serverless computing identifies several unsolved challenges which include: i) the ability to run legacy code on serverless platforms, and ii) the lack of patterns for building serverless solutions.

In order to contribute to address these open issues, this paper describes a programming model designed specifically to create highly-parallel event-driven file-processing serverless applications for serverless architectures. This programming model, in combination with the SCAR framework, allows the user to run generic applications, even legacy ones, on serverless platforms.

## 3 PROGRAMMING MODEL

This section describes the proposed programming model used to create highly-parallel event-driven file-processing serverless architectures in combination with generic execution environments (i.e. containers). Containers are used to allow the users to create customized runtime environments, thus bypassing the provider limitations imposed in their execution environments. The infrastructure used to deploy and test the programming model and framework is the serverless computing platform AWS Lambda.

In addition, this programming model assumes that: i) the user wants to process a set of files that could be in a storage service or in a local machine; and ii) after the function execution, the output files will be transferred to a storage service (outside the space allocated to the lambda function). These assumptions are made because lambda functions are stateless by definition so the architecture making use of them must be designed stateless.

Figure 1 shows the proposed programming model. Notice that it allows users to select between different paths to process their files: 1) submit the file to process using an HTTP request through a previously defined and linked API Gateway; 2) upload a file to an S3 bucket that is linked with the lambda function or read the files from a non-linked S3 bucket. Both ways end up creating one event for each file that is going to be processed. Then, this event is sent to AWS Lambda and used to invoke a lambda function that processes it. All the approaches will end up storing the results in an S3 bucket, which in turn leads to the third approach presented to process files: 3) an S3 bucket, that could be connected to AWS Lambda again, and trigger the execution of more lambda functions automatically, thus effectively implementing serverless workflows. In the following subsections, the proposed approaches are described in more detail.

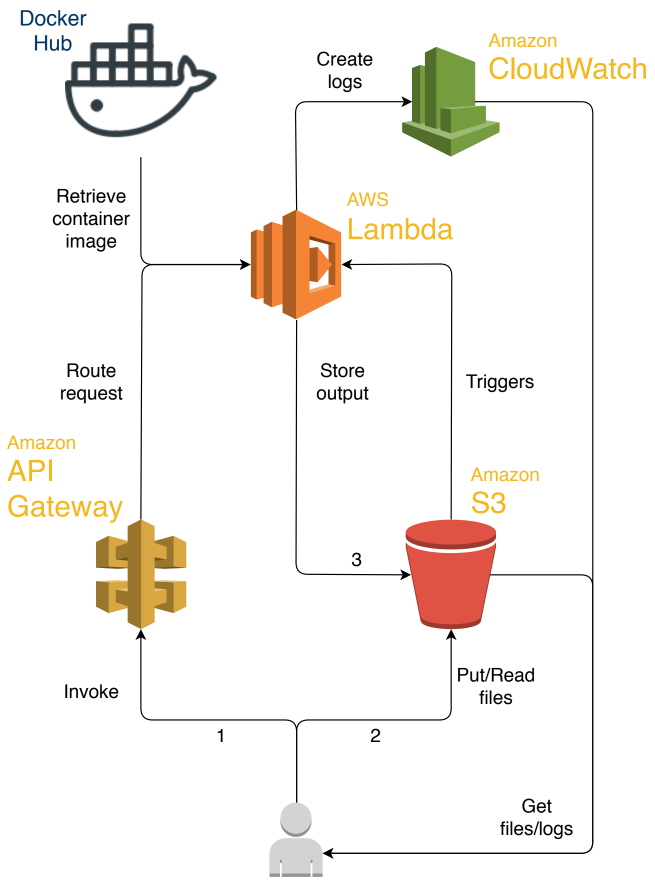

**Figure 1: Programming model for high throughput serverless computing applications. This approach allows the user to execute Docker containers as lambda functions, thus providing not supported environments or legacy applications with high throughput capabilities. Each number corresponds to one of the subsections in section 3.**

## 3.1 Process files using the API Gateway

The first approach, shown in Figure 1, allows the user to invoke a lambda function without having to use an external storage source. However, to be able to execute this approach the user needs to have an API Gateway defined and linked with the function that is going to be triggered (which can also be done automatically with SCAR). The HTTP request sent to the endpoint is routed to AWS Lambda and processed by the function, leaving the file ready to be used by the container executed inside the function. When invoking the function, the user can create a synchronous or an asynchronous invocation. A synchronous invocation tells the API Gateway to wait for the Lambda service response and to send back that response to the user (being the response the output created by the container execution). On the other hand, an asynchronous invocation returns immediately the control to the user, but it cannot ensure the correct invocation and execution of the function. The asynchronous invocation method can be useful to invoke almost simultaneously hundreds of functions, allowing the user to take advantage of the high scalability provided by AWS Lambda, which allows to execute up to 3000 concurrent function invocations.

