On-premises Serverless Computing for Event-Driven Data Processing Applications

Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, Germán Moltó
Instituto de Instrumentación para Imagen Molecular (I3M).
Centro mixto CSIC - Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, España
Email: alpegon3@upv.es, gmolto@dsic.upv.es,{serisgal,dnaranjo,micafer1}@i3m.upv.es

***Abstract*—The advent of open-source serverless computing frameworks has introduced the ability to bring the Functions-as-a-Service (FaaS) paradigm for applications to be executed on-premises. In particular, data-driven scientific applications can benefit from these frameworks with the ability to trigger scalable computation in response to incoming workloads of files to be processed. This paper introduces an open-source framework to achieve on-premises serverless computing for event-driven data processing applications that features: i) the automated provisioning of an elastic Kubernetes cluster that can grow and shrink, in terms of the number of nodes, on multi-Clouds; ii) the automated deployment of a FaaS framework together with a data storage back-end that triggers events upon file uploads; iii) a service that provides a REST API to orchestrate the creation of such functions and iv) a graphical user interface that provides a unified entry point to interact with the aforementioned services. Together, this provides a framework to deploy a computing platform to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers that run on an elastic Kubernetes cluster. The usefulness of this framework is exemplified by means of the execution of a data-driven workflow for optimised object detection on video. The workflow is tested under three different workloads which process ten, a hundred and a thousand functions. The results show that the presented architecture is able to process such workloads taking advantage of its elasticity to make a sensible usage of the resources.**

***Index Terms*—Cloud Computing; Scientific Computing; Distributed Infrastructures; Containers; Docker;**

I. Introduction

Cloud computing has introduced the ability to provide a wide variety of well-known service models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The increased levels of automation together with the advent of lightweight workload isolation mechanisms introduced by Linux containers paved the way for the rise of the Functions as a Service (FaaS) service model. This approach involves invoking user-defined functions, coded in certain supported programming languages in response to events that are triggered in a computing and data storage infrastructure and that are typically run on a customized execution environment provided by a Linux container. Although sometimes used interchangeably, FaaS typically represents a subset of Serverless computing, a trend based on creating application architectures that entirely rely on Cloud services that provide automated resource provisioning on behalf of (and transparent to) the user. Thus, by not explicitly managing servers, developers can focus on the definition of the application logic instead of devoting time to infrastructure provision, configuration and scalability. The SPEC Cloud Group [1] defines three key features of serverless cloud architectures: i) granular billing: the user is only charged when the application is running; ii) minimal operational logic: the Cloud provider is responsible for resource management and autoscaling and iii) event-driven: short-lived execution of functions in response to events.

Public Cloud providers include in their portfolios services to support FaaS. As an example, Amazon Web Services (AWS) provides AWS Lambda, a service that can run thousands of parallel invocations to user-defined functions in response to multiple sources of events, such as an HTTP request, a file upload to an Amazon S3 bucket (an object-based data store) or a message sent to Amazon SQS, a service to create elastic message queues. Notwithstanding the large elasticity, several important limitations are currently exhibited by AWS Lambda. The maximum execution time is restricted to 15 minutes; the ephemeral storage space available to the invocations of a Lambda function is restricted to 512 MB and, finally, the runtime environments are pre-defined depending on the programming language used to code the Lambda functions.

Previous work from the authors introduced SCAR (Serverless Container-aware ARchitectures) [2] a framework to transparently execute containers out of Docker images in AWS Lambda, in order to run generic applications on that platform (for example image and video manipulation tools such as ImageMagick and FFmpeg or deep learning frameworks such as Theano and Darknet) and code in virtually any programming language (for example Ruby, R, Erlang and Elixir). This allowed to introduce a High Throughput Computing model [3] to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers run on AWS Lambda. SCAR provided a convenient approach to run generic applications on AWS Lambda and was rapidly adopted by the community

The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" under grant reference TIN2016-79951-R. This development is partially funded by the EGI Strategic and Innovation Fund and by the Primeros Proyectos de Investigación (PAID-06-18), Vicerrectorado de Investigación, Innovación y Transferencia de la Universitat Politècnica de València (UPV), València, Spain.

(more than 400 stars on GitHub) and it is featured in the Cloud Native Computing Foundation's Serverless Landscape [4].

However, the limited ephemeral storage in AWS Lambda and the rise of on-premises FaaS offerings revealed the need to support scalable event-driven computing for data-processing scientific applications in an on-premises environment. To this aim, this paper introduces OSCAR (Open Source Serverless Computing for Data-Processing Applications)[1] a framework to efficiently support on-premises FaaS (Functions as a Service) for general-purpose file-processing computing applications. The goal is to facilitate the adoption of event-driven computation for scientific applications that require processing data files. This is achieved by providing users with the ability to self-deploy an scalable integrated platform to be accessed by simple graphical user interfaces to define and manage the complete life cycle of functions that will be efficiently triggered upon users uploading files to specified folders.

We aim to abstract away the details concerning the definition of jobs to be executed, the sources of events, the management of the resource contention, the management of the job outputs and, specially, the deployment of the entire platform across multi-Clouds, including both on-premises Cloud Management Platforms (CMP) such as OpenNebula [5] and OpenStack [6] and public Cloud providers as well. Users should be able to upload input files through the web browser which triggers the execution of the functions to process the files. Output file results are made available to the user to be retrieved using the web browser.

