UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Serverless Strategies and Tools in the Cloud Computing Continuum

November 2023

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science at the Universitat Politècnica de València

Author: Sebastián Risco Gallardo

Advisor: Dr. Germán Moltó Martínez

Acknowledgements

En primer lugar, me gustaría agradecer a Germán Moltó por confiar en mí desde el principio y darme la oportunidad de formar parte del grupo. Gracias por toda la ayuda durante estos años y la paciencia para que terminase esta tesis.

Gracias también a todos los compañeros del GRyCAP por la buena acogida y todos los buenos momentos que hemos pasado juntos en el laboratorio. En especial me gustaría agradecer a Miguel por la incontable ayuda que me ha proporcionado siempre; a Alfonso por dejarme coger el testigo de sus desarrollos y por esos primeros congresos juntos; a Carlos por todas las veces que me ha ayudado con los clusters; a Diana por su ayuda con la interfaz de OSCAR y esos artículos juntos; y a Caterina y Sergio, que aunque se unieron en la última etapa han sido fundamentales para el desarrollo del último artículo.

Para terminar, y como no podía ser de otra forma, gracias a mi familia y amigos por apoyarme y estar ahí siempre. Especialmente a mi padre, porque sin su ayuda y sacrificio nunca habría llegado hasta aquí.

Gracias a todos.

Abstract

In recent years, the popularity of Cloud computing has allowed users to access unprecedented compute, network, and storage resources under a pay-per-use model. This popularity led to new services to solve specific large-scale computing challenges and simplify the development and deployment of applications. Among the most prominent services in recent years are FaaS (Function as a Service) platforms, whose primary appeal is the ease of deploying small pieces of code in certain programming languages to perform specific tasks on an event-driven basis. These functions are executed on the Cloud provider's servers without users worrying about their maintenance or elasticity management, always keeping a fine-grained pay-per-use model.

FaaS platforms belong to the computing paradigm known as Serverless, which aims to abstract the management of servers from the users, allowing them to focus their efforts solely on the development of applications. The problem with FaaS is that it focuses on microservices and tends to have limitations regarding the execution time and the computing capabilities (e.g. lack of support for acceleration hardware such as GPUs). However, it has been demonstrated that the self-provisioning capability and high degree of parallelism of these services can be well suited to broader applications. In addition, their inherent event-driven triggering makes functions perfectly suitable to be defined as steps in file processing workflows (e.g. scientific computing workflows).

Furthermore, the rise of smart and embedded devices (IoT), innovations in communication networks and the need to reduce latency in challenging use cases have led to the concept of Edge computing. Edge computing consists of conducting the

processing on devices close to the data sources to improve response times. The coupling of this paradigm together with Cloud computing, involving architectures with devices at different levels depending on their proximity to the source and their compute capability, has been coined as Cloud Computing Continuum (or Computing Continuum).

Therefore, this PhD thesis aims to apply different Serverless strategies to enable the deployment of generalist applications, packaged in software containers, across the different tiers of the Cloud Computing Continuum. To this end, multiple tools have been developed in order to: i) adapt FaaS services from public Cloud providers; ii) integrate different software components to define a Serverless platform on on-premises and Edge infrastructures; iii) leverage acceleration devices on Serverless platforms; and iv) facilitate the deployment of applications and workflows through user interfaces. Additionally, several use cases have been created and adapted to assess the developments achieved.

Resumen

En los últimos años, la popularidad de la computación en nube ha permitido a los usuarios acceder a recursos de cómputo, red y almacenamiento sin precedentes bajo un modelo de pago por uso. Esta popularidad ha propiciado la aparición de nuevos servicios para resolver determinados problemas informáticos a gran escala y simplificar el desarrollo y el despliegue de aplicaciones. Entre los servicios más destacados en los últimos años se encuentran las plataformas FaaS (Función como Servicio), cuyo principal atractivo es la facilidad de despliegue de pequeños fragmentos de código en determinados lenguajes de programación para realizar tareas específicas en respuesta a eventos. Estas funciones son ejecutadas en los servidores del proveedor Cloud sin que los usuarios se preocupen de su mantenimiento ni de la gestión de su elasticidad, manteniendo siempre un modelo de pago por uso de grano fino.

Las plataformas FaaS pertenecen al paradigma informático conocido como Serverless, cuyo propósito es abstraer la gestión de servidores por parte de los usuarios, permitiéndoles centrar sus esfuerzos únicamente en el desarrollo de aplicaciones. El problema del modelo FaaS es que está enfocado principalmente en microservicios y tiende a tener limitaciones en el tiempo de ejecución y en las capacidades de computación (por ejemplo, carece de soporte para hardware de aceleración como GPUs). Sin embargo, se ha demostrado que la capacidad de autoaprovisionamiento y el alto grado de paralelismo de estos servicios pueden ser muy adecuados para una mayor variedad de aplicaciones. Además, su inherente ejecución dirigida por eventos hace que las funciones sean perfectamente adecuadas para ser definidas como pasos en flujos de trabajo de procesamiento de archivos (por ejemplo, flujos de trabajo de computación científica).

Por otra parte, el auge de los dispositivos inteligentes e integrados (IoT), las innovaciones en las redes de comunicación y la necesidad de reducir la latencia en casos de uso complejos han dado lugar al concepto de Edge computing, o computación en el borde. El Edge computing consiste en el procesamiento en dispositivos cercanos a las fuentes de datos para mejorar los tiempos de respuesta. La combinación de este paradigma con la computación en nube, formando arquitecturas con dispositivos a distintos niveles en función de su proximidad a la fuente y su capacidad de cómputo, se ha acuñado como continuo de la computación en la nube (o continuo computacional).

Esta tesis doctoral pretende, por lo tanto, aplicar diferentes estrategias Serverless para permitir el despliegue de aplicaciones generalistas, empaquetadas en contenedores de software, a través de los diferentes niveles del continuo computacional. Para ello, se han desarrollado múltiples herramientas con el fin de: i) adaptar servicios FaaS de proveedores Cloud públicos; ii) integrar diferentes componentes software para definir una plataforma Serverless en infraestructuras privadas y en el borde; iii) aprovechar dispositivos de aceleración en plataformas Serverless; y iv) facilitar el despliegue de aplicaciones y flujos de trabajo a través de interfaces de usuario. Además, se han creado y adaptado varios casos de uso para evaluar los desarrollos conseguidos.

Resum

En els últims anys, la popularitat de la computació al núvol ha permès als usuaris accedir a recursos de còmput, xarxa i emmagatzematge sense precedents sota un model de pagament per ús. Aquesta popularitat ha propiciat l'aparició de nous serveis per resoldre determinats problemes informàtics a gran escala i simplificar el desenvolupament i desplegament d'aplicacions. Entre els serveis més destacats en els darrers anys hi ha les plataformes FaaS (Funcions com a Servei), el principal atractiu de les quals és la facilitat de desplegament de petits fragments de codi en determinats llenguatges de programació per realitzar tasques específiques en resposta a esdeveniments. Aquestes funcions són executades als servidors del proveïdor Cloud sense que els usuaris es preocupen del seu manteniment ni de la gestió de la seva elasticitat, mantenint sempre un model de pagament per ús de gra fi.

Les plataformes FaaS pertanyen al paradigma informàtic conegut com a Serverless, el propòsit del qual és abstraure la gestió de servidors per part dels usuaris, permetent centrar els seus esforços únicament en el desenvolupament d'aplicacions. El problema del model FaaS és que està enfocat principalment a microserveis i tendeix a tenir limitacions en el temps d'execució i en les capacitats de computació (per exemple, no té suport per a maquinari d'acceleració com GPU). Tot i això, s'ha demostrat que la capacitat d'autoaprovisionament i l'alt grau de paral · lelisme d'aquests serveis poden ser molt adequats per a més aplicacions. A més, la seva inherent execució dirigida per esdeveniments fa que les funcions siguen perfectament adequades per ser definides com a passos en fluxos de treball de processament d'arxius (per exemple, fluxos de treball de computació científica).

D'altra banda, l'auge dels dispositius intel · ligents i integrats (IoT), les innovacions a les xarxes de comunicació i la necessitat de reduir la latència en casos d'ús complexos han donat lloc al concepte d'Edge computing, o computació a la vora. L'Edge computing consisteix en el processament en dispositius propers a les fonts de dades per millorar els temps de resposta. La combinació d'aquest paradigma amb la computació en núvol, formant arquitectures amb dispositius a diferents nivells en funció de la proximitat a la font i la capacitat de còmput, s'ha encunyat com a continu de la computació al núvol (o continu computacional).