## 3.2 File upload/read triggers Lambda Function

The second approach shown in Figure 1, uploading/reading files from an S3 bucket allows the user three possible subpaths to launch a lambda function by using the S3 buckets as event sources: i) the user uploads a file to an S3 bucket, which is the event source of a lambda function. When the upload finishes, the bucket sends and event to AWS Lambda with information about the created file. Then, the service invokes the lambda function in combination with the event information; ii) the user copies a file from a second available bucket to the bucket linked with the lambda function, this will cause, as in the first case, the trigger of an event from S3 to Lambda and the invocation of a lambda function. By using this path the user takes advantage of the high transfer rates between S3 buckets and can have all the files pre-uploaded in the AWS infrastructure, thus avoiding the need to upload a file each time a Lambda function needs to be invoked ; iii) the lambda function is invoked in parallel using an S3 bucket that does not need to be the source of events. To follow this path, the user needs to specify a bucket where the files to be processed are stored. Then, for each file found, an event is sent to AWS Lambda, so a lambda function is automatically invoked. This approach allows the user to take advantage of already existing data sets, as in the second path, and also reduces the invocation time between lambda functions by instantiating all of them in parallel.

After the bucket sends an event to AWS Lambda with information about the uploaded file, the Lambda service invokes the lambda function and passes it the event information. AWS Lambda does not copy the uploaded file into the ephemeral local storage allocated for the lambda function, instead, it only passes the event with the bucket and the file information (in a JSON-structured document). The proposed programming model and implementation automatically transfers the file to the lambda function using the event information. Moreover the files uploaded are not only available in the ephemeral local storage but are also made available into the container deployed inside the lambda function (i.e. accessible through environment variables). After the file has been automatically transferred to the lambda function, the script defined by the user when the function was created is executed inside the container specified also by the user. Once the script finishes, the container execution is terminated. If there are files in the output folder (i.e. available in the output path specified also in an environment variable), they are automatically transferred to the folder of the bucket specified by the user (the default value corresponds to the same bucket that started the execution).

## 3.3 Output files trigger new Lambda functions

The third approach presented in Figure 1 allows the user to define a chain of functions communicated by the events triggered by Amazon S3. The user defines the input/output buckets and folders for each of the lambda functions created. Therefore, the output folder defined by one function can be the input folder of another function. By using this approach, the user only has to define the functions with their respective input and output folders, and invoke the first one of the chain using one of the approaches defined above. The following functions are invoked by the respective events created by Amazon S3 upon the creation of the files. This approach paves the way to define data-oriented workflows, even supporting recursive function invocations.

## 4 FRAMEWORK UPDATE AND LIMITATIONS

To demonstrate the feasibility of the programming model presented, the SCAR middleware [32] has been updated to allow the deployment and management of the proposed architecture. In addition to the already available functionality of deploying containers as functions and create triggers between S3 buckets and Lambda functions, the SCAR client can now create REST API endpoints. This feature allows the user to call a lambda function without having to use the SCAR client and opens the way to an improved connection between microservices. Also, the S3 bucket link feature has been refined and now the user can create specific folders inside the buckets and reuse them among different functions.

The limitations of the programming model and the SCAR framework are mainly imposed by the Cloud provider that offers the service, in this case, AWS Lambda. The most restrictive limitations are the storage size and the function execution time. Regarding the storage limitations, AWS Lambda limits the disk storage of the functions to 512 MB. SCAR needs to store the container image and the function data in this folder, thereby imposing a serious limitation on the size of the containers that can be deployed. However, relying on minimalistic Linux distributions has allowed us to introduce support in SCAR for different container-based applications in different areas such as image and video processing (e.g. ImageMagick, FFmpeg), Cloud clients (e.g. AWS CLI), deep learning frameworks (e.g. Theano and Darknet) as well as code in virtually any programming language (e.g. Erlang, Elixir, Ruby and R)[1].

