applied sciences

MDPI

Article

GPU-Enabled Serverless Workflows for Efficient Multimedia Processing

Sebastián Risco * and Germán Moltó

Instituto de Instrumentación para Imagen Molecular (I3M), Centro Mixto CSIC—Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; gmolto@dsic.upv.es
* Correspondence: serisgal@i3m.upv.es

Abstract: Serverless computing has introduced scalable event-driven processing in Cloud infrastructures. However, it is not trivial for multimedia processing to benefit from the elastic capabilities featured by serverless applications. To this aim, this paper introduces the evolution of a framework to support the execution of customized runtime environments in AWS Lambda in order to accommodate workloads that do not satisfy its strict computational requirements: increased execution times and the ability to use GPU-based resources. This has been achieved through the integration of AWS Batch, a managed service to deploy virtual elastic clusters for the execution of containerized jobs. In addition, a Functions Definition Language (FDL) is introduced for the description of data-driven workflows of functions. These workflows can simultaneously leverage both AWS Lambda for the highly-scalable execution of short jobs and AWS Batch, for the execution of compute-intensive jobs that can profit from GPU-based computing. To assess the developed open-source framework, we executed a case study for efficient serverless video processing. The workflow automatically generates subtitles based on the audio and applies GPU-based object recognition to the video frames, thus simultaneously harnessing different computing services. This allows for the creation of cost-effective highly-parallel scale-to-zero serverless workflows in AWS.

Keywords: cloud computing; serverless computing; multimedia processing; workflows; batch processing; containers

Citation: Risco, S.; Moltó, G. GPU-Enabled Serverless Workflows for Efficient Multimedia Processing. *Appl. Sci.* **2021**, *11*, 1438. https://doi.org/10.3390/app11041438

Academic Editor: Miguel García-Pineda
Received: 9 December 2020
Accepted: 2 February 2021
Published: 5 February 2021

1. Introduction

The advent of Cloud Computing introduced the ability to customize the computing infrastructure to the requirements of the applications through the use of virtualization. This resulted in the widespread adoption of Cloud computing for academic, enterprise and scientific workloads. However, migrating an application to a public Cloud required significant expertise in order to adapt the application to the elastic capabilities of the underlying services. In addition, the pay-per-use model typically resulted in a pay-per-deployment, where provisioned Virtual Machines (VMs) are billed regardless of their actual use.

To better accommodate short and spiky workloads, commonly found in microservices architectures, serverless computing was introduced via flagship services such as AWS Lambda [1]. This service allows the execution of user-defined functions coded in certain programming languages supported by the cloud provider in response to certain well-defined events (such as uploading a file to an S3 bucket, i.e., Amazon's object storage system [2] or invoking a REST API provided by API Gateway [3]). A fine-grained pricing scheme billed on milliseconds of execution time resulted in real pay-per-use. In addition, the ability to scale to zero allowed to deploy massively scalable services that can rapidly scale up to 3000 concurrent invocations but incurring in zero cost when the function is not being invoked.

Our previous work in the area is the open-source SCAR tool (SCAR: https://github.com/grycap/scar (accessed on 26 November 2020)) [4] which creates highly-parallel event-

driven file-processing serverless applications that execute on customized runtime environments, defined as Docker images, in AWS Lambda. This tool was successfully adopted to execute generic applications in AWS Lambda, support additional programming languages and even execute deep learning frameworks. However, AWS Lambda, as it happens with other Functions as a Service (FaaS) public services, imposes strict computing requirements. AWS Lambda functions run on a constrained environment, where the execution time and maximum RAM cannot exceed 15 min and 10,240 MB, respectively, the ephemeral disk storage is limited to 512 MB and no GPU support is available.

The main scientific challenge addressed in this contribution is to provide event-driven serverless workflows for data processing that simultaneously feature scale-to-zero, high elasticity and the support for GPU-based resources. This is achieved through the integration in SCAR of AWS Batch [5], a managed service to provision virtual clusters that can grow and shrink depending on the number of jobs to be executed, packaged as Docker containers.

Indeed, multimedia processing applications are both resource-intensive and typically require the definition of data-driven workflows in order to efficiently perform the execution of several phases. The large-scale parallelism of AWS Lambda can be exploited to accommodate the execution of short jobs that can be executed in the restricted execution environment provided by AWS Lambda, which limits the maximum execution time, the allocated RAM and, finally, provides limited ephemeral disk storage, as described earlier. Other more resource-demanding jobs should be executed in AWS Batch. To this aim, this paper describes the evolution of SCAR to: (i) integrate AWS Batch as an additional computing back-end for compute-intensive and GPU-based jobs and (ii) support a Functions Definition Language that can simultaneously use both Lambda and Batch for the execution of data-driven applications composed of multiple steps. This results in a tool that can foster serverless computing adoption for multiple enterprise and scientific domains, supporting any CLI-based file-processing application packaged as a container image. Potential scenarios for exploiting the tool can be event-driven multimedia processing, AI-based inference or large-scale software compilation.

After the introduction, the remainder of the paper is structured as follows. First, Section 2 describes the related work in this area. Then, Section 3 introduces the architecture of the system to support GPU-enabled serverless workflows for data processing. Next, Section 4 describes a use case to assess the benefits of the platform by supporting a serverless workflow for parallel audio and video processing. Later, Section 5 presents the results obtained after the execution of the use case. Finally, Section 6 summarises the main achievements of the paper pointing to future work.

2. Related Work

There are previous works in the literature that have adopted serverless for scientific computing. Indeed, pioneers in this area started to adopt AWS Lambda as a general computing platform aimed at scientific computing in order to take advantage of the massive elasticity of this service. This is the case of Jonas et al. [6], by introducing PyWren to execute distributed Python code across multiple Lambda function invocations to achieve a virtualized supercomputer. In addition, the work by Alventosa et al. [7] used AWS Lambda as the computing platform on which to execute MapReduce jobs for increased elasticity without preprovisioning a Hadoop cluster.

The usage of serverless computing for the execution of workflows has initially started to be explored. For example, the work by Malawski et al. [8] evaluates the aplicability of serverless computing for compute and data intensive scientific workflows through the creation of a prototype. This is in line with the work by Jiang et al. [9] which integrates a combination of Functions as a Service (FaaS)/local clusters execution approach for Montage-based workflows. The work by Skluzacek et al. [10] describes a service that processes large collections of scientific files to extract metadata from diverse file types, relying on Function as a Service models to enable scalability. Furthermore, the work by Chard et al. [11] proposes funcX, a high-performance FaaS platform for flexible, efficient,

and scalable, remote function execution on infrastructures such as clouds, clusters, and supercomputers. The work by Akkus et al. [12] focuses on high-performance serverless computing by introducing a serverless computing system with enhanced capabilities such as application-level sandboxing and a hierarchical message bus in order to achieve better resource usage and more efficient startup.

The adoption of cloud computing for multimedia processing is certainly another active field of study. Indeed, the paper by Sethi et al. [13] already addressed the application of scientific workflows for multimedia content processing, leveraging the Wings [14] framework to efficiently analyse large-scale multimedia content across the Pegasus engine [15], which operates over cloud infrastructures. Another example of the adoption of cloud technologies in this area is the study conducted by Xu et al. [16], where they propose the development of a workflow scheduling system for cloudlets based on Blockchain, ensuring the QoS of multimedia applications in these small-scale data centres near the edge. Finally, the study carried out by Zhang et al. [17] specifically focuses on cost-effective serverless video processing. They quantify the influence of different implementation schemes on the execution duration and economic cost from the perspective of the developer. Nonetheless, support for GPU-based processing within the serverless computing paradigm is an open issue nowadays.