Aquesta tesi doctoral pretén, doncs, aplicar diferents estratègies Serverless per permetre el desplegament d'aplicacions generalistes, empaquetades en contenidors de programari, a través dels diferents nivells del continu computacional. Per això, s'han desenvolupat múltiples eines per tal de: i) adaptar serveis FaaS de proveïdors Cloud públics; ii) integrar diferents components de programari per definir una plataforma Serverless en infraestructures privades i a la vora; iii) aprofitar dispositius d'acceleració a plataformes Serverless; i iv) facilitar el desplegament d'aplicacions i fluxos de treball mitjançant interfícies d'usuari. A més, s'han creat i s'han adaptat diversos casos d'ús per avaluar els desenvolupaments aconseguits.

Contents

List of Figures

Chapter 1

Introduction

In the last years, a revolution has emerged regarding access to computing adapted to the users' requirements through the well-known public Cloud providers, which allow renting access to computation, storage and networking in a straightforward way. This new model of computing access is known as Cloud computing and, in contrast to traditional hosting providers, allows users to pay for the actual use, on a fine-grained model, of the systems used, instead of hiring servers for long periods. As a result, companies can save the costs of purchasing and maintaining data centres and start offering their services quickly through the provider's infrastructure.

The traditional Cloud computing service model is known as IaaS (Infrastructure as a Service). In addition to the advantages introduced above, IaaS platforms allow users to dynamically adapt the provisioned services to their specific needs, i.e. they can increase or decrease the number or the capabilities of their servers depending on the expected workload at a specific time. However, in IaaS platforms, users must manage and configure the software of their virtual machines (operating system, network configuration, applications) themselves.

According to a forecast by the consulting company Gartner [80], the spending on public Cloud services worldwide was expected to grow by 20.4% in the year 2022, possibly reaching $600 billion. As can be appreciated in Figure 1.1 [187], the public Cloud computing market has continued to grow year on year, which in

turn has meant that public cloud providers have continued investing in technology to improve their services.

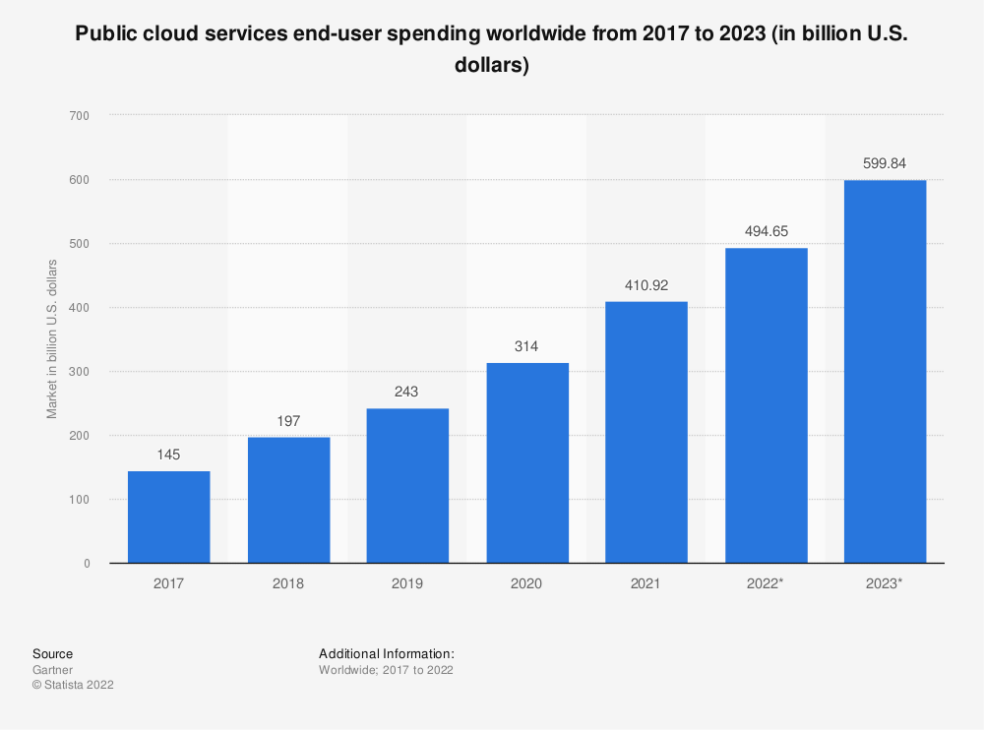


Figure 1.1: Public Cloud services market size 2017-2023.

This continuous growth of Cloud computing led to the emergence of new services in addition to the aforementioned IaaS platforms, usually under the models coined as PaaS (Platform as a Service) and SaaS (Software as a Service). On the one hand, the PaaS model is at a higher level of abstraction than IaaS, providing development tools and frameworks to programmers to develop applications to run directly on the provider's infrastructure, thereby preventing users from dealing with maintenance tasks and virtual machine management. On the other hand, the SaaS model is where applications are offered directly to end users (e.g. Netflix [138], Spotify [185] or Dropbox [70]). It is important to note that the key feature of all Cloud computing service models is elasticity, as users can always scale the capacity of their services on a fine-grained pay-as-you-go basis.

All these advances would not have been possible without the improvements in virtualisation over the last decades. Hypervisor support for hardware virtualisation

led to the widespread adoption of virtual machines as a mechanism for wrapping applications in shared servers. IaaS platforms make use of this technology to allow users to deploy virtual machines on demand in isolated environments within their data centres. However, despite the versatility offered by this paradigm, each virtual machine must have a complete operating system in order to function, which places a certain overhead on the use of system resources if its ultimate purpose is to serve a single application.

Due to the problem mentioned above, software containers emerged, which started to gain adoption in 2013 after the release of Docker [69]. Software containers are based on the capabilities of the Linux kernel to isolate processes (i.e. kernel namespaces, cgroups, etc.) [119], allowing applications and all their dependencies to be encapsulated in images. These images can run isolated on the same operating system, sharing the kernel, but without the ability to know what other processes are being executed on it. This technology rapidly became a de facto standard for the distribution of applications, leading to the emergence of container orchestration platforms, whose greatest example is Kubernetes [114], whose source code was released in 2014 by Google [98], and in 2015 it became part of the Cloud Native Computing Foundation (CNCF) [189].

Container orchestration platforms provide a way to associate a set of nodes, which can be physical (bare-metal) or virtual (VM-based), to construct clusters on which to deploy applications encapsulated in software containers. These platforms facilitate the definition of a wide range of rules to manage the lifecycle of applications, as well as the easy handling of multiple security, monitoring and high availability mechanisms. Such has been the adoption of these platforms that most public Cloud providers offer them as services, allowing users to deploy containers directly on them and pricing them according to the time usage and resources assigned to each container. An example of these services can be the self-managed Kubernetes platforms on the leading providers: Amazon Elastic Kubernetes Service (EKS) [20], Google Kubernetes Engine (GKE) [87] and Azure Kubernetes Service (AKS) [126].

Furthermore, taking advantage of the preceding advances in software container encapsulation of applications and isolation through microVMs [188], the public cloud provider Amazon Web Services announced the AWS Lambda service in November 2014 [105]. AWS Lambda steps into a new service model coined as FaaS (Function as a Service), whose core feature is the execution of pieces of code written in the vendor's supported programming languages directly on its infrastructure. These platforms are offered under a finer-grained pay-per-use model (currently priced per microsecond), where users only pay for the actual execution time. This represents a huge revolution over other service models, as applications

can be released at zero cost if they are not in use, thus entirely abstracting the management of the servers and allowing developers to focus exclusively on the development of their applications. Other public Cloud providers soon announced their own FaaS platforms, being Google Cloud Functions, Azure Functions and Alibaba Function Compute, among the most popular.

In contrast to the IaaS model, in FaaS, users are not concerned about deploying and managing virtual machines on which to serve applications but only add the code to be executed through the platform interface. Then, users only have to determine the event sources from other provider services that will trigger the execution of that code. Some event sources for invoking functions can be a file upload to a data storage system (such as Amazon S3 [8]), an HTTP request to a REST API defined in the provider (such as Amazon API Gateway [173]) or a message received in a queuing system (such as Amazon SQS [13]), among others. Moreover, the code of a function does not have to use vendor libraries to make use of the provider's infrastructure (unlike the PaaS model), so it is arguably less tied to the platform. This adds a higher level of abstraction within the other Cloud service models, as seen in Figure 1.2 [90].