Furthermore the execution time is limited to 900 seconds or 29 seconds if you use the API Gateway with the synchronous invocation. If a function execution exceeds the time limit, AWS Lambda kills the invocation and returns an error to the user. Although it is expected that the thresholds of the current limitations will increase over time, right now those limits impose hard requirements to comply with and they must be taken into account when porting an application that uses the presented architecture.

## 5 CASE STUDIES

In this section, a demonstration of the feasibility of the proposed programming model is carried out. Two case studies and architectures are designed: 1) an architecture to process medical images and 2) an architecture to analyze images in video files. As stated previously, the AWS services will be used to deploy and test the architectures.

The common steps in both case studies involve the creation of the Docker images and the scripts needed to parse the received files. To create the Docker image, the user has to define a $Dockerfile$ that contains the application and all its dependencies. Other option would be using an already defined Docker image that meets the requirements of the application. After the Docker image is created and stored in Docker Hub, the script that manages the file processing and the application execution needs to be created. To create the script, the user must to take into account that: 1) Each lambda invocation processes exactly one input file, therefore the script needs to process only one file at a time; 2) The input file is retrieved automatically by the SCAR supervisor and is accessible from inside the container using the variable `$SCAR_INPUT_FILE`, so the script needs to process the input available at this path; 3) The output created by the container execution must be stored in the `$SCAR_OUTPUT_FOLDER` folder so it can be automatically copied back to the specified bucket by the SCAR supervisor.

Once the environment is defined (Docker image) and the script to be executed is created, the application architecture can be created using the SCAR CLI. The Dockerfiles and scripts used in both architectures are available in the SCAR repository in GitHub [1] and the Docker images are available in DockerCloud[2].

In the following sections 5.1 and 5.2, the architectures of the case studies are presented. Next in section 5.3 the results of both architectures are resumed and to finish in section 5.4 is carried out a cost analysis of the first case study.

### 5.1 Massive Medical Image Analysis Service

In this case study, an architecture that analyzes medical images (i.e. Magnetic Resonance Images - MRIs) on a serverless computing platform is deployed and tested. The architecture is based on the model proposed in section 3.2 in combination with the SCAR framework. Figure 2 describes the complete architecture and the data workflow during the execution of the experiment.

The core system used to analyze the images is presented in the paper by Torro et al. [34]. The system employs the Diffusion-Weighted Imaging method to extract meaningful information about the microscopic motions of water in human tissues. The application receives as input an MRI image to analyze, among other parameters. The application uses OpenMP to perform automated parallelization to take advantage of the number of cores available in the computing resource.

The following trace details the steps taken to carry out the experiment. This steps don't include the image and script definition:

(1) Through the SCAR client, the user creates the lambda function, specifying the Docker image and the script to be executed inside the container. To be able to store the results automatically, the user also needs to specify the S3 bucket used as output by the lambda function.
(2) Using the SCAR client, the user invokes the lambda function by specifying the S3 bucket and folder that is going to be used as event source. This is the last step that requires user intervention.
(3) The SCAR client automatically creates the required events (one for each file in the folder specified) and performs as many invocations of the lambda function as files are available. The invocation of the functions is done asynchronously so the execution benefits from the inherent parallelization capabilities of the infrastructure.
(4) The SCAR supervisor deployed inside the lambda function retrieves the input file from the S3 bucket and stores it in

[1]SCAR Use Cases and Examples - https://github.com/grycap/scar/tree/master/examples