The main contribution of this paper to the state of the art is the development of an open-source tool to support serverless computing for mixed data-driven workflows that involve disparate computing requirements for the different phases, being able to simultaneously harness GPU and CPU computing and the highly elasticity provided by AWS Lambda. To the best of the authors' knowledge this has not been addressed in the past.

3. Architecture of the Serverless Processing Platform

This section outlines the details of the redesigned serverless platform in order to be integrated with AWS Batch, highlighting the different cloud services involved and their respective roles. Additionally, it addresses the implementation details required in SCAR to support the definition of functions on the proposed platform, thus creating an updated framework for the execution of data-driven serverless workflows for container-based applications in the Cloud. As a result of this study, SCAR is now able to orchestrate all the resources needed to run GPU-enabled file-processing workflows while maintaining zero cost for the user when the platform is idle.

3.1. Components

SCAR allows to define functions, which are triggered in response to well-defined events, to execute in AWS Lambda a user-defined script inside a container created out of a Docker image. This job is in charge of performing the processing of the data that triggered the event. SCAR supports a programming model to create highly-parallel event-driven file-processing serverless applications, as described in the work by Pérez et al. [18]. The SCAR platform is built on several AWS services and these functions can be remotely invoked through HTTP-based endpoints created by API Gateway or by uploading files to Amazon S3 buckets, allowing the event-driven processing of files. Moreover, the platform automatically stores the job execution logs in Amazon CloudWatch [19]. Docker container images are usually fetched from publicly available container registries such as Docker Hub [20]. However, it has been designed in such a way that the command line interface is decoupled from the service provider's client. Therefore, in future releases it could be integrated with other public Cloud providers offering similar serverless services, such as Google Cloud [21] or Microsoft Azure [22].

It is important to point out that AWS Lambda recently included the ability to use certain Docker images as part of the runtime environment, in line with our previous developments. However, this support precludes using arbitrary images from Docker Hub, a widely used public repository for application delivery based on Docker images.

Figure 1 illustrates the different services involved and their interaction to support workloads that overcome the limits imposed by AWS Lambda and allow increased control in the definition of computational resources. A more detailed description of the SCAR architecture has been previously described in [4]. Therefore, the main goal of this contribution is to extend SCAR with the ability to deal with data-driven serverless workflows that involve resource-intensive jobs that exceed the computing capabilities of AWS Lambda, by integrating seamless support for AWS Batch.

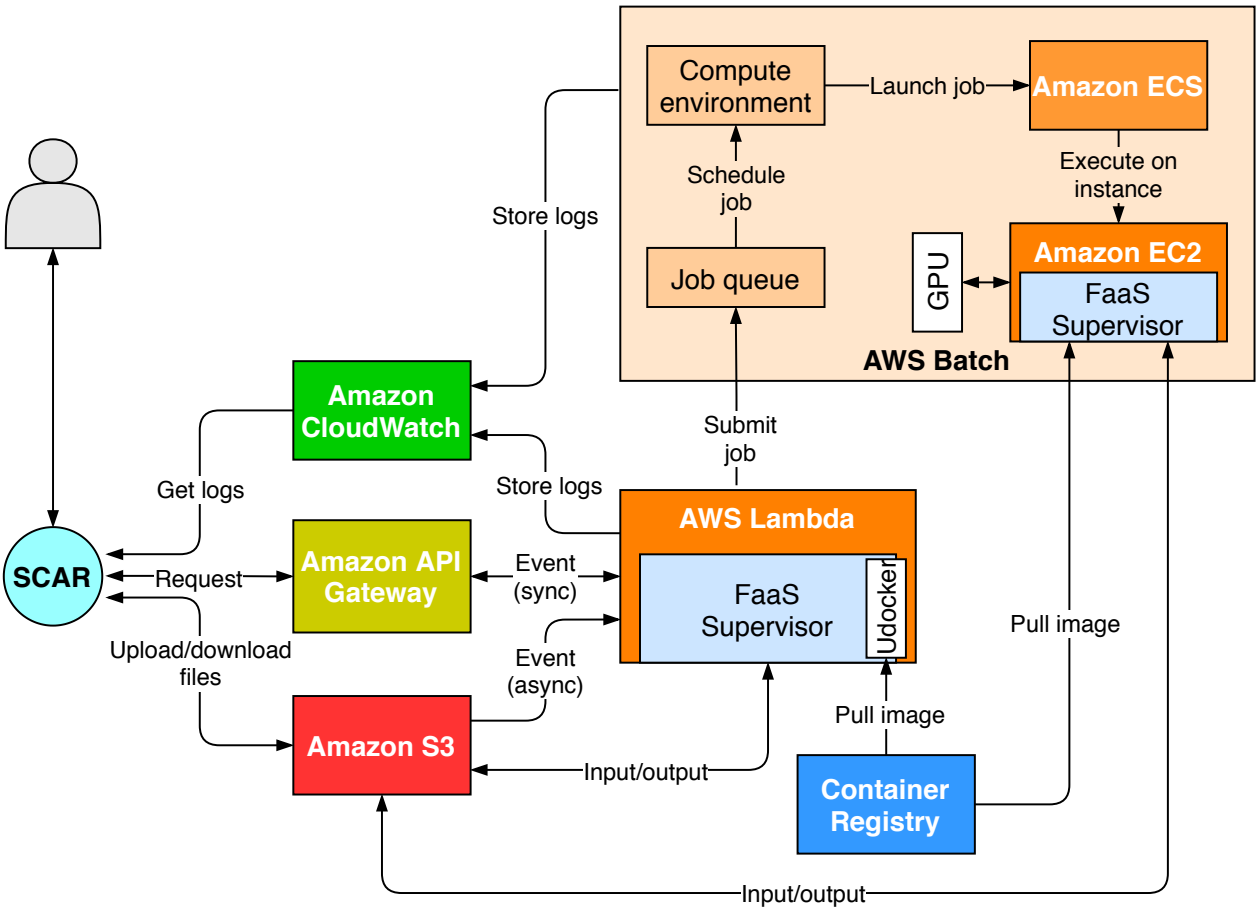


Figure 1. Architecture of the SCAR platform integrated with AWS Batch in order to support long-running and GPU-accelerated jobs.

AWS Batch runs Docker-based computational jobs on EC2 [23] instances. Compared to AWS Lambda, the resource requirements of these jobs can be configured by the user in terms of: an increased memory allocation, the assignment of the desired number of CPUs, the instance types to be used and the assignment of GPU devices to containers, among others. AWS Batch is composed by several modules that must be defined before the actual execution of jobs:

- *Compute environment*: Computing resources on which the jobs will be executed, described in terms of instance types together with the maximum and minimum number of available nodes of the ECS [24] cluster that will be automatically created. The cluster features elasticity so that additional nodes will be automatically added (up to the maximum number of nodes) depending on the number of pending jobs to be executed and will be automatically terminated when no longer required according to a set of predefined policies enforced by AWS Batch.
- *Job queue*: Managed queues where the jobs are submitted and stored until the compute environment they are assigned to is ready to perform the execution.