After the introduction, the reminder of the paper is structured as follows. First, section II introduces the related work in the area. Next, section III provides an overview of the architecture of the OSCAR framework. Then, section IV presents an use case related to object detection in video in order to test the scalability and reliability of the platform. Section V presents the results obtained from the case-study and finally, section VI summarises the main contributions of this paper and presents the future works.

II. Related Work

Serverless computing pursues the adoption of dynamic elasticity handled by the Cloud provider for data-driven and compute-driven applications. Few works can be found in the literature concerning the application of serverless computing in the field of scientific computing, mainly attributed to the relatively young age of these computational approaches.

One of the pioneer works in this area is the paper by Jonas et al. [7] which introduced the PyWren framework in order to perform Python-based distributed computing on AWS Lambda. This simplifies the access to distributed computing by avoiding to provision and configure complex clusters and, instead, define stateless functions to be run on the Cloud. The authors extended the previous work in the contribution by Shankar et al. [8] in order to introduce *numpywren* a system for linear algebra that runs on AWS Lambda. They also introduced LAmbdaPACK a domain-specific language to implement linear algebra algorithms that are highly parallel, assessing the increased compute efficiency achieved and highlighting the limitations of the Cloud provider. In fact, the work by Spillner et al. [9] assesses the benefits of adopting serverless computing for multiple scientific domains: mathematics, computer graphics, cryptology and meteorology, using both public Cloud providers and self-hosted FaaS runtimes. The work by Giménez-Alventosa et al. [10] uses AWS Lambda to execute highly-parallel MapReduce jobs.

Authors such as Baldini et al. [11] address the problem of function composition entirely performed by serverless functions. Indeed, they demonstrate that function composition in serverless applications is achievable but exhibit several constraints to be considered such as avoiding double billing, adopting a substitution principle and treating the functions as black boxes.

The work by Adam Eivy [12] warns about the economic benefits of serverless computing, which strongly depends on the usage patterns and application workloads. Even though the pricing of services such as AWS Lambda are billed in the fraction of 100ms of execution time, these can rapidly add up to surpass the cost of traditional computing approaches involving virtual machines or even dedicated hardware.

Bringing the benefits of event-driven serverless computing, especially concerning FaaS, to on-premises environments has paved the way for multiple open-source FaaS frameworks to appear. Some examples are OpenFaaS [13], Knative [14], Kubeless [15], Fission [16], and Nuclio [17]. These platforms support the definition and execution of functions in response to events and they typically vary in the degree of support to multiple source of events, their support to programming languages and the usage of a certain Container Orchestration Platform, such as Kubernetes. There can also be found in the literature works related to serverless computing with these kind of frameworks. This is the case of the work by Hendrickson et al. [18] in which the authors introduce OpenLambda, an open-source platform for building serverless applications on-premises based on Linux containers. The authors further evolve this platform to accommodate lean microservices that depend on large libraries that start slowly and have an impact on elasticity. For this, they introduce Pipsqueak, a package-aware computing platform based on OpenLambda.

Indeed, the work by Baldini et al. [19] identifies several challenges related to serverless computing that are tackled by our proposed work. First, the ability to use *declarative approaches* to control what is deployed and the required tools to support it. We use a high-level declarative language to define the deployment of the OSCAR framework in order to achieve reproducible deployments across multi-Clouds. Second, the *support for long running jobs*, by adopting the Kubernetes container orchestration platform in order to manage the execution of the jobs in response to the events, but conveniently supplemented with an elasticity module to support the horizontal elasticity of the nodes of the cluster. The work by Erwin van Eyk et al. [1] also identifies some

[1]OSCAR - https://github.com/grycap/oscar

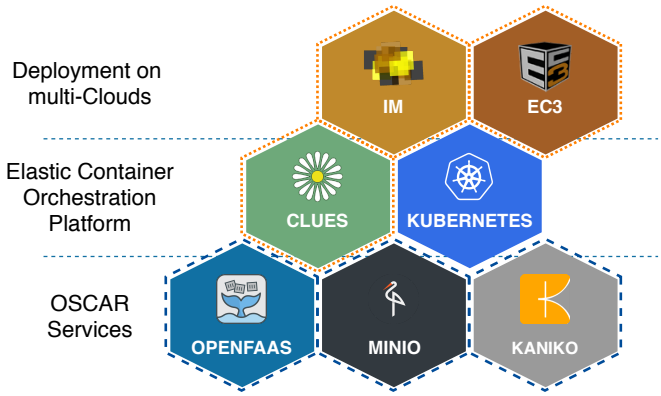

Fig. 1. Main components used in the OSCAR architecture. The orange dotted line marks the components that allow elasticity at the level of virtual machines (i.e. powering on and off nodes). The blue dashed line delimits the components that provide the functions as a service and the event programming model.

perspectives regarding the direction of the serverless field. They highlight the need to support hybrid Clouds, where an application could be composed by functions deployed on on-premises clusters, executing proprietary code, while other parts of the application could be run on the FaaS service offered by a public Cloud provider. For this, the ability to deploy on-demand an event-driven FaaS platform on top of an elastic Kubernetes cluster dynamically provisioned from an on-premises Cloud represents a step forward towards this aim.