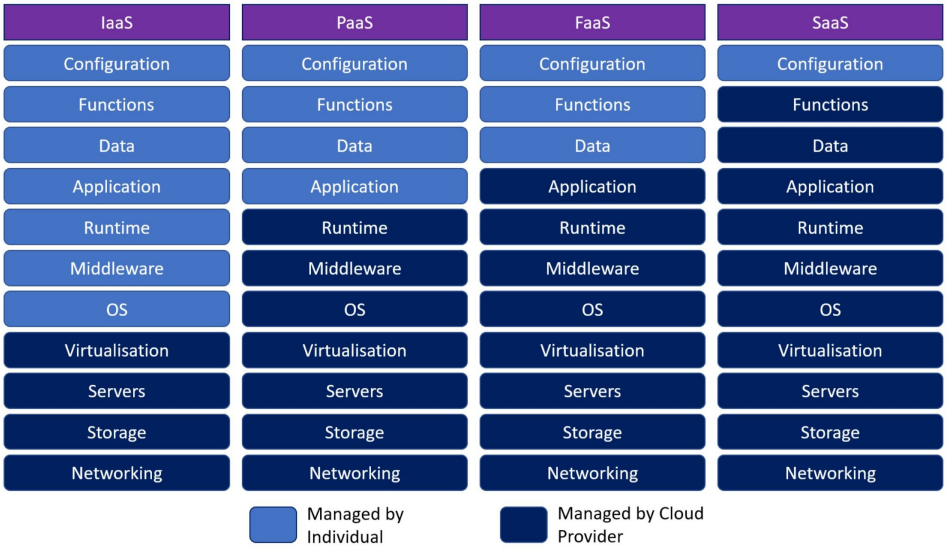


Figure 1.2: Comparison between the main Cloud computing service models.

FaaS platforms were quickly adopted by a large number of companies for the agile development of microservices-based applications [89] due to their ease of use and high parallelism capabilities. In fact, pioneering FaaS services in the area,

such as AWS Lambda, support the concurrent execution of 3000 functions by default [19], which means unprecedented elasticity with no need to configure any auto-scaling mechanism. Furthermore, in AWS Lambda, each processing instance consists of a microVM driven by the Firecracker software [18], which is capable of starting up in as little as 125 ms and ensures that each invocation of a function has the amount of resources allocated to it. This guarantees that, unlike traditional HTTP backends, each request always has a fixed resource allocation, preventing instances from overloading in the case of large load spikes.

This service model belongs to the so-called Serverless Computing. Serverless computing is a computing paradigm in which users do not have to worry about managing the underlying computational infrastructure of their applications. Although the term Serverless is often confused with FaaS, it is important to stress that FaaS are not the only application of this paradigm, which also encompasses other services, i.e. self-managed databases such as Amazon DynamoDB [17] or container orchestration systems managed by the Cloud provider, such as AWS Fargate [22].

Faced with such a disruption to the development and production deployment of applications, it was not long before open source Function-as-a-Service frameworks appeared. The main idea of these frameworks is to implement FaaS platforms on top of on-premises resources, usually exploiting the capabilities of container orchestration systems. Examples of these frameworks are OpenFaaS [144], Nuclio [139], Apache OpenWhisk [25] or Knative [112]. As a result, companies or institutions that already own their servers can benefit from these platforms, and thus save resources in the management and scalability of applications.

Moreover, open-source Function-as-a-Service frameworks also allow leveraging these platforms at other layers of the Computing Continuum. Due to the large number of lightweight devices connected to the internet nowadays, i.e. Internet of Things, there arises the need for low latency processing in several use cases. Precisely, the term Computing Continuum refers to the different computing layers depending on their proximity to the data source. As shown in Figure 1.3 [7], the closest level of processing is known as the Edge, which in many cases can be lightweight devices directly attached to the data acquisition devices (e.g. smartphones, embedded devices). At a higher level is what is referred to as Fog, which is at an intermediate level between the Edge and the Cloud, and could be exemplified as nearby data centres or small clusters of low-powered devices (e.g. Raspberry PI).

Indeed, having the advantages of serverless computing through FaaS platforms across the Computing Continuum would allow the easy development of workflows

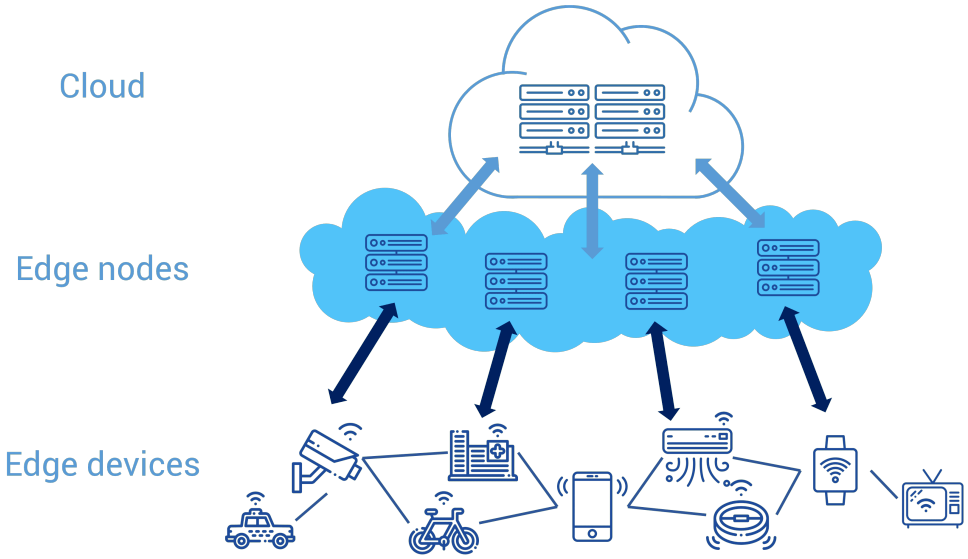


Figure 1.3: Simplified view of the Computing Continuum.

for massive data processing by entirely abstracting the infrastructure from the developed applications.

1.1 Motivation

Although the advances introduced by FaaS platforms have significant advantages for the development of applications based on microservices, it must be taken into account that they also have some significant limitations. For example, at the time these FaaS services appeared, most public Cloud providers only support a few programming languages, which makes it impossible to take advantage of these platforms to run existing applications coded in other languages. Furthermore, even though a Serverless function does not depend on any software library of the provider on which it is executed, the function code must follow a specific format to work properly, i.e. there must be a main method that receives the event that triggers its execution for further processing. This is why the Grid and High Performance Computing (GRyCAP) research group[1] of the Institute of Instrumentation for Molecular Imaging (I3M)[2] at the Polytechnic University of Valencia developed the SCAR tool at the end of 2017.

[1] `https://www.grycap.upv.es`
[2] `https://www.i3m.upv.es`

SCAR [97] [153] is a command-line application developed as the central part of Alfonso Pérez's doctoral thesis [6], whose primary purpose is to run software containers on AWS Lambda. To do so, it uses the *udocker* tool [84], written in Python, which allows running Docker containers in user space, i.e. without the need to install the application with superuser permissions. This allows running a wide range of scientific and generalist applications on the AWS Lambda platform. In addition, SCAR introduces an event-driven programming model [151] for automatic data processing with no need for users to adapt their applications. To this end, the *faas-supervisor* [92] component was developed, which is responsible for receiving the events generated by the Amazon S3 storage system, downloading the file that generated the event into the container, launching a user-defined script and, finally, uploading the resulting files to an output bucket.

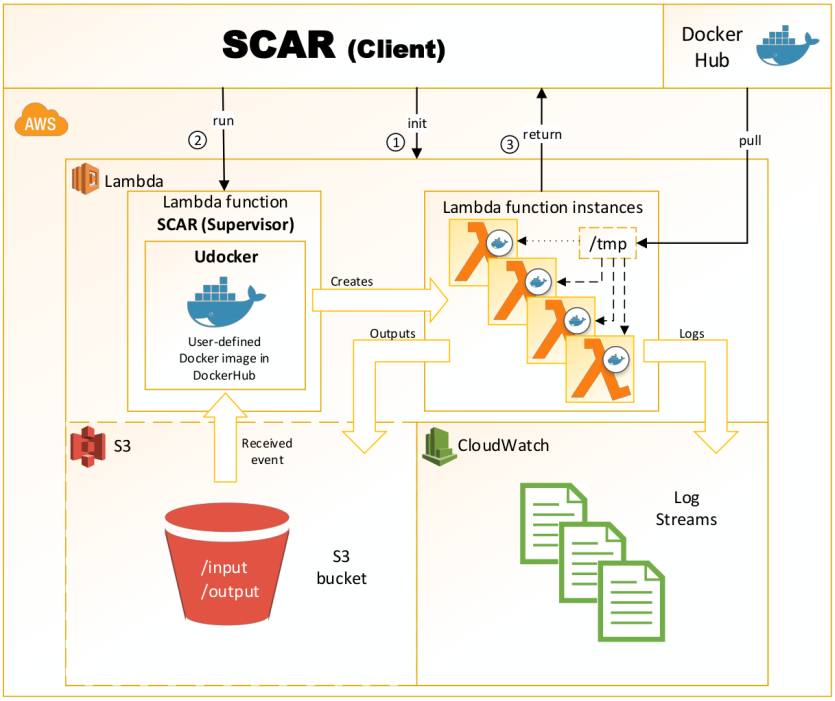


Figure 1.4: SCAR architecture.