[2]https://cloud.docker.com/u/grycap/

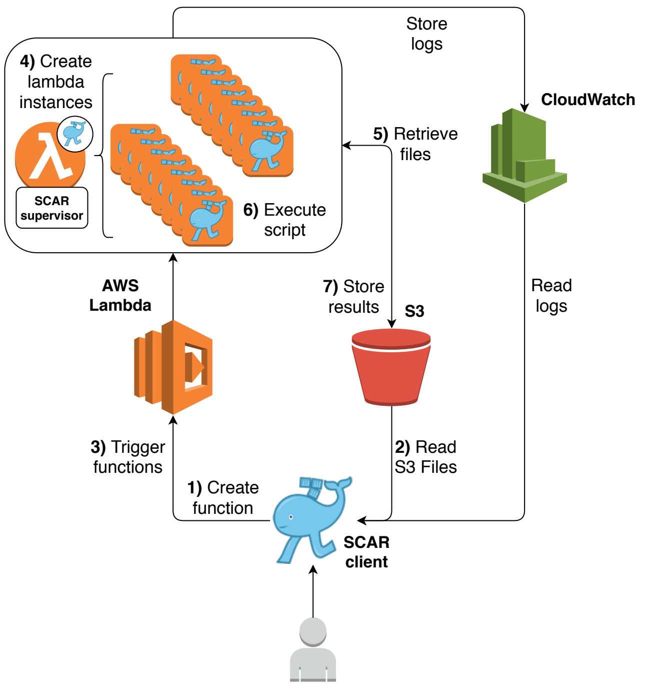

**Figure 2: Workflow of the Massive Medical Image Analysis Service. The user, through the SCAR client, 1) creates the lambda function and 2) reads the files needed to process. Then SCAR automatically 3) sends the events to AWS Lambda making 4) AWS Lambda to invoke the function as many times as events received. 5) Then, the supervisor deployed inside each invocation retrieves the required files 6) and executes the user script. 7) To finish, the supervisor stores the output files generated by the lambda invocations at the defined bucket.**

a temporal folder. This folder is made available inside the container that will host the execution of the application.

(5) Each lambda invocation, through the SCAR supervisor, executes the container and runs the script specified by the user. All the logs generated by the invocations are stored in the log service (i.e. Amazon CloudWatch).

(6) The last step of the execution consists on transferring the output files from the $SCAR_OUTPUT_FOLDER of the lambda function to the S3 bucket. This is also done transparently to the user by the SCAR supervisor.

After the execution finishes, using the SCAR CLI, the user can check the log files and retrieve the generated output files stored in the S3 bucket.

## 5.2 Video Analysis Service

This case study presents an architecture that takes a video file as an input and stores as a result the analysis of the keyframes of such video using an state-of-the-art real time object detection system [33]. The goal is to apply object recognition to certain frames of