- *Job definition*: The basic specification of a job, which includes the Docker image to be used, the number of vCPUs and memory allocated, the request for GPUs, the command to be executed and the environment variables. All jobs must be linked to a job definition. However, jobs can add and modify certain parameters when they are created. AWS Batch automatically executes the jobs that request GPU access with the NVIDIA container runtime [25].

Although AWS Batch is a traditional batch processing system, it can scale-to-zero, that is, terminate all the nodes in the cluster while maintaining at no extra cost the managed job queues to receive subsequent job execution requests. This, together with the event-driven execution mechanisms of AWS Lambda implemented by SCAR, allow one to submit jobs automatically when new files are uploaded to a bucket. Hence, the AWS Batch service has been integrated while maintaining the principles of serverless computing. However, the boot time of the EC2 instances is substantially larger than the initialization time of AWS Lambda functions, as it will be shown in Section 5.3. Furthermore, the pricing scheme of AWS Batch uses per-second billing of the provisioned EC2 instances that compose the cluster, instead of per millisecond, as it happens in AWS Lambda.

3.2. Integration of SCAR with AWS Batch

Introducing support for AWS Batch in SCAR required extending the framework in order to create and configure the Batch resources employed depending on the execution modes selected. The development used the AWS SDK for Python (Boto3) [26]. Furthermore, it also required extending the *faas-supervisor* (FaaS Supervisor: https://github.com/grycap/faas-supervisor (accessed on 26 November 2020)), an open-source library to manage the execution of user scripts and containers in AWS Lambda and also in charge of managing the input and output of data on the Amazon S3 storage back-end. This was redesigned to delegate jobs to AWS Batch. To do this, the FaaS Supervisor, which runs in the AWS Lambda runtime as a Layer, is able to identify the execution mode specified in the function definition. When it must delegate the execution to AWS Batch, it will submit a new job to the function's job queue based on the job definition previously created by the SCAR client, embedding the event in an environment variable. Thus, three execution modes are now supported depending on the user's preference regarding where the job will be executed:

- *lambda*: The jobs are ran as user-defined container images on AWS Lambda. The FaaS Supervisor employs *udocker* [27] to pull the Docker image from a container registry and execute it inside the AWS Lambda runtime.
- *batch*: AWS Lambda acts as a gateway for events but function invocations are translated into AWS Batch jobs. The event description is passed down to the job as an environment variable, allowing the FaaS Supervisor, which runs on the EC2 instance, to parse it to perform data stage in/out. When functions are defined in this mode, the SCAR client is responsible for creating the required AWS Batch components (compute environment, job queue and job definition).
- *lambda-batch*: Functions are first executed on AWS Lambda. If the execution fails or the function timeout has almost been reached, the job is automatically delegated to AWS Batch. This mode allows using AWS Lambda to effectively scale upon a large burst of short jobs while ensuring that more demanding jobs will be eventually processed whenever the AWS Lambda limits are exceeded.

The FaaS Supervisor runs on the AWS Batch jobs as a binary that is downloaded during the startup of each EC2 instance belonging to the ECS cluster created by AWS Batch. To do this, the SCAR client automatically creates a launch template containing the download commands in the *cloud-init* [28] user data. Then, the path containing the FaaS Supervisor is mounted automatically as a volume on the job containers.

3.3. Functions Definition Language for Serverless Workflows

To facilitate the creation of data-driven serverless workflows from configuration files, a Functions Definition Language (FDL) has been defined that, in contrast to previous versions of SCAR, supports the definition of multiple functions from a single YAML file. The processing of these files is possible due to the implementation of a completely redesigned parser in the SCAR client.

This language focuses on the definition of the resources for each function (i.e., container image, script to be executed, memory, CPUs, GPU access, etc.) and allows to set them to use the aforementioned execution modes. This way, a performance preprofiling of the multiple stages of a scientific workflow determines whether a certain function should be executed (i) exclusively in AWS Lambda, because it complies with its computing limitations, (ii) exclusively in AWS Batch, because the application may require additional memory/execution time beyond the maximum available in AWS Lambda or, finally, (iii) using the *lambda-batch* execution mode to easily accommodate disparate computing requirements.

In contrast to the classic FaaS platforms, in the FDL a user script has to be defined for each function, containing the commands to process the file that triggers the event. Hence, previously developed multimedia applications are supported without the need to adapt them to the Functions as a Service model.

The different functions are linked together through Amazon S3 buckets, which can be defined within the `input` and `output` variables. This allows data-workflows to be easily created, being the output bucket of one function the input of another, which will result in a new event that will trigger it. As an enhancement, the FaaS Supervisor component has also been improved to handle the new FDL, as well as to filter the output files according to their names and/or extensions in a postprocessing stage. This stage allows the upload of several files to different output buckets, thus allowing to split workflows into different branches, as shown in the use case described in Section 4.

A detailed example of the FDL shown in Figure 2, together with a test video and deployment manual are available in GitHub (*av-workflow* example: https://github.com/grycap/scar/tree/master/examples/av-workflow (accessed on 26 November 2020)). This corresponds to the serverless workflow used as a case of study in the following section, for the sake of reproducibility of the results. As can be seen in the *scar-av-workflow-yolov3* function, enabling GPU-accelerated computing is as simple as setting the `enable_gpu` variable to `true` and choosing an instance type that has at least one graphics processing unit. Of course, the application must support the execution on a GPU.

```
---
functions:
aws:
- lambda:
name: scar-av-workflow-ffmpeg
container:
image: jrottenberg/ffmpeg:4.1-ubuntu
init_script: ffmpeg-script.sh
execution_mode: lambda-batch
memory: 1024
timeout: 900
input:
- storage_provider: s3
path: scar-av-workflow/start
output:
- storage_provider: s3
path: scar-av-workflow/video
suffix:
- avi
- storage_provider: s3
path: scar-av-workflow/audio
suffix:
- wav
- lambda:
name: scar-av-workflow-audio2srt
container:
image: grycap/audio2srt:mini
init_script: audio2srt-script.sh
execution_mode: lambda-batch
memory: 1024
timeout: 900
input:
- storage_provider: s3
path: scar-av-workflow/audio
output:
- storage_provider: s3
path: scar-av-workflow/result
- lambda:
name: scar-av-workflow-yolov3
container:
image: grycap/yolov3:opencv-cudnn
init_script: yolov3-script.sh
execution_mode: batch
memory: 128
input:
- storage_provider: s3
path: scar-av-workflow/video
output:
- storage_provider: s3
path: scar-av-workflow/result
batch:
vcpus: 4
memory: 12288
enable_gpu: true
compute_resources:
max_v_cpus: 12
instance_types:
- g3s.xlarge
```

Figure 2. Functions Definition Language example.

4. Serverless Workflow for Multimedia Processing

In order to demonstrate the benefits and performance of the platform, a use case has been defined that builds a serverless workflow to perform frame-level object detection in video together with the inclusion of subtitles from the audio transcript, with potential applications in surveillance. This demonstrates the ability of SCAR to provide an event-driven service for multimedia processing, triggered by video uploads to an S3 bucket that can automatically scale up to multiple function invocations and several EC2 instances to cope with the workload and then automatically scale down to zero provisioned resources.