However, although SCAR managed to overcome the limitations of AWS Lambda mentioned above, there are still some critical issues, such as the maximum execution time, which at the time of writing this thesis cannot exceed 900 seconds. In addition, the execution environment cannot exceed 512 MB in size in line with the

AWS Lambda restriction at that time. However, in 2021 AWS introduced native support for Docker containers stored in the Amazon ECR image registry system, supporting containers up to 10 GB [29]. The platform also does not support the use of hardware acceleration devices such as GPUs, widely used in the field of artificial intelligence.

That is why, in 2018, as part of the author's master's thesis [166], it was decided to develop an open-source platform for event-driven Serverless file processing. This platform, called OSCAR [95] [152], began as a prototype that integrated different open-source tools to emulate the behaviour of SCAR, but being able to run on top of the Kubernetes container orchestration system. This opened up the possibility of installing the platform not only on different public cloud providers, but also on on-premises IaaS platforms belonging to any company or research centre.

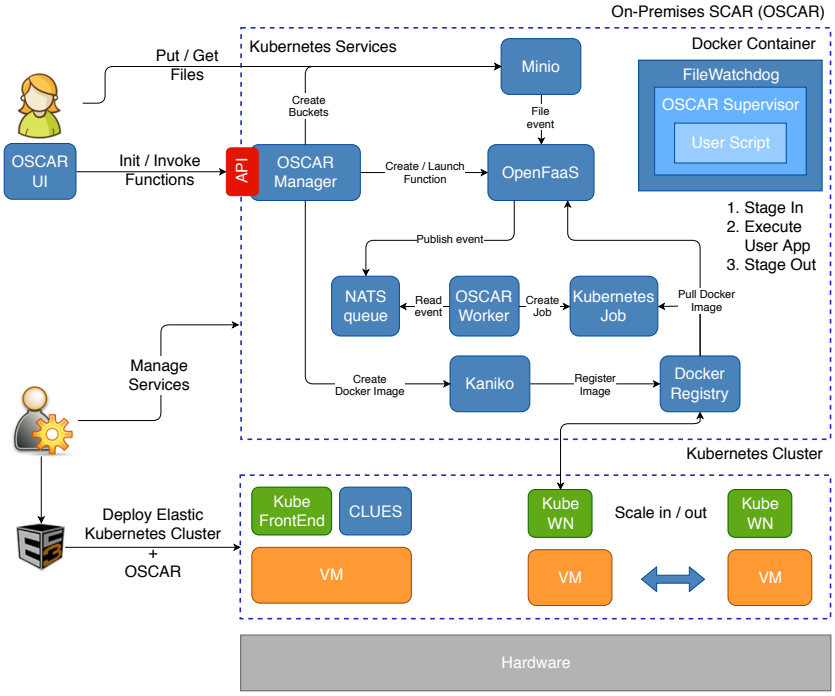


Figure 1.5: Initial OSCAR architecture.

Despite the enhancements introduced in the OSCAR platform, the prototype had to evolve to achieve greater scalability since, initially, it was a simple API written in Python whose primary purpose was the creation of functions based on OpenFaaS. Also, the Kaniko tool [88] was used to build the software containers

containing all the necessary components (i.e. *faas-supervisor* and user-defined script), which caused a slowdown in the function creation time. Therefore, one of the main gaps to be covered by this PhD thesis consists of improving the OSCAR platform, better integrating its main components, and increasing scalability and usability by leveraging different resources of the Kubernetes container orchestration platform itself in a native way. Another essential improvement for the platform would be enabling it to run on IoT devices located at the Edge of the Computing Continuum. For this purpose, it is intended to make use of a programming language capable of being easily compiled for different architectures, as well as the use of continuous delivery techniques and tools to automate the creation of multi-arch software containers for easy deployment.

Finally, it would be precious to integrate the SCAR tool and the OSCAR platform to facilitate the composition of serverless workflows across the Computing Continuum. This integration would represent a breakthrough in the state-of-the-art of serverless computing and open up a wide range of possibilities to run numerous modern use cases. To this aim, we establish the following objectives.

1.2 Objectives

The main objective of this thesis is to facilitate the deployment and execution of complex container-based Serverless applications across the different layers of the Computing Continuum. To this end, the aim is to evolve and integrate the software components presented in the previous section, enabling the definition of applications that exploit different strategies for Serverless computing. To facilitate the achievement of this objective, the following milestones are proposed and will be addressed during the development of the doctoral thesis:

- Define elastic execution models in hybrid platforms composed of public Serverless services (as is the case of AWS Lambda) and on-premises FaaS platforms that enable the automatic scaling of the computational infrastructure. It will build on SCAR and OSCAR developments by adding the required functionality to integrate public and on-premises deployments, with the possibility of sharing the same storage backend to centralise data storage. In addition, the OSCAR deployment will be adapted to run on lightweight Kubernetes clusters on top of IoT devices, based on the ARM architecture, in order to support data processing at the Edge of the Continuum.

- Produce a workflow description language that supports the expression of dependencies between event-driven computational tasks to be executed through a function invocation model (FaaS). This language will enable the composi-

tion of hybrid event-driven scientific pipelines that involve on-premises and public computational resources or exclusively public services, thus achieving the desired zero-scaling feature. In this way, a scientific application, composed of multiple computational steps, can be deployed in the Cloud at zero cost. Only when invoked the necessary infrastructure for its execution is automatically deployed and scaled.

- Study and integrate the support for hardware acceleration devices such as GPUs, adopting and extending existing support from container orchestration platforms (such as Kubernetes) or services managed by a public provider (such as AWS Batch [14]) applied to existing developments. This integration will support a broader range of use cases that require this acceleration for processing, such as inference applications through artificial intelligence models.

- Assess and develop delegation mechanisms for Serverless workloads across the different layers of the Computing Continuum. Using such strategies it is intended to balance the load between multiple clusters of the OSCAR platform and delegate the processing of events to FaaS services from public Cloud providers when faced with load peaks to reduce processing times in critical systems.

- Achieve the integration of the OSCAR platform in the EOSC (European Open Science Cloud) Marketplace[3], aiming to facilitate the solution's visibility and adoption among the European scientific community. For this purpose, services already integrated into the EOSC Marketplace that were previously developed in the research group will be used. This is the case of the Infrastructure Manager (IM)[4] [41] and the EGI Applications on Demand portal[5], where the EC3 tool[6] [42] is currently integrated to deploy virtual elastic clusters on multi-Clouds. Hence, it is necessary to define Ansible [161] roles and TOSCA [142] recipes for the automated deployment of OSCAR clusters.

- Adapt scientific use cases from different European research projects. On the one hand, it is intended to cooperate with the partners of the DEEP-Hybrid DataCloud project [66], in which the research group participated, to integrate the trained models in the OSCAR platform and thus be able to perform parallel inference in response to images uploaded by users. On

[3] `https://marketplace.eosc-portal.eu`
[4] `https://im.egi.eu`
[5] `https://www.egi.eu/services/`
[6] `https://grycap.upv.es/ec3`

the other hand, we aim to integrate the developments derived from this thesis in the AI-SPRINT project [186], which is focused on developing a secure platform for deploying AI-based applications through the Computing Continuum. This project includes several use cases, such as agriculture 4.0, medical data analysis and wind power plant maintenance through the processing of images captured by drones.

Finally, it is considered of utmost importance that all the developments resulting from this doctoral thesis are shared publicly through software repositories[7]. For this purpose, the Apache 2.0 public license [23] will be used, guaranteeing that any user can distribute and modify the software and facilitating the community's adoption and enhancement of the tools.

1.3 Organisation of this document

This dissertation comprises a compendium of research articles published during the development of this PhD in computer science. Therefore, after the introduction chapter, the four main chapters will comprise the articles resulting from the research conducted. The organisation of the document, made up of the following chapters, is detailed below:

- Chapter 2, "GPU-Enabled Serverless Workflows for Efficient Multimedia Processing", presents the evolution of the SCAR tool for composing Serverless workflows with support for GPU acceleration through its integration with the AWS Batch service. The paper includes a use case for event-driven multimedia processing, exemplified by audio and video analysis leveraging open-source AI-based applications for object recognition and automatic subtitle generation. This work [167] has been published in the JCR-indexed journal "Applied Sciences", which in the year of publication belonged to the second quartile (Q2).
- Chapter 3 "Serverless Workflows for Containerised Applications in the Cloud Continuum", introduces the primary outcomes of the development of the thesis. This article describes the redesign of the OSCAR platform, overcoming the drawbacks mentioned in section 1.1, and the integration with the SCAR tool through a common Functions Definition Language. As a result of this integration, Serverless workflows can be composed in the different layers of the Computing Continuum through OSCAR clusters deployed on ARM-based devices located on the Edge or in private clouds, as well as on

[7] `https://github.com/grycap`

public providers such as AWS Lambda, thanks to SCAR. To exemplify the use of such workflows, a use case is presented for face mask recognition, preprocessing the images to anonymise them in an OSCAR cluster and performing the processing in AWS Lambda. This work [168] has been published in the JCR-indexed journal "Journal of Grid Computing", which in the year of publication belonged to the first quartile (Q1).