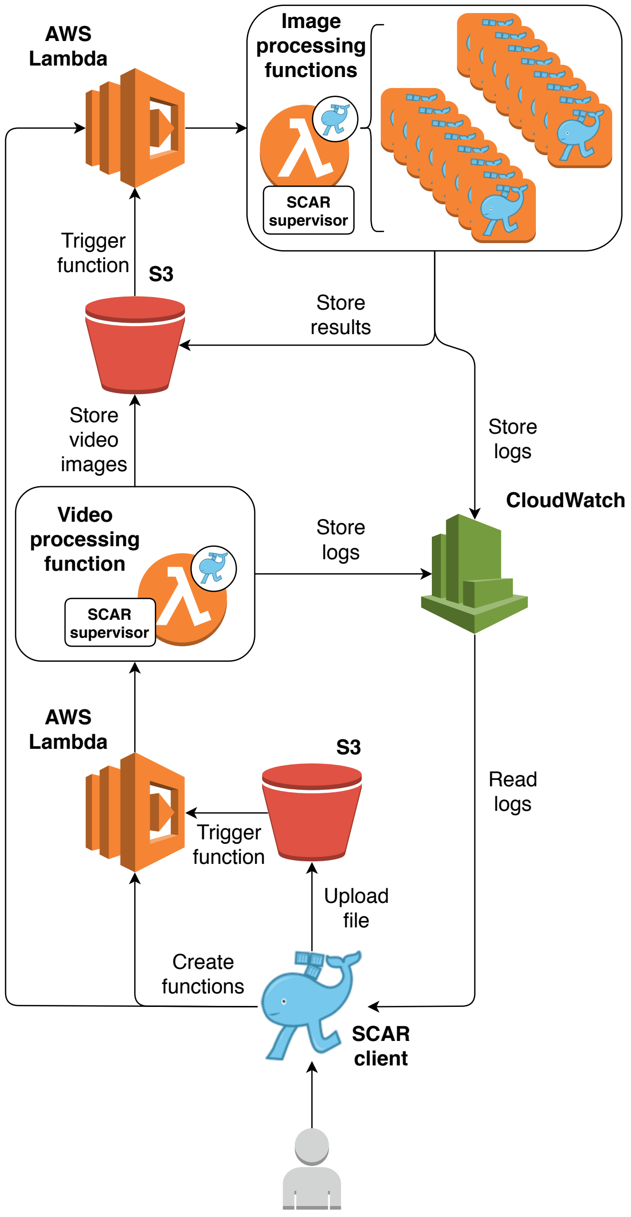

**Figure 3: Workflow of the Video Analysis Service. Two lambda functions are used. First, a function to extract the keyframes of the video and second, a function to analyze such keyframes. One instance of the second type of function is launched automatically for each keyframe found in the input video.**

the video in order to reason about the content of the video. Figure 3 shows the workflow of the architecture proposed.

The *ffmpeg* library is used to extract the keyframes from the input video and the *darknet* framework in combination with the yolov3 library [33] is used to analyze the extracted keyframes. The darknet application has been compiled to use CPUs (since GPUs cannot be currently used in AWS Lambda).

The following trace details the steps taken to carry out the experiment. These steps skip the image and script definition:

(1) Through the SCAR client, the user creates the two functions needed for the architecture. The video processing function is linked with two different buckets as input and output. The SCAR framework allows to link different functions to the same bucket, but in this experiment we are using two different ones. Indeed, in order to chain the functions and compose the pipe-lining workflow, the input folder of the image processing function needs to be the same as the output folder of the video processing function.
(2) Using the SCAR client, the user uploads a video to the input folder of the S3 bucket linked with the video processing function. This is the last step that requires user intervention.
(3) After the video upload finishes, the S3 bucket automatically sends a trigger to activate the video processing function. The SCAR supervisor deployed inside the lambda function retrieves the video from the S3 bucket and stores it in a folder shared with the container. Then the container is launched and the function extracts the keyframes of the video and stores the in the `$SCAR_OUTPUT_FOLDER`. The SCAR supervisor stores those files in the output folder of the specified S3 bucket, thus triggering the image processing lambda function.
(4) To finish, each image stored triggers an image processing lambda instance that analyzes the keyframe and through the supervisor, stores the result in the output folder of the S3 bucket.

The logs generated by all the invocations are stored automatically in the log service (i.e. Amazon CloudWatch). After the execution finishes, using the SCAR client, the user can check the log files and retrieve the generated output files stored in the S3 bucket.

## 5.3 Results

To test the scalability and parallel file processing capabilities of the architectures several workloads were tested. This section resumes the results from the two case studies. Figure 4 shows the execution times of the medical image analysis service. Figure 5 shows the execution times of the video analysis service. In both figures, the vertical axis (i.e. seconds) is represented using a logarithmic scale.

Both case studies were executed in four different environments: 1) a local PC machine with 4 CPUs (model i5-4590) and 8GB of RAM, 2) a c5.large EC2 machine with 2 virtual CPUs and 4GB of RAM, 3) a c5.18xlarge EC2 machine with 72 virtual CPUs and 144GB of RAM, and 4) a Lambda function with 3008MB of RAM.

To avoid the cold start of the lambda functions (see [36] and [32] for details), the SCAR client *preheats* the lambda functions doing a synchronous first call during the initialization step. Once the initialization finishes, the invocations done during the running phase (i.e. step 3 of Figure 2) or the function invocations from the S3 bucket, are performed asynchronously and executed in parallel.

In Figure 4 it can be seen how AWS lambda starts to outperform the rest of platforms when a high level of parallelism is required. Also, due to the high number of cores and threads available in the c5.18xlarge machine this EC2 machine presents also a good performance when facing up to 100 images, although when the load is increased it suffers from the lack of scalability like the other static environments.