III. The OSCAR framework

This section provides an overview of the architecture of the OSCAR framework, together with the open-source existing developments integrated in the platform, as well as the open-source developments carried out in order to produce such integrated platform.

A. Architecture of OSCAR

Figure 1 provides a high-level overview of the open-source components that are used in the OSCAR framework. In order to facilitate the deployment of the elastic Kubernetes cluster configured with all the services we rely on these tools:

- *Infrastructure Manager (IM)* [20], an open-source tool to describe complex application architectures using high-level declarative languages such as RADL (Resource Application Description Language) [21] and the standard specification TOSCA (Topology and Orchestration Specification for Cloud Applications) [22] in order to deploy them on multiple back-ends such as public Clouds: AWS, Microsoft Azure, Google Cloud Platform and on-premises Cloud Management Platforms: OpenNebula and OpenStack.
- *CLUster Elasticity System (CLUES)* [23], an open-source modular elasticity system that supports a wide variety of plugins in order to introduce elasticity capabilities for cluster-based computing. Many plugins are supported in order to introduce horizontal elasticity for different types of clusters: i) based on an LRMS (Local Resource Management System), supporting SLURM and PBS/Torque; ii) based on a Container Orchestration Platform, supporting Apache Mesos, Kubernetes and Nomad and iii) based on High Throughput Computing (HTC), supporting HTCondor.
- *Elastic Cloud Computing Cluster (EC3)* [24], [25], an open-source tool to deploy through the IM elastic compute clusters on multi-Clouds that can scale in/out in terms of the number of nodes according to certain elasticity rules defined in the corresponding CLUES plugin.

The aforementioned components are currently being used in production in the EGI Federated Cloud [26], a federated IaaS Cloud, composed of academic private clouds and virtualised resources and built around open standards, whose development is driven by the requirements of the scientific communities [27].

In addition, the open-source software developments that are used by OSCAR are:

- *OpenFaaS*. A framework for building serverless functions with Docker and Kubernetes.
- *Minio*. An object storage server that features an Amazon S3 compatible API.
- *Kaniko*. A tool to build container images from a Dockerfile, inside a Kubernetes cluster.

Figure 2 provides an overview of the interaction among the services deployed in the elastic Kubernetes cluster dynamically provisioned by the OSCAR framework.

OSCAR Manager is the service that provides the orchestration of the other services. It offers a REST API that allows the user to initialize and invoke functions. The process to create a function is completely transparent to the user and is comprised of the following steps: 1) Using the web interface, the user generates a request to create a function that is received by the OSCAR Manager service; 2) The required Docker image is created by Kaniko using as base image the user's image, in order to inject a supervisor in charge of managing the input data required and the output data generated by the application execution. Once the Kaniko build is finished, the image is registered in the Docker registry deployed in Kubernetes; 3) The required input/output buckets are created in Minio; 4) The function is created in OpenFaaS. OpenFaaS uses as image for the function the image created by Kaniko and retrieves it from the Docker registry.

OpenFaaS is designed to process short-lived requests and, therefore, attempting to execute several long running jobs at the same time could end up collapsing the cluster due to the lack of resources for all the processes. To be able to support long running jobs we developed a new service (i.e. OSCAR worker in Fig 2) that transforms asynchronous requests sent to OpenFaaS into Kubernetes jobs. Thus, the steps taken when executing a function are as follows. First, when the user uploads a file to a Minio bucket, an event is triggered and sent to OpenFaaS as an asynchronous request which is stored in the NATS queue provided by OpenFaaS.