- Chapter 4, "A serverless gateway for event-driven machine learning inference in multiple clouds", exemplifies the primary outcomes of the thesis through a gateway that integrates the OSCAR platform with the SCAR tool to serve AI-based applications from the European research project DEEP Hybrid-DataCloud. The article details the development of a Serverless-based web interface (through the use of different AWS services) that interacts with OSCAR and SCAR for easy file processing and visualisation of the results by scientific users. The work [137] has been published in the JCR-indexed journal "Concurrency and Computation: Practice and Experience", which in the year of publication belonged to the third quartile (Q3).

- Chapter 5, "Rescheduling Serverless Workloads Across the Cloud-to-Edge Continuum", presents the study and implementation of two strategies for delegating Serverless jobs from OSCAR clusters to other clusters or HTTP endpoints, enabling the integration with Lambda functions built with SCAR. The paper demonstrates how these two new strategies (Rescheduler and Resource Manager) work through a use case for fire detection from surveillance images with processing at different levels of the Continuum. At this manuscript publication date, the paper is submitted to the "Future Generation Computer Systems" journal.

- After the four main chapters composed of the articles published during this thesis, chapter 6 discusses the results obtained, detailing the contributions made by this PhD and adding other successful cases of the software developed in section 6.2.

- Finally, chapter 7 summarises the results obtained and outlines future work.

Chapter 2

GPU-Enabled Serverless Workflows for Efficient Multimedia Processing

Published as

Sebastián Risco and Germán Moltó. (2021) GPU-Enabled Serverless Workflows for Efficient Multimedia Processing. Applied Sciences, 1438 (11), 1 - 17. https://doi.org/10.3390/app11041438

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.

2.1 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 [15]. 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 [8] or invoking a REST API provided by API Gateway [9]). 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[1] [153] 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.

[1]SCAR: `https://github.com/grycap/scar`

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 [14], 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 2.2 describes the related work in this area. Then, Section 2.3 introduces the architecture of the system to support GPU-enabled serverless workflows for data processing. Next, Section 2.4 describes a use case to assess the benefits of the platform by supporting a serverless workflow for parallel audio and video processing. Later, Section 2.5 presents the results obtained after the execution of the use case. Finally, Section 2.6 summarises the main achievements of the paper pointing to future work.

2.2 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. [108], by introducing PyWren to execute distributed Python code across mul-

tiple Lambda function invocations to achieve a virtualized supercomputer. In addition, the work by Alventosa et al. [82] 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. [122] 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. [106] which integrates a combination of Functions as a Service (FaaS)/local clusters execution approach for Montage-based workflows. The work by Skluzacek et al. [181] 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. [49] 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. [3] 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. [174] already addressed the application of scientific workflows for multimedia content processing, leveraging the Wings [81] framework to efficiently analyse large-scale multimedia content across the Pegasus engine [61], which operates over cloud infrastructures. Another example of the adoption of cloud technologies in this area is the study conducted by Xu et al. [194], 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. [197] 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.

2.3 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.

2.3.1 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. [151]. 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 [10]. Docker container images are usually fetched from publicly available container registries such as Docker Hub [68]. 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 [86] or Microsoft Azure [125].

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 2.1 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 [153]. 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.

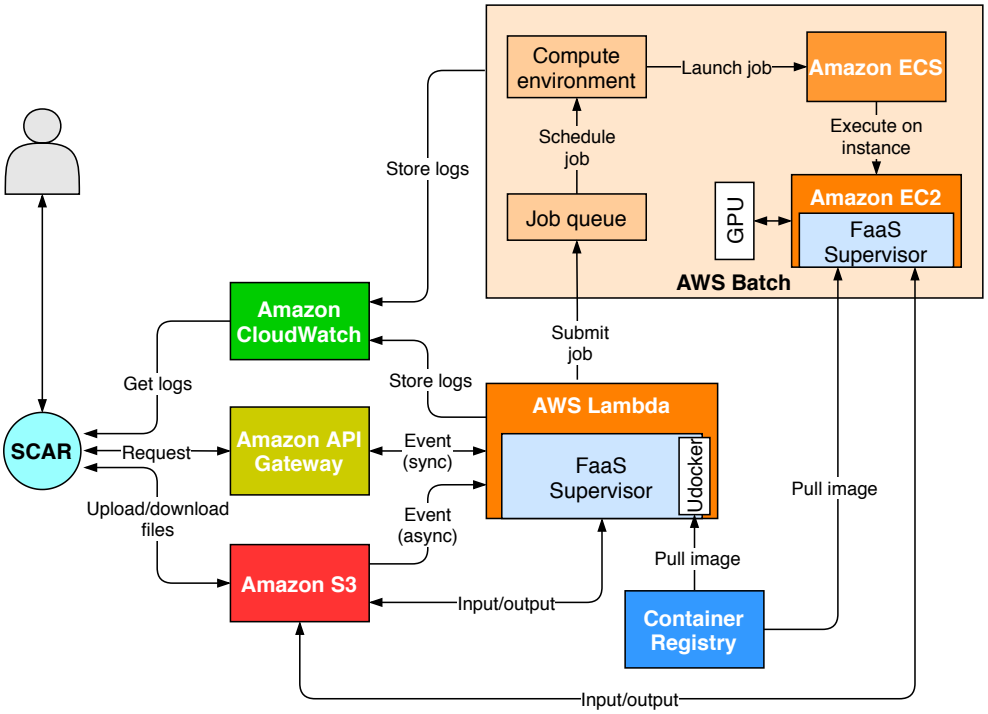


Figure 2.1: Architecture of the SCAR platform integrated with AWS Batch in order to support long-running and GPU-accelerated jobs.

AWS Batch runs Docker-based computational jobs on EC2 [12] 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 [11] 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) depend-

ing 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 [141].

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 2.5.3. 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.

2.3.2 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) [16]. Furthermore, it also required extending the *faas-supervisor*[2], 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

[2]FaaS Supervisor: `https://github.com/grycap/faas-supervisor`

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* [84] 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* [45] user data. Then, the path containing the FaaS Supervisor is mounted automatically as a volume on the job containers.

2.3.3 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 2.4.

A detailed example of the FDL shown in Figure 2.2, together with a test video and deployment manual are available in GitHub[3]. 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.