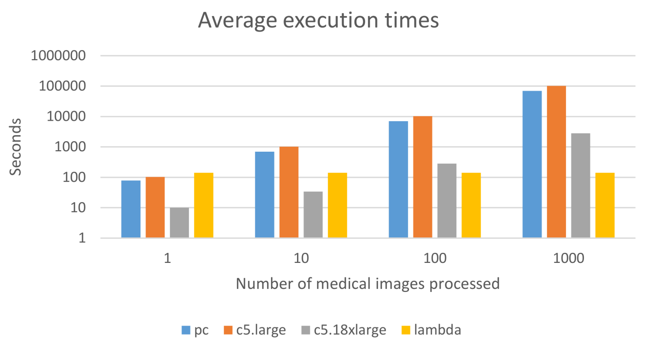

**Figure 4: Average execution times for the process of different number of medical images in different execution environments.**

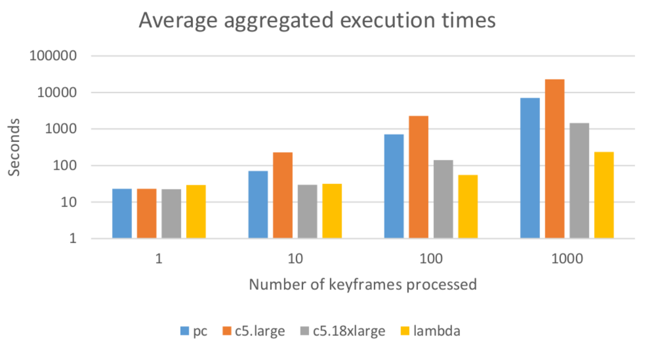

**Figure 5: Average execution times for processing videos with different number of keyframes in different execution environments.**

Figure 5 shows how the video processing function is acting as a bottleneck. Since each video is processed by exactly one invocation of this function, the more keyframes we extract the more time the function takes, thus not taking advantage of the parallel infrastructure. Notice that an extension of this architecture using a coordinator function that spawns multiple invocations of video processing function to process part of the video would increase the throughput.

To finish, it is important to remind that the parallel execution of the 1000 instances is achieved automatically by the Lambda service without any extra configuration and services like a load balancer or a job scheduler. Moreover, matching the execution times achieved by the lambda functions with the c5.large instances would require a significantly large number of instances, beyond the default limits set by the Cloud provider.

## 5.4 Cost analysis

This section provides an analysis of the economic cost of the case study carried out in section 5.1. To analyze if it is cost-wise to use

| AWS Service | Type | Cost ($/h) | CPU units (ECU) |
|---|---|---|---|
| Lambda | 3008MB | 0.176292 | 5.75* |
| EC2 | c5.large | 0.085 | 8 |
|  | c5.18xlarge | 3.06 | 278 |

**Table 1: AWS services used in the case study shown in section 5.1. Data extracted from the AWS documentation. The EC2 instances used are on-demand. ECUs for lambda are estimated on the basis of the execution times of the case study.**

the serverless programming model in combination with the architecture proposed, we are going to calculate the costs of processing 1000 medical images using different AWS services (i.e. Lambda and EC2), representing two different computational paradigms. Table 1 resumes the AWS infrastructures used and their respective properties.

In order to compare different instance types with different underlying architectures, AWS introduces the EC2 Compute unit or ECU[3]. As it can be seen in Table 1, only the ECU units for the `c5.large` and the `c5.18xlarge` are defined in the official documentation[4]. Thus, the ECU provided by the Lambda service are estimated by comparing the execution time of processing the same test image in an already measured machine (i.e. `c5.large`) and then applying $\frac{c5.large\_cpu\_time*c5.large\_ECU}{instance\_time}$ being *instance_time* the time used by a lambda invocation to process one image.

Table 2 summarizes the cost calculations for the case study. The equation used to calculate the cost of different EC2 instances (represented in the 4th column (Cost ($)) of Table 2 is the following:

$$(instance\_time * 60)min * \left\lceil \frac{instance\_cost}{60} \right\rceil \$/min \tag{1}$$

In equation 1, we adapt the calculations made for EC2 instances to the new per-second billing policies of Amazon EC2[5].

Also, Table 2 shows the number of concurrent instances needed to match the lambda execution time and its respective costs. To calculate the total cost presented in the last column, first it is calculated the number of machines needed, dividing the total execution time of the application in each instance by the time used by lambda and rounding up. Then, with the number of machines ($N$) and using equation 2, it can be calculated the total cost of a multi-instance execution.

$$N * \left\lceil \frac{lambda\_time}{60} \right\rceil min * \frac{instance\_cost}{60} \$/min \tag{2}$$

In Lambda, the average execution time for the complete execution is 142 seconds. An invocation is performed for every image to analyze, so SCAR concurrently instantiates 1000 lambda functions. When using lambda functions the time to process 1 or 1000 images is the same and the total cost of the execution is, applying equation 1, 6.934$.