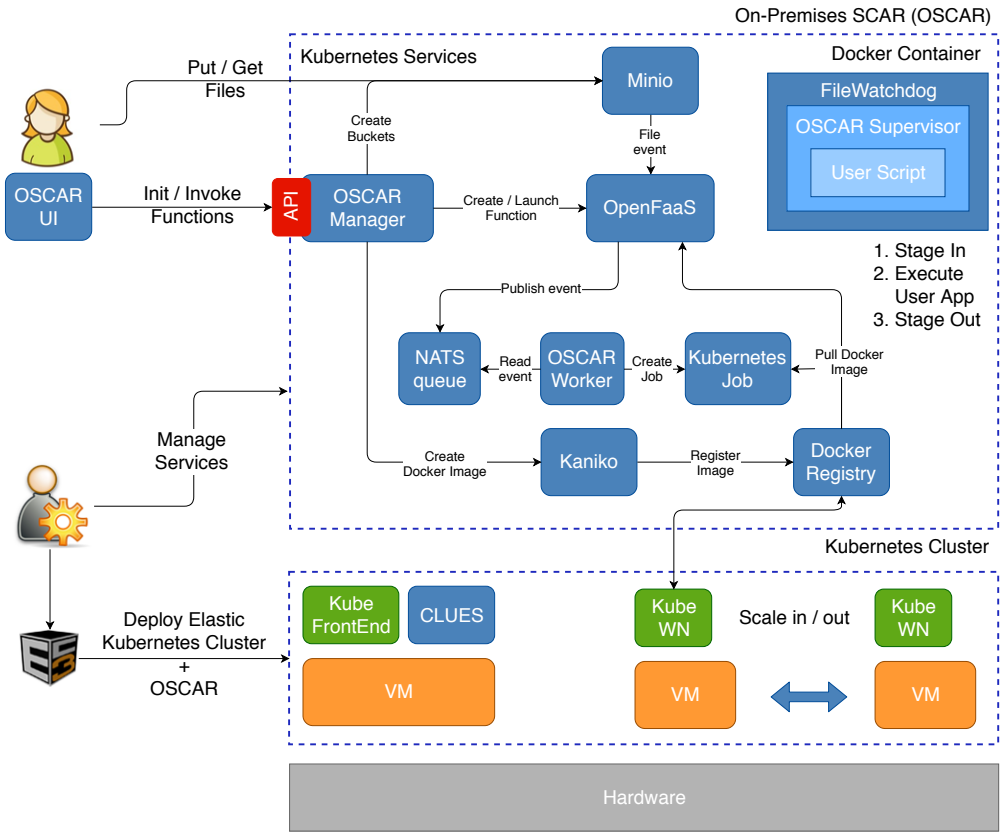

Fig. 2. Architectural approach for supporting container-based file-processing applications on serverless platforms.

Second, the OSCAR worker (that is subscribed to the NATS queue) reads the asynchronous request and then creates and submits a new job to Kubernetes. Submitting a job to Kubernetes allows OSCAR to delegate the resource management to both Kubernetes and the CLUES elasticity system. CLUES detects when the Kubernetes cluster needs more resources and provisions new nodes accordingly. Likewise, if CLUES detects that Kubernetes has spare nodes that are no longer needed, it terminates them.

For the development of the OSCAR UI (User Interface), VueJS and Vuetify were used. Both are accessible and versatile frameworks for building user interfaces. VueJS is a progressive JavaScript framework, with intuitive, modern and easy to use features, and has a very active community. Vuetify is a semantic component framework for VueJS. It aims to provide clean, semantic, and reusable components.

The graphical user interface is deployed inside the Kubernetes cluster, so it is necessary to externally expose the application through a port. Since VueJS is a frontend framework, and the application is executed on the client side, it was necessary to create a Node.js application that interacts with other internal services of the Kubernetes cluster such as OSCAR Manager, Minio, and OpenFaaS. The application was created using Express which is a robust, fast and flexible framework for Node.js applications.

Figure 3 depicts the Functions tab where you can create, edit or delete functions and Figure 4 shows the Storage tab where the information of the buckets is shown, as well as the stored files. In the Storage tab users can upload the files to be processed, remove them from the buckets or download the output files generated by a function.

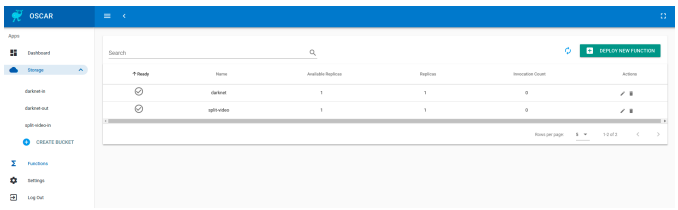

Fig. 3. OSCAR graphical user interface. The functions tab shows the user the functions created and the function status.

All the aforementioned components are dynamically provisioned inside the Kubernetes cluster via the corresponding Ansible Roles made available as open-source contributions in GitHub[2]

IV. Case Study - Video Processing Service

In order to assess the OSCAR framework we are going to deploy a serverless video processing service in an on-premises OpenNebula-based Cloud. This service is comprised of two

[2]GRyCAP GitHub organization: https://github.com/grycap

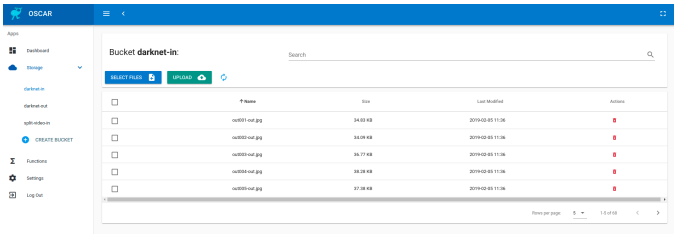

Fig. 4. OSCAR graphical user interface. The storage tab shows the user the Minio buckets automatically created when the function is deployed in the infrastructure and the stored files inside those buckets.

functions linked by means of an storage bucket, so when the first function finishes, the second function is triggered automatically. The goal of this architecture is to apply object recognition to the frames of the video uploaded by the user as input. Figure 5 shows the workflow of the architecture proposed. The files needed to reproduce the case study are open-source and available in GitHub[3].

The video processing function uses the *ffmpeg* library to extract the keyframes from the input video. To be able to create different workloads, the keyframe extraction rate has been changed to generate different amounts of images. The image processing function uses the *darknet* framework in combination with the YOLOv3 library [28] to detect the objects in the image. All the libraries and frameworks have been compiled to support CPU-based executions.