Hence, the platform allows the deployment of a highly available multimedia file-processing service with automated elasticity to support large workloads while maintaining zero cost when it is not in use.

Figure 3 shows the different functions that compose the workflow, as well as the folders in the S3 bucket used for input and output. These container-based functions use the following open-source software:

- *FFmpeg* [29]. Used to preprocess the videos uploaded to the *start* folder, extracting and converting the audio to the input format expected by the *audio2srt* function, as well as converting the videos to a suitable format (if they are not) for the object detection stage with YOLOv3. This function has been configured with the *lambda-batch* execution mode, since, depending on the quality and duration of videos, it could not fit in the Lambda execution environment. According to the new output postprocessing stage, the function will upload the resulting files to the *audio* or *video* folder depending on their extension (*.avi* or *.wav*).
- *audio2srt* [30]. A small application to generate subtitles from audio transcripts obtained through CMUSphinx [31], which uses acoustic models for speech recognition. The application, together with the models, has been packaged in a Docker container so that it can be defined as a serverless function in SCAR. This function will be triggered when the FFmpeg function uploads the extracted audio to the *audio* folder and, after processing, will store the resulting subtitle file to the *result* folder.
- *YOLOv3* [32,33]. A real-time object detection system that, using the Darknet [34] neural network framework, can run on GPUs to accelerate the video inference process. It has been compiled with CUDA [35] support and packaged as a container to be executed in GPU-accelerated AWS Batch compute environments. GPU access has been enabled in the definition of the function, which has also been configured to be triggered when videos are uploaded to the S3 *video* folder and to store the result in the *result* folder.

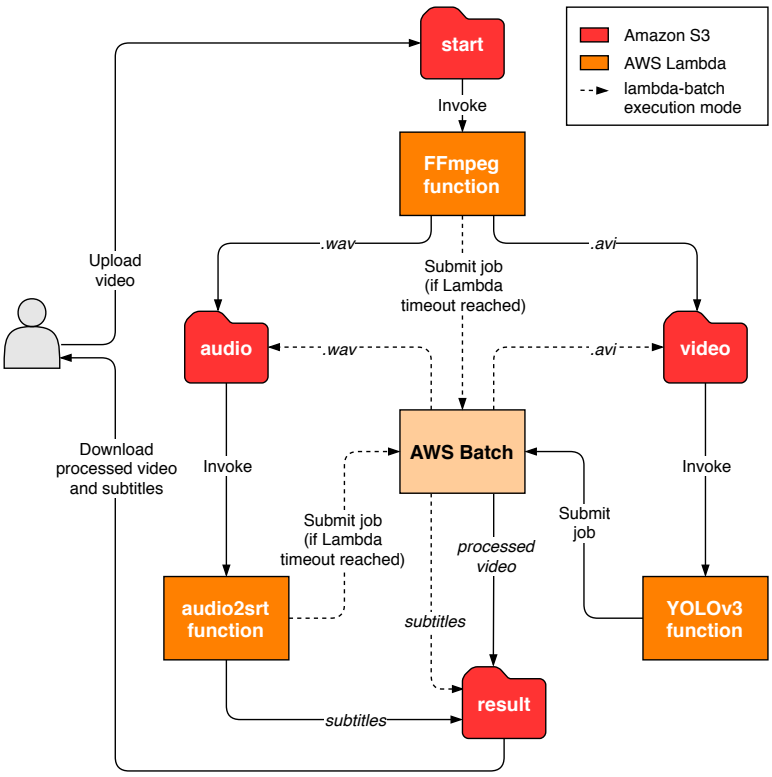


Figure 3. Simplified diagram of the multimedia processing workflow.

Users will only have to download the files from the *result* folder through the SCAR tool and open them in a multimedia player in order to watch the resulting video with prediction boxes together with the automatically generated subtitles, as shown in Figure 4.

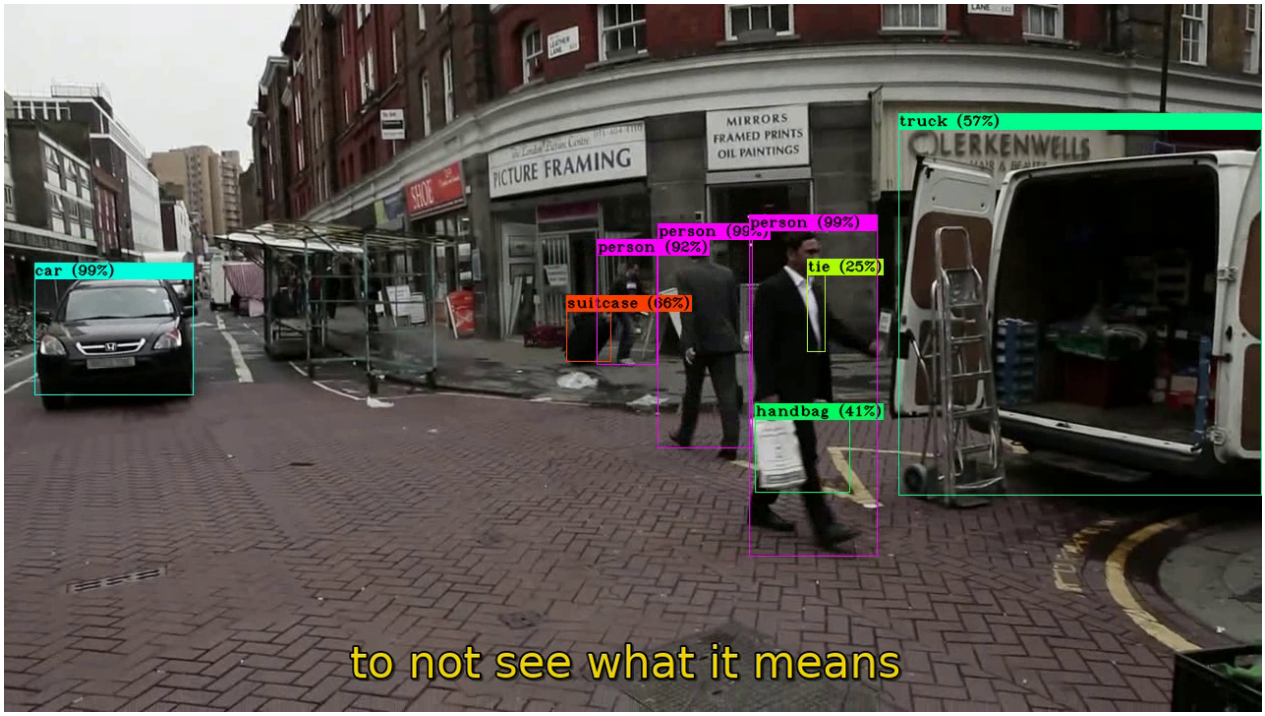


Figure 4. Snapshot of a video resulting from the object recognition function along with the automatically generated subtitles after a workflow execution.