[3]http://aws.amazon.com/ec2/faqs/

[4]https://aws.amazon.com/ec2/pricing/on-demand/

[5]https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing/

In Figure 4 we can see that the `c5.large` instance takes 102500 seconds (i.e. almost 29 hours) to process the 1000 images with a cost of 2.42$. Using `c5.large` instances concurrently would require 722 machines to match the Lambda execution time. The final cost is not calculated because 722 EC2 instances surpasses by far the default maximum number of machines allowed by the provider for each zone (i.e. 20). On the other hand, the `c5.18xlarge` takes 2820 seconds (aprox. 47 min) with a cost of 2.397$. If we wanted to execute the case study in the same time as lambda, we would need 20 `c5.18xlarge` machines and it would cost us 2.414$.

## 6 DISCUSSION

The following section discuss the cost-effectiveness of the proposed architectures, based on serverless computing, compared to traditional solutions based on Virtual Machines, i.e. Infrastructure as a Service Cloud Computing. Two important aspects need to be emphasized at this point. First, AWS sets a default limit in the number of concurrent instances running at the same time at 20, although the limit can be raised if it is requested to the provider. Second, and the most important, launching several instances at the same time to execute a high number of jobs inside them requires orchestration. We would need to use a system able to deploy all the instances concurrently, like Auto Scaling, and also a job scheduler (i.e., SLURM, Torque, HTCondor or similar) to ensure the job execution and tracking. This complicates the multi-instance execution, in contrast with the easy parallelism that the Lambda services offer in combination with the middleware presented.

Considering the cost analysis done in section 5.4, it can be outlined that the ease of use and the reduced execution times of the Lambda platform supposes a price increment. The AWS Lambda service is more expensive than the EC2 instances, but is easier to configure and launch, specially when used in combination with SCAR. Moreover, cheaper solutions involve increasing the complexity of the execution, since orchestration tools would be needed to manage the execution (as explained above). Thus, in cost terms, better solutions than Lambda exist, but featuring important drawbacks that can made them unfeasible to the user with little or no experience in infrastructure deployment, which are the target users of the programming model and middleware presented in this paper. Also, it is important to point that the SCAR enables to deploy a complex application and automatically provide an HTTP endpoint to trigger its execution where the cost is linearly dependent on the amount of requests to the endpoint and the resources consumed. Exposing a highly-available highly-scalable cost-effective endpoint for a generic application on a Cloud platforms paves the way for the adoption of serverless computing for the execution of complex scientific applications, even data-oriented workflows.

## 7 CONCLUSIONS AND FUTURE WORK

This paper has introduced a programming model and an update of an existing middleware to create highly-parallel event-driven serverless applications. The execution environment for this applications was provided by containers created out of customized Docker images. To test the proposal, the deployment of two different applications was done in a real world provider, that is the AWS Lambda serverless computing platform.

| AWS Service | Type | Time to process 1000 images (s) | Cost ($) | Number of machines needed to match lambda execution time | Cost of the machines needed to match lambda execution time ($) |
|---|---|---|---|---|---|
| Lambda | 3008MB | 142 (2.36 min.) | 6.934 | - | - |
| EC2 | c5.large | 102500 (28.47 h.) | 2.42 | 722 | - |
|  | c5.18xlarge | 2820 (47 min.) | 2.397 | 20 | 2.414 |

**Table 2: Summary of the costs of the medical image case study. The cost uses data from Table 1, adopting per-minute billing.**

The ability to run code in response to events and the large-scale elasticity provided by the underlying serverless platform opens new avenues for efficient High Throughput Computing tasks. This was demonstrated by the case studies where the programming model abstracted away many implementation details typically required on computing frameworks. Several challenges are also identified, such as limitations in the amount of memory allocated to each function invocation and, the most limiting one, the maximum execution time. Moreover a cost analysis was done comparing the serverless programming models presented and the usual Cloud Computing architectures. Although the cost analysis revealed that running a serverless architecture could be costlier than deploying a simple EC2 machine, the savings in configuration and execution time in combination with the pay-per-use model offered by AWS make the serverless architectures a good option to deploy applications that have to deal with a high amount of short lived tasks.