The following points explain the steps taken during the experiment execution:

1) Using the OSCAR UI the user creates the video processing and the image processing functions. By defining the output bucket of the video processing function as the input bucket of the image processing function the user is creating the workflow that is going to be triggered when a file is uploaded into the input bucket. Bear in mind that the creation of the required containers, the container registration in the internal Docker registry, the creation of the needed Minio buckets, and the creation of the OpenFaaS function is automatically performed without any user interaction.
2) Through the OSCAR UI, the user uploads to the input bucket of the video function the video to process. After this step the user interaction is not required anymore until the retrieval of the output data.
3) After the video upload finishes, Minio creates an event that is pushed to OpenFaaS which stores the event received in the NATS queue. Afterwards, the OSCAR worker reads the NATS queue and launches the function as a Kubernetes job. During the function execution, the OSCAR supervisor library that is deployed inside the function retrieves the video from the Minio bucket and stores it inside the pod. Then, the user script is executed and the generated output is stored in an specified output folder. As a last step in the execution of the video

[3]https://github.com/grycap/oscar/tree/master/examples/video-process

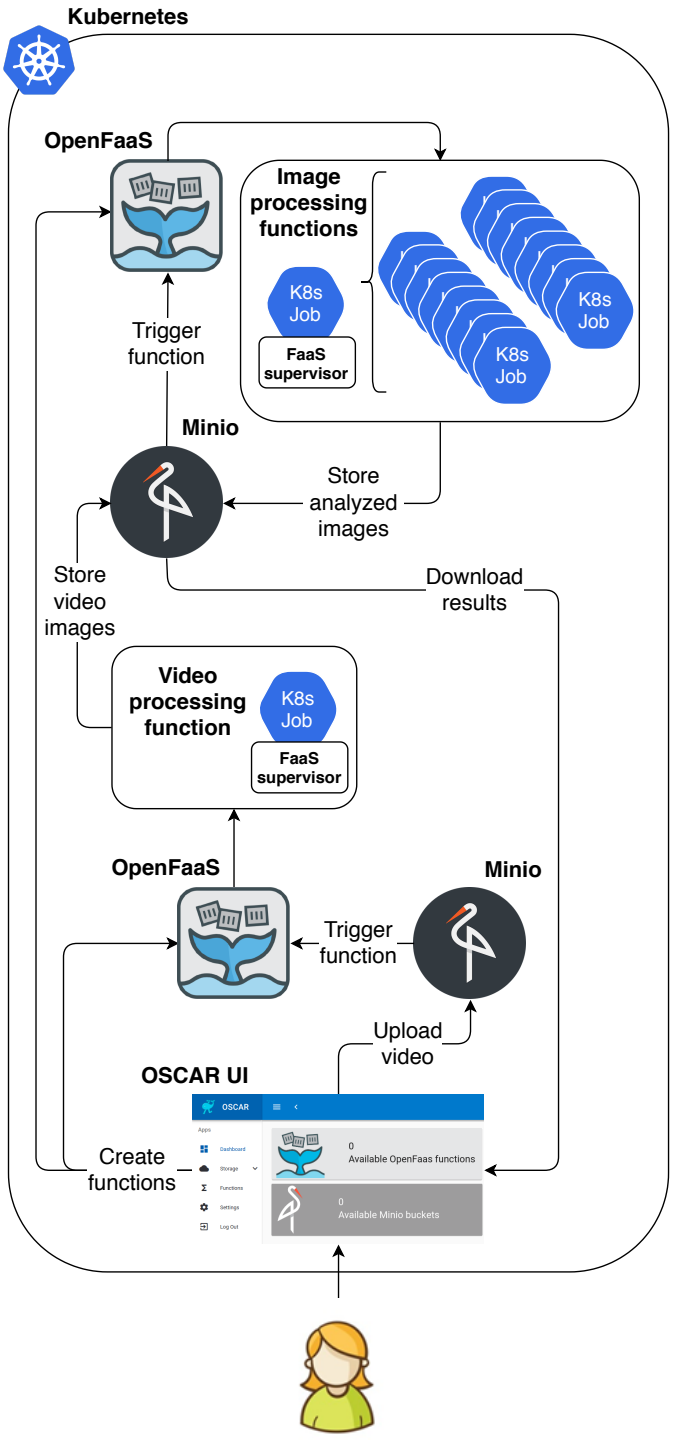

Fig. 5. Simplified workflow of the video processing service. Two OpenFaaS functions are used to process the video. First, a function to extract the video keyframes, and second, a function to analyze the generated keyframes. One function with the image processing functionality is triggered automatically for each keyframe created by the video processing function.

function, the OSCAR supervisor uploads all the files present in the output folder to the output bucket defined, thus triggering the image processing function.

4) Minio detects the new files in the input bucket and starts the process again but this time triggering the image processing function. Minio is going to generate an event for each file uploaded to the bucket, so we automatically end up with a function being launched for each image stored in the input bucket. The invocation and execution process of the function is the same as in the previous step but changing the container and the script executed. After the image processing function finishes, the output files generated are stored in the output bucket linked to the function.
5) The last step involves the user downloading the files generated by all the executed functions.