2.4 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

[3] *av-workflow* example: `https://github.com/grycap/scar/tree/master/examples/av-workflow`

```
---
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 2.2: Functions Definition Language example.

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 2.3 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* [75]. 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* [170]. A small application to generate subtitles from audio transcripts obtained through CMUSphinx [179], 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* [164, 163]. A real-time object detection system that, using the Darknet [162] neural network framework, can run on GPUs to accelerate the video inference process. It has been compiled with CUDA [140] 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.

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 2.4.

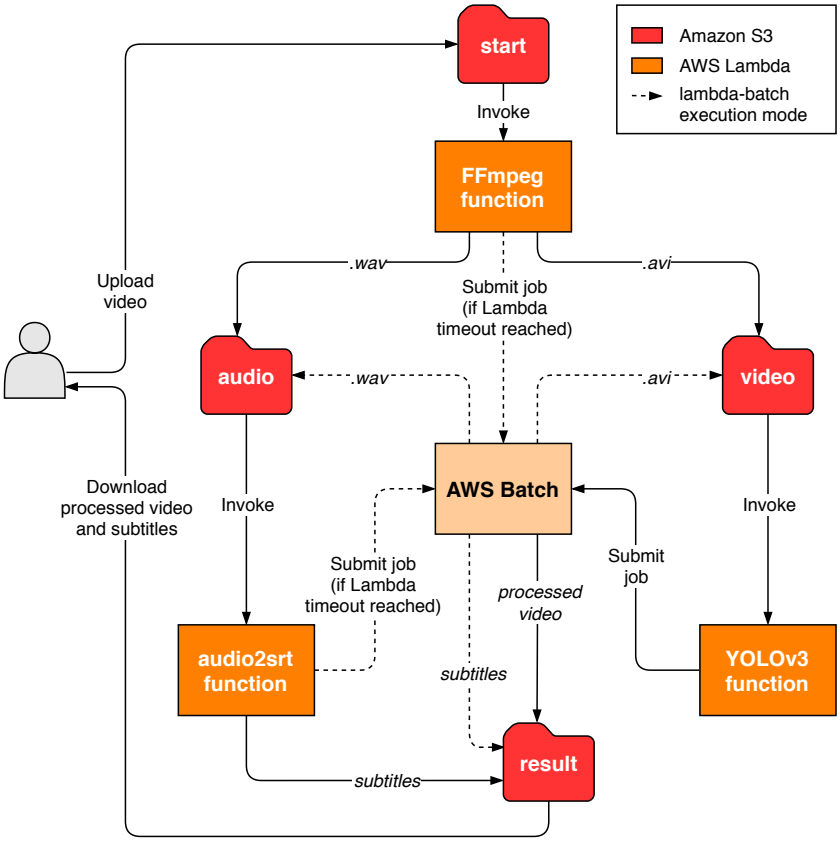


Figure 2.3: Simplified diagram of the multimedia processing workflow.

2.5 Results and Discussion

The experimentation carried out is divided into four differentiated stages. First, Section 2.5.1 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 2.5.2 compares different instance types to be used in the video object detection function with the aim of choosing the most efficient one. Then, Section 2.5.3 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 2.5.4 discusses the results of the execution of the serverless workflow for processing several videos.

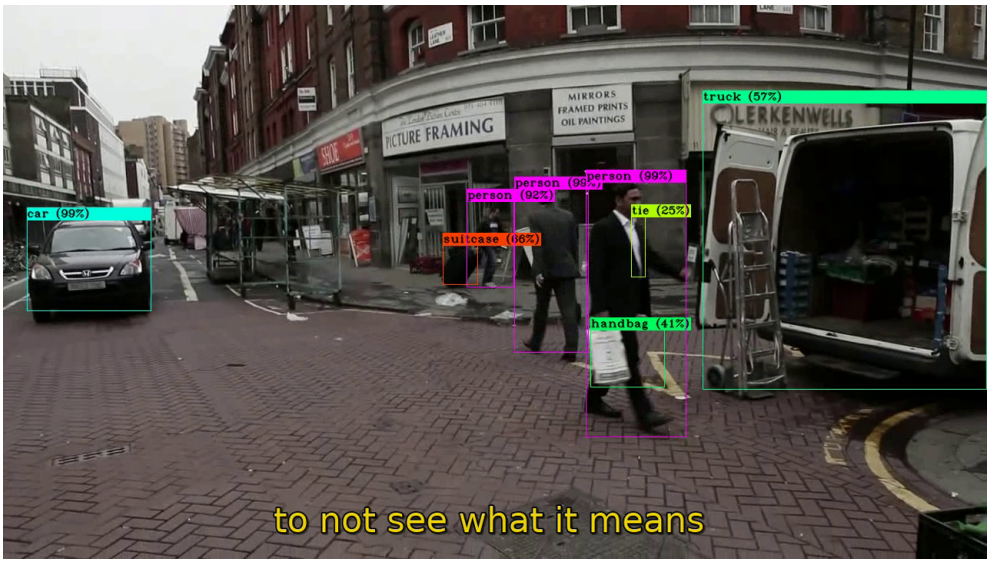


Figure 2.4: Snapshot of a video resulting from the object recognition function along with the automatically generated subtitles after a workflow execution.

2.5.1 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.

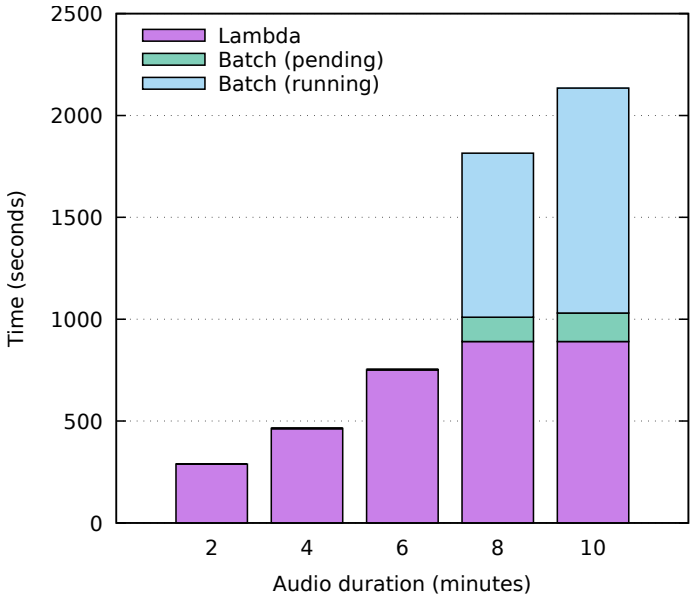
In the field of multimedia processing we can find different applications optimised to support accelerated computing or that directly require a execution time longer than 15 min. However, there may be uses in which the processing fits into AWS Lambda in most cases but eventually exceeds the execution 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

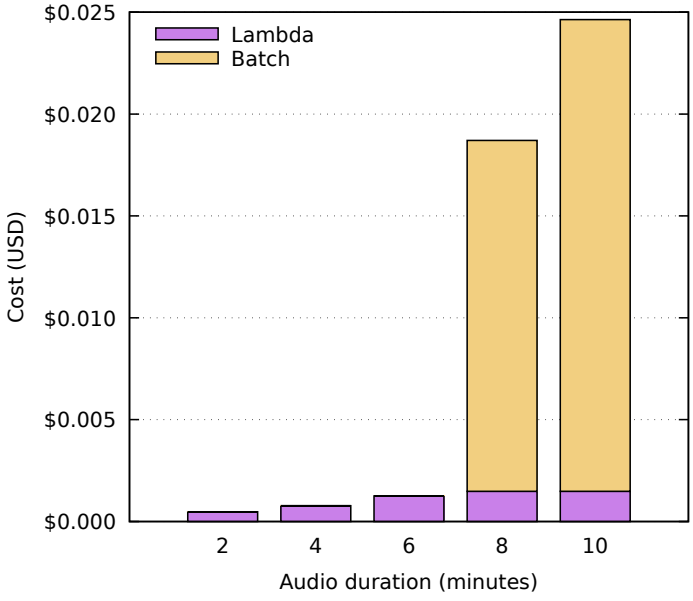
audio2srt function of the workflow presented in the previous section. This analysis consists of processing several waveform audio files of different lengths (i.e., 2, 4, 6, 8 and 10 min). 1 GB of RAM has been allocated for the function in both services (Lambda and Batch). The `m3.medium` instance type 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 2.5a 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 2.5.3. 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-minute 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 2.5.3. 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 2.5b 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



(**a**) Average processing times



(**b**) Average processing cost

Figure 2.5: Time and cost analysis of the *audio2srt* function for different audio durations.

mode, since the reduced start-up times provided by AWS Lambda can benefit these kind of applications.

2.5.2 GPU and CPU Comparison for Video Processing

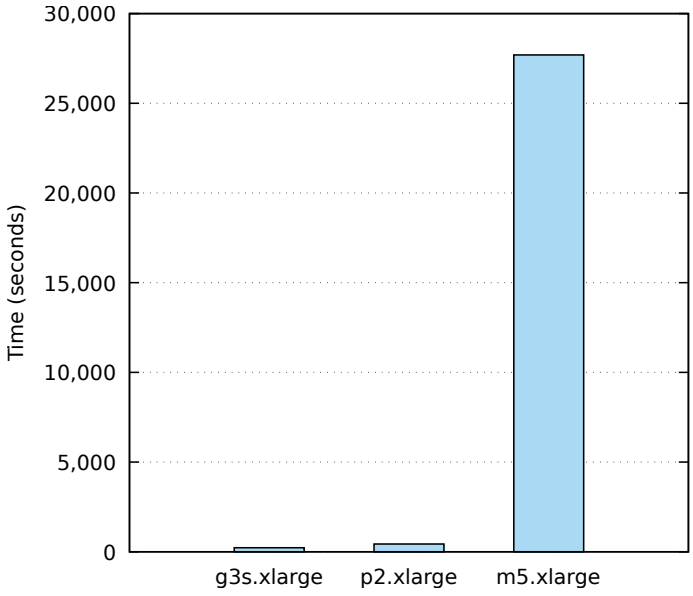
Before the experimentation of the whole workflow, the video processing function was evaluated in order to determine the instance type that would best suit the requirements of the application in terms of execution time and cost. To demonstrate that GPU acceleration is worthwhile in deep learning inference processes, a comparison was performed using different EC2 instances to process the same 4-minute video with a 1280x720 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 3.1GHz, 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.

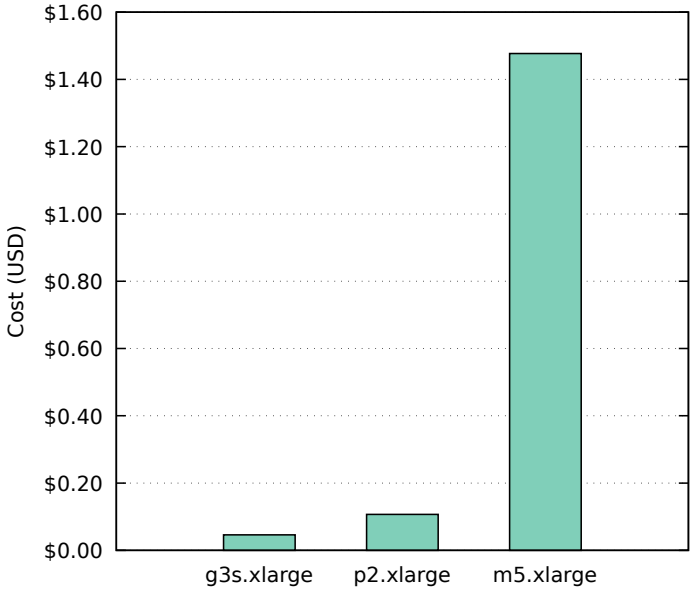
The performance over GPUs was measured using two different instance types: `p2.xlarge`, with 1 NVIDIA Tesla K80 GPU, 4 vCPUs, and 61 GB of RAM, which costs $0.9 USD per hour on demand; and `g3s.xlarge` with 1 NVIDIA Tesla M60 GPU, 4 vCPUs, and 30.5 GB of memory, with an on-demand cost of $0.75 USD per hour. To leverage GPU acceleration, Darknet was compiled with CUDA.

Figure 2.6 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.

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.



(**a**) Total execution time



(**b**) Total execution cost

Figure 2.6: Comparison between CPU and GPU instances for video object detection.

2.5.3 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 [2]. Furthermore, the boot and initialisation time of the instances must also be considered.

Figure 2.7 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.

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.

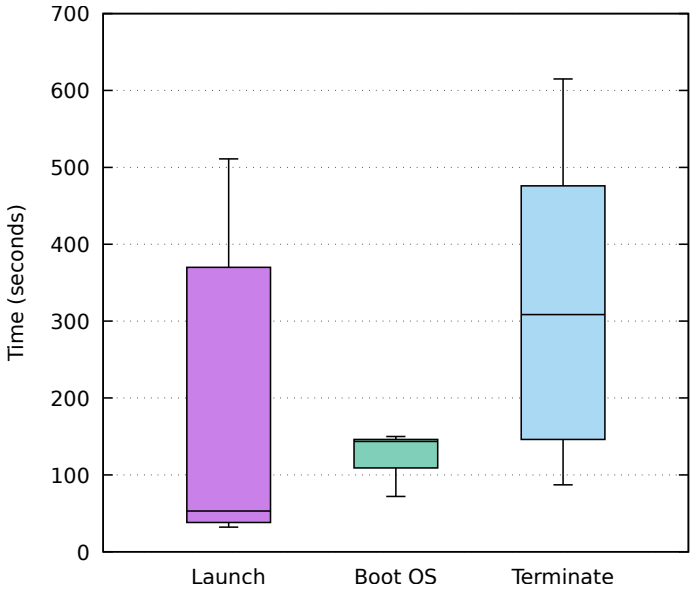


Figure 2.7: Launch, boot OS and terminate time of instances on AWS Batch.

2.5.4 Workflow Execution

With the aim of testing the platform, the use case defined in Section 2.4 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 2.5.1. 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 2.5.2. 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 2.8. 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.

Table 2.1 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 2.5.2, 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.

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

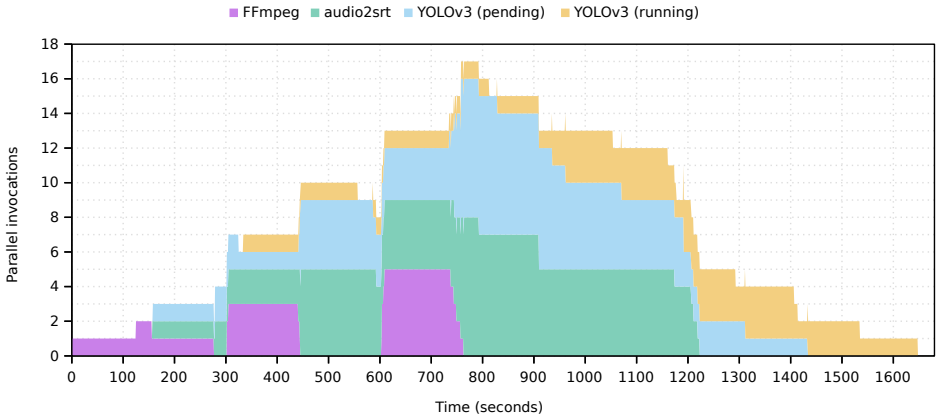


Figure 2.8: Time chart for the processing of ten videos using the workflow. Parallel invocations of the functions appear stacked. For the YOLOv3 function (running on AWS Batch) the pending and running states are distinguished.

Table 2.1: 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

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.

2.6 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* [47]. In addition, we plan to incorporate external data sources for long-term persistence outside AWS, such as the EGI DataHub [192].

Chapter 3

Serverless Workflows for Containerised Applications in the Cloud Continuum

Published as

Sebastián Risco, Germán Moltó, Diana M. Naranjo and Ignacio Blanquer. (2021) Serverless Workflows for Containerised Applications in the Cloud Continuum. Journal of Grid Computing, 3 (19). https://doi.org/10.1007/s10723-021-09570-2

Abstract

This paper introduces an open-source platform to support serverless computing for scientific data-processing workflow-based applications across the Cloud continuum (i.e. simultaneously involving both on-premises and public Cloud platforms to process data captured at the edge). This is achieved via dynamic resource provisioning for FaaS platforms compatible with scale-to-zero approaches that minimise resource usage and cost for dynamic workloads with different elasticity requirements. The platform combines the usage of dynamically deployed auto-scaled Kubernetes clusters on on-premises Clouds and automated Cloud bursting into AWS Lambda to achieve higher levels of elasticity. A use case in public health for smart cities is used to assess the platform, in charge of detecting people not wearing face masks from captured videos. Faces are blurred for enhanced anonymity in the on-premises Cloud and detection via Deep Learning models is

performed in AWS Lambda for this data-driven containerised workflow. The results indicate that hybrid workflows across the Cloud continuum can efficiently perform local data processing for enhanced regulations compliance and perform Cloud bursting for increased levels of elasticity.

3.1 Introduction