In the future, we plan to simplify the definition of data driven workflows, so the user can define complete applications by using a simple infrastructure definition file. Moreover, we have plans to add support for more cloud providers and also tackle on-premises functions-as-a-service frameworks. This would lead to the deployment of hybrid serverless applications that encompass the high scalability capabilities of the cloud providers and the less restricted environments of the on-premises deployments.

## ACKNOWLEDGEMENTS

The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" with reference number TIN2016-79951-R.

## REFERENCES

[1] Amazon. Amazon Lambda (AWS Lambda). https://aws.amazon.com/lambda/. [Online; accessed 24-September-2018].

[2] Amazon. Amazon Web Services (AWS). http://aws.amazon.com. [Online; accessed 24-September-2018].

[3] Amazon. AWS Serverless Application Model (AWS SAM). https://github.com/awslabs/serverless-application-model. [Online; accessed 24-September-2018].

[4] Apex. Up, deploy serverless apps in seconds. http://apex.run/. [Online; accessed 24-September-2018].

[5] AWS. Amazon EC2 Container Service (ECS). https://aws.amazon.com/es/ecs/. [Online; accessed 24-September-2018].

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. Serverless Computing: Current Trends and Open Problems. pages 1–20, jun 2017.

[7] CoreOS. rkt. https://coreos.com/rkt/. [Online; accessed 24-September-2018].

[8] Databricks. Databricks Serverless. https://goo.gl/uagZ8E. [Online; accessed 24-September-2018].

[9] Docker. Docker. https://www.docker.com/. [Online; accessed 24-September-2018].

[10] Docker. Swarm Mode in Docker Engine. https://docs.docker.com/engine/swarm/. [Online; accessed 24-September-2018].

[11] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[12] T. A. S. Foundation. Apache openwhisk. http://openwhisk.org/. [Online; accessed 24-September-2018].

[13] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. aug 2017.

[14] D. Gannon. Observations about Serverless Computing With a few examples from AWS Lambda, Azure Functions and Open Whisk. Technical report, 2017.

[15] A. Glikson. TRANSIT: Flexible pipeline for IoT data with Bluemix and OpenWhisk. https://goo.gl/ThN2TR. [Online; accessed 24-September-2018].

[16] Google. Container Engine. https://cloud.google.com/container-engine. [Online; accessed 24-September-2018].

[17] Google. Google Cloud Functions. https://cloud.google.com/functions/. [Online; accessed 24-September-2018].

[18] Google. Google Cloud Platform. https://cloud.google.com. [Online; accessed 24-September-2018].

[19] Hashicorp. Nomad. https://www.nomadproject.io/. [Online; accessed 24-September-2018].

[20] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[21] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. feb 2017.

[22] Kubernetes. Kubernetes. https://kubernetes.io/. [Online; accessed 24-September-2018].

[23] LXC. LXC. https://linuxcontainers.org/. [Online; accessed 24-September-2018].

[24] G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 400–406, June 2016.

[25] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, mar 2014.

[26] Microsoft. Azure Container Service. https://azure.microsoft.com/services/container-service/. [Online; accessed 24-September-2018].

[27] Microsoft. Microsoft Azure. https://azure.microsoft.com. [Online; accessed 24-September-2018].

[28] Microsoft. Microsoft Azure Functions. https://azure.microsoft.com/en-in/services/functions/. [Online; accessed 24-September-2018].

[29] OpenNebula. OpenNebula. https://opennebula.org. [Online; accessed 24-September-2018].

[30] OpenStack. OpenStack. http://openstack.org. [Online; accessed 24-September-2018].

[31] D. O. S. Platform. Ooso, serverless mapreduce. https://github.com/d2si-oss/ooso. [Online; accessed 24-September-2018].

[32] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50 – 59, 2018.

[33] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[34] F. B. Torro, J. D. S. Quilis, I. B. Espert, A. A. Bayarri, and L. M. Bonmatí. Accelerating the diffusion-weighted imaging biomarker in the clinical practice: comparative study. *Procedia Computer Science*, 108(Supplement C):1185 – 1194, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[35] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, 2017.

[36] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking Behind the Curtains of Serverless Platforms. *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.