5. Results and Discussion

The experimentation carried out is divided into four differentiated stages. First, Section 5.1 covers the *lambda-batch* execution mode and discusses how it affects the processing of variable duration audio files in terms of time and cost. Next, Section 5.2 compares different instance types to be used in the video object detection function with the aim of choosing the most efficient one. Then, Section 5.3 analyses the time taken by the Batch scheduler to scale the number of instances belonging to its compute environment, as well as their boot time. Finally, Section 5.4 discusses the results of the execution of the serverless workflow for processing several videos.

5.1. Analysis of the Lambda-Batch Execution Mode

A key contribution of this study has been the integration of SCAR with AWS Batch to support functions that require more resources than those allowed by AWS Lambda, including support for GPU-based processing. In addition, overcoming the limitation of execution time to 15 min has been an important motivation for this development.

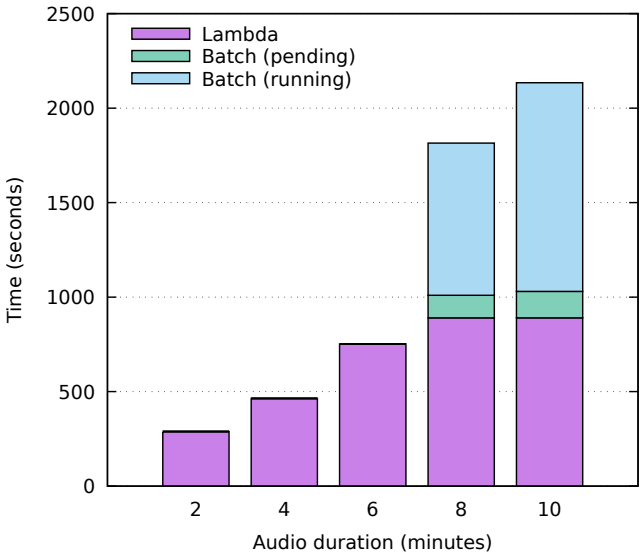
In the field of multimedia processing we can find different applications optimised to support accelerated computing or that directly require a execution time longer than 15 min. However, there may be uses in which the processing fits into AWS Lambda in most cases but eventually exceeds the execution time limit. It is precisely with this possibility in mind that the *lambda-batch* execution mode has been developed. The *lambda-batch* mode ensures that the file to be processed is handled even if the AWS Lambda timeout is reached. For this purpose, the *faas-supervisor* component has a timeout threshold that reserves a few seconds of the execution time. Thus, if the processing exceeds the maximum execution time, the *faas-supervisor* will have time to delegate the processing to AWS Batch.

In order to decide in which cases this execution mode is appropriate and to assess its impact on execution time and cost, an analysis has been carried out on the *audio2srt* function

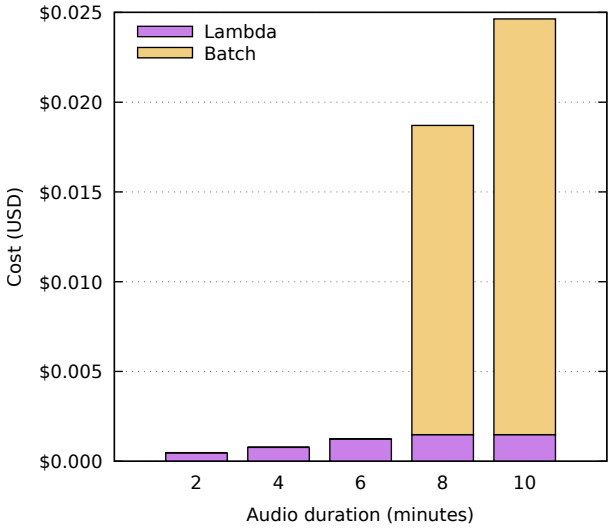
of the workflow presented in the previous section. This analysis consists of processing several waveform audio files of different lengths (i.e., 2, 4, 6, 8 and 10 min). 1 GB of RAM has been allocated for the function in both services (Lambda and Batch). The `m3.medium` instance type has been used in the AWS Batch compute environment, which has an Intel Xeon E5-2670 processor and an hourly cost of $0.067, billed by the second. Furthermore, the cost of the function in AWS Lambda is $0.0000000167 per millisecond.

Figure 5a shows the times obtained after five different executions of each audio file. The execution time in AWS Lambda is depicted in purple, while the time taken to be processed in AWS Batch has been divided into two categories: the time the job remains in the job queue (green) and the actual run time (blue). It is important to mention that no variations have been appreciated in the different executions of the analysis, apart from the pending time of the jobs executed in AWS Batch, which will be discussed in detail in Section 5.3. As can be seen, the audio files with lengths of 2, 4 and 6 min are performed entirely on AWS Lambda. Out of the different files tested, the 6-min file is the last one that could be processed completely on Lambda, with an average time of 753 s. Notice that the executions of the audio files with corresponding lengths of 8 and 10 min did not complete before reaching the 15 min execution timeout and, therefore, they were delegated to AWS Batch. The total processing time in such cases increases considerably, since the AWS Lambda timeout must first be reached and then the AWS Batch computing environment needs to start the required instances in order to subsequently execute the job. The Batch (pending) time shown in green has been calculated on average, since this time can vary significantly, as it depends on the AWS Batch autoscaling scheduler, which is discussed in Section 5.3. In addition, in the Batch (running) time shown in blue it can be seen that, although the function has the same amount of memory on both platforms, the processing time in Batch is lower due to the higher performance of the processor in the instances of the computing environment. Figure 5b shows how the processing cost also increases in these cases, since AWS Batch's pricing is not as fine-grained (per second instead of per millisecond) and, generally, the cost per hour of the instances used is also higher.

Therefore, the choice of this execution mode is worthwhile when dealing with applications whose execution time is close to the Lambda timeout or when the size of the files to be processed is variable and the processing time is not a requirement. Consequently, both the *batch* and the *lambda-batch* execution modes would not be suitable for real-time processing due to the increased time to provision the underlying computing instances. This is in contrast to the *lambda* execution mode, since the reduced start-up times provided by AWS Lambda can benefit these kind of applications.



(**a**) Average processing times



(**b**) Average processing cost

Figure 5. Time and cost analysis of the *audio2srt* function for different audio durations.

5.2. GPU and CPU Comparison for Video Processing

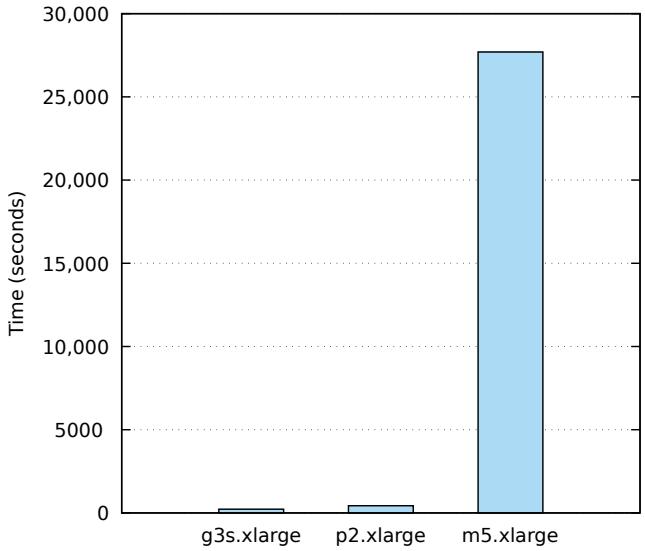
Before the experimentation of the whole workflow, the video processing function was evaluated in order to determine the instance type that would best suit the requirements of the application in terms of execution time and cost. To demonstrate that GPU acceleration is worthwhile in deep learning inference processes, a comparison was performed using different EC2 instances to process the same 4-min video with a 1280 $\times$ 720 resolution, consisting of a total of 6000 frames.