V. Results

To test the scalability and reliability of the infrastructure proposed we used three different workloads based on the number of images extracted from each video. Thus, for the first workload we extracted 10 images, 100 for the second workload and 1000 for the third. These workloads involved the invocation of 11, 101 and 1001 functions respectively in the OSCAR cluster, one for the video processing and the remaining for the image processing.

The specifications for the virtual cluster used in the case study are the following: the front-end has 8 virtual CPUs and 16 GiB of RAM. The working nodes have 4 virtual CPUs and 8 GiB of RAM. The specifications were selected to simulate two of the most common instances used to process compute-intensive workloads in Amazon Web Services. The front-end is equivalent to a c5.2xlarge and the nodes are equivalent to a c5.xlarge. The complete virtual cluster is composed by 1 front-end and a maximum of 10 nodes which will be powered on on-demand. All the machines used are virtual machines deployed in an OpenNebula-based on-premises cloud.

Figure 6 shows the state of the nodes during the execution of the three proposed workloads. The colors of the areas represent the following: the blue area represents the nodes that are idle (i.e. waiting for jobs); the dark orange area represents the nodes that are busy processing Kubernetes jobs; the grey area shows the number of nodes that are powering on and the gold area shows the number of nodes that are powering off. The data in the graph is stacked, thus the areas that have no color represent nodes that are powered off and are not consuming resources in the infrastructure.

In order to immediately process small workloads, the deployed cluster always has a working node available. This is represented by the orange area along the first 10 minutes in Figure 6. Afterwards, the first workload starts (i.e. process a video and ten extracted images). In the minute 10, the video is processed by the available node and the images to analyze are generated. CLUES realizes that it does not have the required resources to process the new function invocations and provisions additional nodes to process the incoming workload. This process can be seen in the first grey area in minute 12. The nodes in power on state take 3 minutes to deploy which is enough time for the already deployed working node to process the 10 jobs in the queue. Therefore, the new nodes that are deployed are in idle state and after a couple of minutes are powered off to save resources (this is represented by the gold area in minutes 14-16).

The second workload starts in minute 17. As in the first workload, the available working node is enough to process the uploaded video. This time 100 functions are generated, so the CLUES system powers on all the available nodes to attend the requests (this is represented by the second grey area between the minutes 18 and 26). The new nodes powered on start executing the functions just after being initialized so no idle nodes are seen until the functions allocated in those nodes are finished. The execution of the second workload finishes in minute 36 and it is represented by the highest peak of the idle area (i.e. the blue area).

With the third workload we wanted to test the reliability of the infrastructure under a high load. As stated at the beginning of the section, 1001 function invocations were launched and processed. In Figure 6 it can be seen that this was carried out between the minutes 37 and 78. In minute 37 all the working nodes that were idle receive new function invocations to process and the cluster continues processing them until minute 73 where the first working nodes start to be idle. After being idle for 5 minutes and not receiving new function invocations, CLUES started to power off nodes until only one working node is left.

Figure 7 shows the RAM memory and CPU usage of the nodes along the execution of the three workloads tested. The graph represent the stacked resources for each node and the green line represents the total amount of resources reserved in the cluster infrastructure. Since each function invocation has the same resource requirements and all the functions use the maximum resources available in each execution,the graphs of RAM and CPU usage can be combined. It is also important to know that no more than three functions per node could be deployed due to the image function requirements (i.e. 1 CPU and 2 GiB of RAM) and that Kubernetes also needs to deploy their own pods to control each node (those pods also use CPU and RAM resources). This behaviour caused that several GiB of RAM and CPUs were unused because the unused space in each node was less than the minimum space required by the functions and that is represented by the white area under the green line which is the total amount of resources.

As Figure 6, Figure 7 clearly depicts the execution of the three workloads. From minutes 10 to 12 there is a peak in resources consumption in the available working node. After the cluster finishes deploying new nodes this peak has disappeared (i.e. the execution of the functions has finished) and the new reserved resources (the working nodes) are freed again (terminated). The second workload starts in minute 17 and has its maximum peak of RAM and CPU consumption in minute 26 after all the nodes have been deployed. As the functions are processed, the usage of the resources of the working nodes