Cloud computing has become in the last decade the premier option for virtualised computing. It has increased hardware resource utilization and provided the ability to execute disparate computing workloads with complex requirements on shared computing infrastructures. Initial service delivery models, such as Infrastructure as a Service (IaaS), were exemplified by public Cloud services such as Amazon EC2 [12] and on-premises Cloud Management Platforms (CMPs) such as OpenStack [146]. These were later extended to accommodate additional models such as Platform as a Service (PaaS) and, more recently, Functions as a Service (FaaS). FaaS aims to rise the level of abstraction for application developers at the expense of relying on the infrastructure provider for automated elasticity, efficient virtual infrastructure provisioning and improved resource allocation.

Initial FaaS services, exemplified by public Cloud services such as AWS Lambda [15] and Azure Functions [124], provide event-driven execution of functions coded in certain supported programming languages, offering automated resource allocation and ultra-elastic capabilities that superseded the ones found in traditional IaaS offerings. For example, a Lambda function can support up to 3000 concurrent executions which is two orders of magnitude beyond the default number of virtual machines that can be deployed in a newly created AWS account, which is 20 (and can of course be increased upon request). This could only be achieved by using lightweight virtualization technologies, as is the case of Firecracker [2] which allows deploying micro Virtual Machines (microVMs) in less than a second. In general, lightweight virtualization such as Linux containers (LXC) [119], introduced in 2008, also paved the way for this success. Indeed, the increased popularity gained by Linux software containers fostered the emergence of Docker [69] in 2013, which spawned an entire ecosystem of tools that boosted innovation and widespread adoption.

Both enterprise-based workloads and scientific computing benefited from this trend to provide encapsulated applications with all the dependencies to guarantee successful execution across a myriad of computing platforms. Docker images turned into a *de facto* approach for consistent multi-platform application delivery. The advent of containers, together with the sustained development of Container

Orchestration Platforms (COPs) such as Kubernetes [39], paved the way for implementing the event-driven capabilities of FaaS under open-source platforms such as OpenFaaS [144], Knative [112] and Apache OpenWhisk [25]. These platforms mimic the functionality of public FaaS offerings for the execution of functions coded in certain languages within the premises of an organization. This computing paradigm for the Cloud, in which dynamic resource allocation is managed by the Cloud infrastructure provider, was coined as serverless computing [31].

However, the benefits of serverless computing cannot be restricted to just function-based computing, especially in the case of scientific computing [99] where complex software dependencies [102], resource-intensive requirements [191, 150] and, sometimes, the necessity of accelerated hardware is required [59], features that are not currently available in public Cloud serverless offerings. Currently, an AWS Lambda function cannot run beyond 15 minutes, use more than 10240 MBytes of RAM, use any accelerated computing device such as GPUs or use an ephemeral storage space greater than 512 MB, thus jeopardizing the adoption of these platforms by scientific computing.

Also, new challenges arise for scientific applications to harness the computing continuum, as indicated in the work by Beckman et al. [35], where they identify multiple infrastructures on which computing takes place such as interconnected sensors from IoT/Edge devices to computer clusters and Cloud infrastructures. Workflow-like applications may benefit from the orchestration of resources along the computing continuum. These applications may gather data at the edge, perform local processing to comply with privacy regulations on on-premises computing platforms and seamlessly profit from the elasticity of public Cloud infrastructures to reduce overall makespan.

Towards this vision, this paper introduces an architecture composed of open-source components that supports the execution of workflow-based data-processing applications packaged as Docker containers that can elastically provision resources from on-premises Clouds and perform automated bursting into a public Cloud using an event-driven serverless approach. The flexibility of this architecture provides a step forward in defining data-driven workflows that can execute along the Cloud continuum.

After the introduction, the reminder of this paper is structured as follows. First, section 3.2 introduces the related work in the area of serverless scientific computing. Next, section 3.3 describes the components of the designed platform and the Functions Definition Language created to support data-driven workflows along the Cloud continuum. Later, section 3.4 describes a use case to assess the benefits of the developed platform that integrates Deep Learning models with serverless

computing to produce cost-effective processing of videos from surveillance cameras to detect mask usage by the population. Finally, section 3.5 summarizes the main achievements of the paper and points to future work.

3.2 Related Work

Several research groups understood from the early beginning that serverless computing could certainly benefit scientific computing. This is the case of the work by Jonas et al. [108] who introduced the PyWren tool to perform distributed computing using AWS Lambda, in order to support several programming models, building on the assumption that stateless functions can be a natural fit for data processing. Our earlier work in the area, MARLA (MapReduce on AWS Lambda)[1] by Giménez-Alventosa et al. [82] created a framework to execute Python-based MapReduce applications on AWS Lambda, thus producing a high-performant serverless open-source tool to execute High Throughput Computing (HTC) jobs without requiring any pre-provisioned computing infrastructure by the user. In this work we identified the unbalanced performance properties of serverless platforms such as AWS Lambda and produced a thorough research which identified performance variabilities in both network throughput and CPU performance for different invocations of the same Lambda function even with the same allocated resources. This sparked the need to create appropriate serverless load balancing strategies for HTC jobs that can minimise both execution time through proper dynamic load assignment thus resulting in reduced cost using the fine-grained billing models, as described in [83]. Moreover, the work by Fouladi et al. [77] also envisioned the mapping of thousands of parallel threads to multiple invocations of a Lambda function in order to achieve close to near-interactive completion times. They produced the *gg* software tool[2] which performs distributed compilation of large code bases, together with other use cases such as video encoding, offering an API with bindings for Python and C++.

The report by Sewak et al. [175] summarises the different applications of serverless computing along with the advantages and disadvantages of the main FaaS platforms in public Clouds, anticipating their growth and adoption in the near future, as well as indicating the need for new tools to harness the capabilities of these platforms and facilitating their adoption by developers. The applicability of serverless architectures to serve AI models has also been investigated in numerous studies. For example, the study carried out by Ishakian et al. [101] analyses the application of AWS Lambda to serve lightweight deep learning models, as

[1] MARLA - `https://github.com/grycap/marla`
[2] gg - `https://github.com/StanfordSNR/gg`