To test the execution time on CPU, the Batch compute environment was configured to use `m5.xlarge` instances, which have four virtual CPUs of the Intel Xeon Platinum (Skylake-SP) processor with a clock speed of up to 3.1 GHz, with 16 GB of RAM. The on-demand cost of this instance type is $0.192 per hour. In order to take advantage of the multiple vCPUs, Darknet was compiled with OpenMP support and jobs were configured to simultaneously use all the four vCPUs.

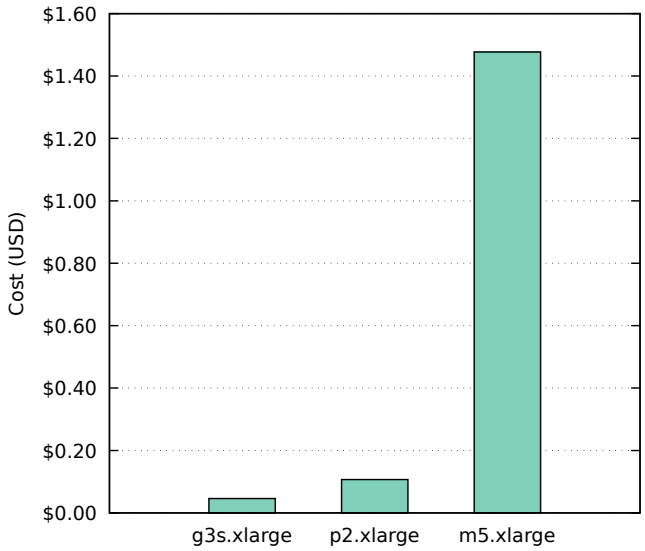
The performance over GPUs was measured using two different instance types: `p2.xlarge`, with 1 NVIDIA Tesla K80 GPU, 4 vCPUs, and 61 GB of RAM, which costs $0.9 USD per hour on demand; and `g3s.xlarge` with 1 NVIDIA Tesla M60 GPU, 4 vCPUs, and 30.5 GB of memory,

with an on-demand cost of $0.75 USD per hour. To leverage GPU acceleration, Darknet was compiled with CUDA.

Figure 6 shows the time and cost of an execution to process the same video using our platform with different EC2 instances. A single execution has been deemed adequate as the results show the considerable advantage of using GPU-based acceleration. As can be seen, the reduced cost per hour of the `m5.xlarge` instance does not outweigh the long duration spent for the execution. Therefore, it is highly recommended to accelerate via GPU such applications, not only to improve the processing time but also to increase savings.



(**a**) Total execution time



(**b**) Total execution cost

Figure 6. Comparison between CPU and GPU instances for video object detection.

Among the analysed GPU instances, the best performer was the `g3s.xlarge`. This is due to the fact that it has a lower end graphics card but of a later generation. This way, even with fewer GPU memory, i.e., 8 GB instead of the 12 GB available in each NVIDIA Tesla K80 GPU, it is able to perform a higher amount of operations at the same time. Furthermore, it has less RAM, which also affects its price. These reasons have led to the choice of the `g3s.xlarge` instance in the compute environment used by the video processing function of the workflow.

5.3. AWS Batch Auto Scaling

As mentioned above, AWS Batch compute environments are based on ECS cluster. These clusters execute the jobs in autoscaled groups of EC2 instances whose size grows and shrinks according to the workload. The main feature of this service is the scale to zero, which allows a real pay-per-use model. However, the time taken by the scheduler to launch and terminate instances is considerably longer than that of Functions as a Service platforms. This is due to the usage of traditional virtual machines instead of the container-based microVMs used by AWS Lambda [36]. Furthermore, the boot and initialisation time of the instances must also be considered.

Figure 7 shows the measured times after launching 20 jobs into empty job queues of compute environment in AWS Batch. The first box displays the time taken by the scheduler to launch an instance since a job is received. This time, which averages 166.8 s, indicates that executions delegated to AWS Batch are likely to have a substantially longer start-up time than on FaaS platforms. Depending on the requirements of the application to be deployed, it could be advisable to adjust the compute environment to keep one instance up and running, although this would have a negative impact on the deployment cost and the main advantage of the serverless paradigm would be lost.

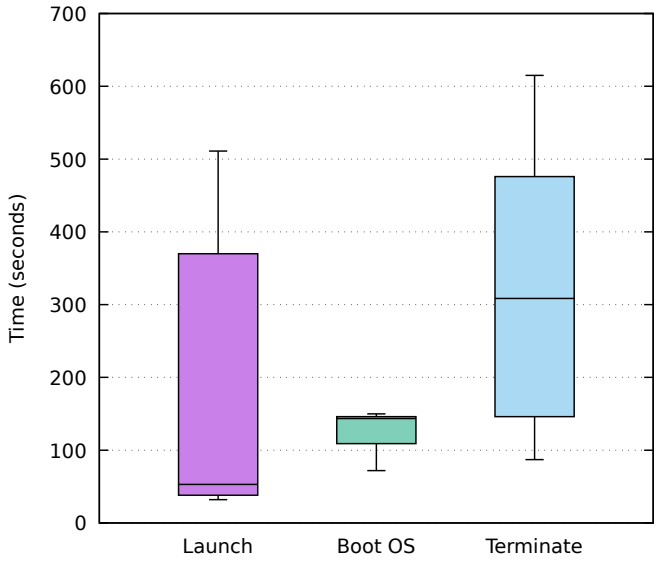


Figure 7. Launch, boot OS and terminate time of instances on AWS Batch.

The second box shows the boot time of the EC2 instance's operating system, which is on average 129 s with low standard deviation. This happens because the vendor manages compute environments that always run the same Amazon ECS-optimised AMI and only handles the ECS container agent startup as well as the download of the FaaS Supervisor component from the *cloud-init* user data.

Lastly, the time taken by the AWS Batch scheduler to scale down to zero the number of instances when the job queue becomes empty is displayed in the third box. This time, just like the launch one, depends on the internal operation of the scheduler and represents an additional cost to the actual usage of the machines. However, managed services such as AWS Batch represent a viable platform for sporadic executions of long-running, resource-intensive accelerated jobs.

5.4. Workflow Execution

With the aim of testing the platform, the use case defined in Section 4 was deployed on AWS. Both the FFmpeg and *audio2srt* functions were configured with the *lambda-batch* execution mode in order to support videos of variable duration, since they could exceed the maximum running time of AWS Lambda just as shown in Section 5.1. The instance

type chosen for their compute environments was `m3.medium` (the default in the SCAR configuration) due to its reduced on-demand cost ($0.067 per hour) and the lack of need for GPU acceleration. The memory allocated for both functions was 1024 MB as a result of the analysis of the applications involved. Although it is true that the functions could execute with less RAM, this amount was decided due to the Lambda linear allocation of CPU proportional to the memory. The same amount of memory plus 1 vCPU was specified for the job definition in AWS Batch.