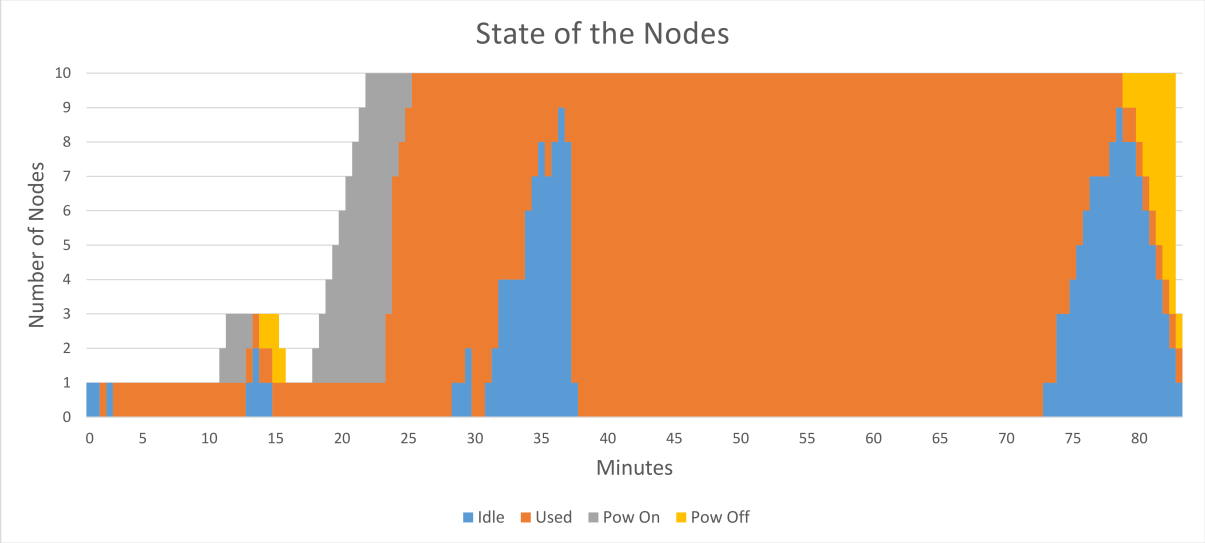

Fig. 6. State of the nodes during the execution of the three different workloads.

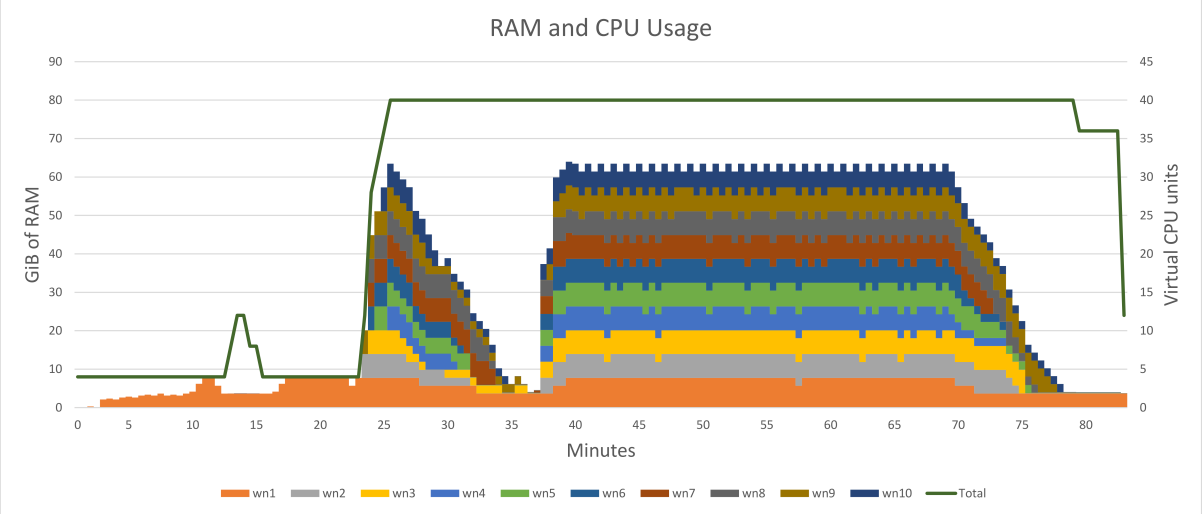

Fig. 7. RAM and CPU usage during the execution of the three different workloads (11, 101 and 1001 functions executed respectively).

decreases but the nodes are not terminated, thus not releasing the reserved infrastructure resources. The third workload starts in minute 37, then the maximum number of functions per node are deployed again and this behaviour continues until the 1000 images are processed in the minute 78. Once CLUES detects that the working nodes are idle for 5 minutes, it terminates them. The green line (total amount of resources reserved) at the end of the graph going down represents this release of resources.

As a summary of the results, the framework was able to process three different workloads, executing 11, 101 and 1001 functions. The first 11 function invocations were processed in 3 minutes and no extra nodes were needed (two new nodes were powered on but were never used). Processing the 101 function invocations made the cluster reach its top performance as seen in Figure 7 and it took 19 minutes to finish (including the deployment time of nodes which is 3 minutes). The third workload, processing 1001 functions, also fills all the available processing slots of the infrastructure's nodes. The third workload finishes in 41 minutes. This time could be improved by deploying a bigger cluster (e.g. 20 nodes instead of 10). Thanks to the elasticity of the cluster, these nodes would only be used when a high amount of function invocations are needed to be processed, being powered off the remaining time. To deploy a bigger cluster, the user only has to change the maximum number of nodes available when deploying a new cluster. The OSCAR framework will manage everything else to use those new resources.

VI. Conclusions and Future Work

This paper has introduced a framework to support serverless computing in on-premises platforms for event-driven data-

processing applications. First of all, a plugin to enable horizontal scalability of a Kubernetes cluster has been created, in order to cope with incoming workloads by provisioning additional virtual machines from the underlying Cloud computing platform employed. Second, the automated deployment and orchestration of the multiple services required to support this framework is performed with the help of the EC3 and IM tools, including a FaaS framework, an event-aware data storage back-end, and support for building and storing Docker images. Third, an integrated web-based graphical user interface is provided in order to simplify the interaction with the computing platform and that interacts with the services deployed inside the Kubernetes cluster.