The YOLOv3 video processing function, however, was defined with the *batch* execution mode and the `g3s.xlarge` instance type, as indicated in Section 5.2. Since these instances have a single GPU, several of them will be needed to process different videos in parallel. Therefore, to test the scaling of multiple VMs without incurring excessive costs, a maximum of three instances was determined for the compute environment. The job definition specification associated to this function was adjusted to take up all the resources of a `g3s.xlarge` instance (4 vCPU and 30.5 GB), considering that only one job can be scheduled at a time. Notice that in the *batch* execution mode, AWS Lambda is used as the entry point for events that are then delegated to AWS Batch. These intermediate functions are therefore also priced according to the assigned memory and running time. In this case, 128 MB of memory was selected to avoid unnecessary cost. The average running time obtained in these job-delegating functions was 1230ms, which can be considered negligible in the total processing budget.

The experiment carried out consisted in processing ten four-minute videos with a resolution of 1280 × 720. These videos were uploaded to the S3 *start* folder in order to trigger the execution of the workloads. At the starting point, a single video is uploaded and, after a minute, another one. Five minutes after the beginning, three videos are uploaded simultaneously and, to finish, another five videos are uploaded at minute ten of the test. A summary of the overall execution takes place in Figure 8. The FFmpeg (purple) and *audio2srt* (green) functions have been completely processed on AWS Lambda, taking advantage of their high parallelism for file processing. Furthermore, the visualisation of the YOLOv3 function, which runs entirely on AWS Batch, has been divided into two categories: the time that jobs remain pending in the job queue is shown in blue and, when they are processed on an EC2 instance, in yellow. As shown in the figure, after the autoscaling, up to three jobs are processed in parallel. This parallelism can be easily increased by configuring the compute environment to host a larger number of instances, thus allowing the platform to be customised according to the needs of the application preventing the user from explicitly managing the underlying computing infrastructure.

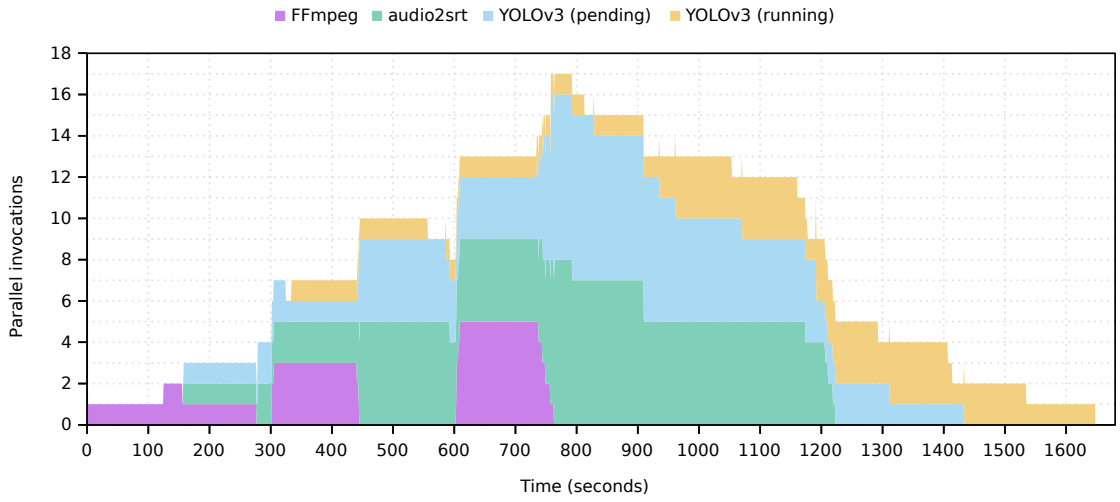


Figure 8. Time chart for the processing of ten videos using the workflow. Parallel invocations of the functions appear stacked. For the YOLOv3 function (running on AWS Batch) the pending and running states are distinguished.

Table 1 shows the running costs of the workflow for processing the first video, differentiating between the costs generated by AWS Lambda and the EC2 instance launched by AWS Batch. Amazon CloudWatch and S3 costs have been omitted since their low usage for this case study is covered by the AWS Free Tier. Similarly, AWS Lambda offers 400,000 GB-seconds of compute time per month, which would not incur costs if the platform does not exceed that threshold. Furthermore, as mentioned in Section 5.2, the cost per processing on GPU-enabled instances on AWS Batch is lower than if CPUs were used, since the processing time is reduced. As a result, the platform enables the deployment of serverless workflows in a cost-effective manner under a pay-per-use model.

Table 1. Cost analysis of the first workflow execution distinguishing between the two AWS services used for the processing.

	AWS Lambda	**AWS Batch/EC2**	**Total**
FFmpeg	$0.00255510	-	$0.00255510
audio2srt	$0.00758180	-	$0.00758180
YOLOv3	$0.00000258	$0.04687500	$0.04687758
Workflow	$0.01013948	$0.04687500	$0.05701448

Notice that this framework allows to create GPU-enabled data-driven serverless workflows that require no infrastructure preprovision and that are deployed at zero cost when the service is not being used. This rapidly and automatically scales upon uploading a file to the bucket, up to the limits defined by the workflow creator. This flexibility paves the way for increased adoption of event-driven scalable computing for multimedia and scientific applications.

6. Conclusions and Future Work

This paper has described the extension of the SCAR framework to support GPU-enabled serverless workflows for efficient data processing across diverse computing infrastructures. By combining the use of both AWS Lambda, for the execution of a large number of short jobs, and AWS Batch, for the execution of resource-intensive GPU-enabled applications, an open-source event-driven managed platform has been developed to create scale-to-zero serverless workflows. To test its performance, a case study has been defined and deployed on AWS. The behaviour of the platform, along with an analysis of deep learning inference applications running on GPUs and CPUs in the cloud has been exposed, highlighting the contributions of this study. The developments have been released as an open-source contribution to the SCAR tool, publicly available to reproduce the results described in this paper.

Future works involve the integration of the developed platform with on-premises serverless providers, as well as further extending the semantics of the Functions Definition Language (FDL) to accommodate additional workflow operators, thus allowing the definition of enhanced hybrid serverless workflows. In order to avoid failed executions of functions in AWS Lambda when applications reach the timeout and to find the most suitable allocation of memory for the *lambda* and *lambda-batch* execution modes, we consider integrating SCAR with a preprofiling tool such as *AWS Lambda Power Tuning* [37]. In addition, we plan to incorporate external data sources for long-term persistence outside AWS, such as the EGI DataHub [38].

Author Contributions: Conceptualisation, G.M.; methodology, S.R. and G.M.; software, S.R.; validation, S.R.; investigation, S.R.; resources, G.M.; data curation, S.R.; writing—original draft preparation, S.R. and G.M.; writing—review and editing, S.R. and G.M.; visualisation, S.R.; supervision, G.M.; project administration, G.M.; funding acquisition, G.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" with reference number TIN2016-79951-R.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Amazon Web Services. AWS Lambda. Available online: https://aws.amazon.com/lambda/ (accessed on 26 November 2020).
2. Amazon Web Services. Amazon Simple Storage Service (S3). Available online: https://aws.amazon.com/s3/ (accessed on 26 November 2020).
3. Amazon Web Services. Amazon API Gateway. Available online: https://aws.amazon.com/api-gateway/ (accessed on 26 November 2020).
4. Pérez, A.; Moltó, G.; Caballer, M.; Calatrava, A. Serverless computing for container-based architectures. *Future Gener. Comput. Syst.* **2018**, *83*, 50–59. [CrossRef]
5. Amazon Web Services. AWS Batch. Available online: https://aws.amazon.com/batch/ (accessed on 26 November 2020).
6. Jonas, E.; Pu, Q.; Venkataraman, S.; Stoica, I.; Recht, B. Occupy the cloud: Distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing—SoCC '17, Santa Clara, CA, USA, 25–27 September 2017; ACM Press: New York, NY, USA, 2017; pp. 445–451. [CrossRef]
7. Giménez-Alventosa, V.; Moltó, G.; Caballer, M. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Gener. Comput. Syst.* **2019**, *97*, 259–274. [CrossRef]
8. Malawski, M.; Gajek, A.; Zima, A.; Balis, B.; Figiela, K. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Gener. Comput. Syst.* **2017**. [CrossRef]
9. Jiang, Q.; Lee, Y.C.; Zomaya, A.Y. *Serverless Execution Of Scientific Workflows*; Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Springer: Berlin/Heidelberg, Germany, 2017; Volume 10601 LNCS, pp. 706–721. [CrossRef]
10. Skluzacek, T.J.; Chard, R.; Wong, R.; Li, Z.; Babuji, Y.N.; Ward, L.; Blaiszik, B.; Chard, K.; Foster, I. Serverless Workflows for Indexing Large Scientific Data. In Proceedings of the 5th International Workshop on Serverless Computing (WOSC '19), Davis, CA, USA, 9–13 December 2019; Association for Computing Machinery (ACM): New York, NY, USA, 2019; pp. 43–48. [CrossRef]
11. Chard, R.; Skluzacek, T.J.; Li, Z.; Babuji, Y.N.; Woodard, A.; Blaiszik, B.; Tuecke, S.; Foster, I.T.; Chard, K. Serverless Supercomputing: High Performance Function as a Service for Science. *CoRR* **2019**, arXiv:1908.04907.
12. Akkus, I.E.; Chen, R.; Rimac, I.; Satzke, M.S.K.; Beck, A.; Aditya, P.; Hilt, V. SAND: Towards high-performance serverless computing. In Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, 11–13 July 2018; pp. 923–935.
13. Sethi, R.J.; Gil, Y.; Jo, H.; Philpot, A. Large-Scale Multimedia Content Analysis Using Scientific Workflows. In Proceedings of the 21st ACM International Conference on Multimedia, Barcelona, Spain, 21–25 October 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 813–822. [CrossRef]
14. Gil, Y.; Ratnakar, V.; Kim, J.; Gonzalez-Calero, P.A.; Groth, P.; Moody, J.; Deelman, E. Wings: Intelligent Workflow-Based Design of Computational Experiments. *IEEE Intell. Syst.* **2011**, *26*. [CrossRef]
15. Deelman, E.; Vahi, K.; Juve, G.; Rynge, M.; Callaghan, S.; Maechling, P.J.; Mayani, R.; Chen, W.; Ferreira da Silva, R.; Livny, M.; et al. Pegasus: A Workflow Management System for Science Automation. *Future Gener. Comput. Syst.* **2015**, *46*, 17–35. [CrossRef]
16. Xu, X.; Chen, Y.; Yuan, Y.; Huang, T.; Zhang, X.; Qi, L. Blockchain-based cloudlet management for multimedia workflow in mobile cloud computing. *Multimed. Tools Appl.* **2020**, *79*, 9819–9844. [CrossRef]
17. Zhang, M.; Zhu, Y.; Zhang, C.; Liu, J. Video processing with serverless computing. In Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video—NOSSDAV '19, Amherst, MA, USA, 21 June 2019; ACM Press: New York, NY, USA, 2019; pp. 61–66. [CrossRef]
18. Pérez, A.; Caballer, M.; Moltó, G.; Calatrava, A. A programming model and middleware for high throughput serverless computing applications. In *Proceedings of the ACM Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 106–113. [CrossRef]
19. Amazon Web Services. Amazon CloudWatch. Available online: https://aws.amazon.com/cloudwatch/ (accessed on 26 November 2020).
20. Docker. Docker Hub. Available online: https://hub.docker.com/ (accessed on 26 November 2020).
21. Google Cloud. Cloud Computing Services. Available online: https://cloud.google.com/ (accessed on 26 November 2020).
22. Microsoft Azure. Cloud Computing Services. Available online: https://azure.microsoft.com/en-us/ (accessed on 26 November 2020).

23. Amazon Web Services. Amazon EC2. Available online: https://aws.amazon.com/ec2/ (accessed on 26 November 2020).
24. Amazon Web Services. Amazon ECS. Available online: https://aws.amazon.com/ecs/ (accessed on 26 November 2020).
25. NVIDIA. NVIDIA Container Runtime. Available online: https://github.com/NVIDIA/nvidia-container-runtime (accessed on 2 December 2020).
26. Amazon Web Services. AWS SDK for Python. Available online: https://aws.amazon.com/sdk-for-python/ (accessed on 26 November 2020).
27. Gomes, J.; Bagnaschi, E.; Campos, I.; David, M.; Alves, L.; Martins, J.; Pina, J.; López-García, A.; Orviz, P. Enabling rootless Linux Containers in multi-user environments: The udocker tool. *Comput. Phys. Commun.* **2018**, *232*, 84–97. [CrossRef]
28. Canonical. Cloud-Init: The Standard for Customising Cloud Instances. Available online: https://cloud-init.io/ (accessed on 2 December 2020).
29. FFmpeg. FFmpeg—A Complete, Cross-Platform Solution to Record, Convert and Stream Audio and Video. Available online: https://www.ffmpeg.org/ (accessed on 26 November 2020).
30. RunasSudo. audio2srt. Available online: https://gitlab.com/RunasSudo/audio2srt (accessed on 26 November 2020).
31. Shmyrev, N. CMUSphinx Open Source Speech Recognition. Available online: http://cmusphinx.github.io/ (accessed on 26 November 2020).
32. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.* **2016**, *2016*, 779–788. [CrossRef]
33. Redmon, J.; Farhadi, A. YOLOv3: An Incremental Improvement. *CoRR* **2018**, arXiv:1804.02767.
34. Redmon, J. Darknet: Open Source Neural Networks in C. Available online: https://pjreddie.com/darknet/ (accessed on 26 November 2020).
35. NVIDIA. CUDA Zone. Available online: https://developer.nvidia.com/cuda-zone (accessed on 26 November 2020).
36. Agache, A.; Brooker, M.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.M. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), Santa Clara, CA, USA, 2 January 2020; USENIX Association: Berkeley, CA, USA, 2020; pp. 419–434.
37. Casalboni, A. AWS Lambda Power Tuning. Available online: https://github.com/alexcasalboni/aws-lambda-power-tuning (accessed on 26 January 2021).
38. Viljoen, M.; Dutka, Ł.; Kryza, B.; Chen, Y. Towards European Open Science Commons: The EGI Open Data Platform and the EGI DataHub. *Procedia Comput. Sci.* **2016**, *97*, 148–152. [CrossRef]