Users are provided with an open-source platform offered via a convenient web interface that simplifies the creation and execution of the functions. The users just need to upload their files in order to trigger the concurrent execution of the application. The application will process the uploaded file and leave the output data files in the corresponding folder for the users to retrieve them. Being able to interact with a computing platform without requiring the definition of jobs using complex domain specific languages (DSLs), and by means of a web browser represents a step forward towards simplifying application execution for data-processing applications.

Finally, due to the resource requirements of the Kubernetes infrastructure, the RAM memory and CPU resources of the working nodes could not be completely used. Further work in the infrastructure refining the requirements and the behaviour of the required pods could lead to a better usage of the cluster resources and thus to a higher throughput when processing functions. Also future work involves integrating additional sources of events for multiple scientific storage back-ends such as Onedata or dCache. In addition, we plan to integrate OSCAR with SCAR in order to achieve event-driven hybrid serverless workloads across on-premises and public Clouds.

REFERENCES

[1] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC cloud group's research vision on FaaS and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*. New York, New York, USA: ACM Press, 2017, pp. 1–4. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3154847.3154848

[2] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, pp. 50–59, jun 2018. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167739X17316485

[3] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "A programming model and middleware for high throughput serverless computing applications," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '19. New York, NY, USA: ACM, 2019, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/3297280.3297292

[4] "CNCF Cloud Native Computing Foundation Interactive Landscape." [Online]. Available: http://s.cncf.io

[5] OpenNebula, "OpenNebula." [Online]. Available: https://opennebula.org

[6] OpenStack, "OpenStack." [Online]. Available: http://openstack.org

[7] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing - SoCC '17*. New York, New York, USA: ACM Press, 2017, pp. 445–451. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3127479.3128601

[8] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," oct 2018. [Online]. Available: https://arxiv.org/abs/1810.09679

[9] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: the prospect of serverless scientific computing and HPC," in *Communications in Computer and Information Science*, vol. 796. Springer, Cham, 2018, pp. 154–168. [Online]. Available: http://link.springer.com/10.1007/978-3-319-73353-1_11

[10] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS Lambda," *Future Generation Computer Systems*, mar 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18325172

[11] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: function composition for serverless computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*. New York, New York, USA: ACM Press, 2017, pp. 89–103. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3133850.3133855

[12] A. Eivy, "Be Wary of the Economics of Serverless Cloud Computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, mar 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7912239/

[13] "OpenFaaS." [Online]. Available: https://www.openfaas.com/

[14] "Knative." [Online]. Available: https://cloud.google.com/knative/

[15] "Kubeless." [Online]. Available: https://kubeless.io/

[16] "Fission." [Online]. Available: https://fission.io/

[17] "Nuclio." [Online]. Available: https://nuclio.io/

[18] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with openLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2016, pp. 33–39. [Online]. Available: https://dl.acm.org/citation.cfm?id=3027047http://dl.acm.org/citation.cfm?id=3027041.3027047

[19] 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," in *Research Advances in Cloud Computing*. Singapore: Springer Singapore, 2017, pp. 1–20. [Online]. Available: http://link.springer.com/10.1007/978-981-10-5026-8{_}1

[20] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso, "Dynamic Management of Virtual Infrastructures," *Journal of Grid Computing*, vol. 13, no. 1, pp. 53–70, mar 2015. [Online]. Available: http://link.springer.com/article/10.1007/s10723-014-9296-5http://link.springer.com/10.1007/s10723-014-9296-5

[21] "RADL." [Online]. Available: https://imdocs.readthedocs.io/en/latest/radl.html

[22] D. Palma, M. Rutkowski, and T. Spatzier, "TOSCA Simple Profile in YAML Version 1.1," Tech. Rep., 2016. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html

[23] C. de Alfonso, M. Caballer, A. Calatrava, G. Moltó, and I. Blanquer, "Multi-elastic Datacenters: Auto-scaled Virtual Clusters on Energy-Aware Physical Infrastructures," *Journal of Grid Computing*, jul 2018. [Online]. Available: http://link.springer.com/10.1007/s10723-018-9449-z

[24] M. Caballer, C. de Alfonso, F. Alvarruiz, and G. Moltó, "EC3: Elastic Cloud Computing Cluster," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1341–1351, dec 2013. [Online]. Available: http://authors.elsevier.com/sd/article/S0022000013001141http://linkinghub.elsevier.com/retrieve/pii/S0022000013001141

[25] A. Calatrava, E. Romero, G. Moltó, M. Caballer, and J. M. Alonso, "Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures," *Future Generation Computer Systems*, vol. 61, pp. 13–25, aug 2016. [Online]. Available: http://authors.elsevier.com/sd/article/S0167739X16300024

[26] E. Fernández-Del-Castillo, D. Scardaci, and Á. L. García, "The EGI Federated Cloud e-Infrastructure," in *Procedia Computer Science*, vol. 68. Elsevier, jan 2015, pp. 196–205. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705091503080X

[27] E. Foundation, "EGI Federated Cloud." [Online]. Available: https://www.egi.eu/federation/egi-federated-cloud/

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