the maximum available ephemeral storage in a function is 512 MB, concluding that such platforms can be suitable for workloads running on warm functions. However, their results show how cold starts can add significant overhead in latency times when compared to conventional services deployed on virtual machines. Aditionally, the papers conducted by Christidis et al. [53, 54] propose a set of optimisations for deploying machine learning workloads on serverless platforms. Some of these optimisations are in fact aligned with those implemented in our work, such as minimizing container images or loading them on the ephemeral storage of functions in order to overcome the maximum size of the deployment package. These studies further conclude that it is worth adapting such applications to serverless platforms in view of the potential savings and robust elasticity, and point to the growing need to support specialised AI-accelerated hardware on such platforms.

Indeed, supporting serverless computing for scientific computing requires solving specific challenges that lie ahead the development of our early prototype. To begin with the first challenge, a problem that remains unsolved is chaining function composition to produce serverless workflows that can fully exploit resources from on-premises to public Clouds including computing at the edge for local data preprocessing. The serverless trilemma by Baldini et al. [32] identified that engineering function composition for a serverless application is possible but function composition must obey a substitution principle with respect to synchronous invocation and invocations should not be double-billed, what poses additional constraints to enact serverless workflows with respect to traditional workflow systems.

An early work by Malawski [122] explored the idea of serverless workflows for processing background tasks of Web applications and how to rethink serverless architectures for executing scientific workflows, introducing a prototype based on Google Cloud Functions coupled with the Hyperflow workflow engine. In this line, the work by Skluzacek et al. introduced Xtract, a service to process large collections of scientific files to extract metadata from various file types. They used funcX [48] to develop the prototype, a federated FaaS system to enable function execution across heterogeneous distributed resources. These functions are snippets of Python code and the system relies on Globus transfer to perform data staging. The authors found that it can be difficult to modify applications for stateful execution, since the state is not easily shared among functions. Thus, poorly designed solutions may lead to significant communication overhead.

Despite the large number of open-source FaaS frameworks, few research has been dedicated to serverless workflows, specially to those that are inherently data-driven because they require processing data across multiple stages of the work-

flow. For example, Faas-flow [172] provides function composition for the OpenFaaS framework by creating chains of functions that can be executed both synchronously and asynchronously with support for parallel execution with branching, even upon certain conditions. Other workflow engines that run on top of Kubernetes may be used to provide some support for serverless workflows. This is the case of Argo Workflows [26], an open-source container-native workflow engine for orchestrating parallel jobs on Kubernetes which models multi-step workflows as sequences of tasks via DAGs (Directed Acyclic Graphs) and which provides support for event-driven workflow automation.

In fact, serverless workflows is an active research area where several contributions are being proposed. For example, the work by Ristov et al. [169] introduces a language to describe function choreographies to connect serverless functions. Indeed, the Serverless Workflow Specification (SWS) [57] was recently approved as a Cloud native Sandbox level project to define declarative workflow models that orchestrate event-driven serverless applications. We expect that this specification will bring benefits in the area of consistency, providing a common way of describing serverless workflows, portability and accessibility, to provide interoperability among serverless workflow runtimes. However, for the time being, this specification allows to compose a workflow from pre-existing serverless functions and, therefore, does not involve function provisioning. Techniques such as *Dynamic parallelism* supported by AWS Step Functions are beneficial for the orchestration of microservices-based applications. Nevertheless, this technique is mainly employed for control-driven workflows, where the connections between the activities or tasks in a workflow represent a transfer of control from the proceedings task (or tasks) to the one (or ones) that follow [178]. However, the focus of our work is on data-driven workflows where a task input depends on the output data generated by the previous task.

Concerning the support to the computing continuum, several authors have previously explored this topic. For example, the work by Balouek-Thomert et al. [33] presents a vision to enable such a computing continuum and they set the focus on enabling edge-to-cloud integration to support data-driven workflows. They focus on stream-oriented workflows to filter data near the sources but they do not use a serverless approach and no open-source implementation is provided. The work by Baresi et al. [34] introduces the A3-E unified model for the Mobile-Edge-Cloud continuum which exploits the FaaS model to bring computation to the continuum. It uses Apache OpenWhisk to support the implementation together with AWS Lambda. However, no support for workflows is introduced.

It is precisely at the verge of this state-of-the-art that lies this contribution, producing an open-source platform that provides serverless scientific computing

along the Cloud continuum, including both on-premises and public Clouds, and that supports data-driven workflow enactment in serverless platforms and multi-Cloud hybrid deployments of infrastructures. To best of the author's knowledge, this is the first platform that supports event-driven serverless scientific computing simultaneously harnessing resources from multiple Clouds (exemplified in our case via OpenStack and AWS Lambda). The platform, together with the definition of the use case described in this paper has been released as open-source, publicly available in GitHub, for the sake of reproducibility.

3.3 Components to Support Serverless Workflows along the Cloud continuum

This section identifies the main components employed to support hybrid serverless workflows that can span across on-premises Clouds and public Cloud platforms to process data that may be captured at the edge. First, the SCAR[3] [153] software for serverless scientific computing in public Clouds is described. Later, the open-source OSCAR[4] [152] framework to support serverless computing for data-processing applications in on-premises Clouds is covered. Finally, this section introduces the Functions Definition Language (FDL) created to define the functions together with its relationship with data-driven serverless computing workflows.

The main contribution of this paper lies in the development of a new version of the OSCAR framework to match the same computing model provided by SCAR. This allowed the integration of both components to support the same computing model across both on-premises and public FaaS platforms for data-processing applications. Another key contribution is the development of a novel FDL to define data-driven serverless workflows that can execute along the Cloud continuum, in order to support the definition of use cases that require processing at different levels of this continuum. Notice that, by building on existing open-source software that is being used in production we aim to foster long-term sustainability of the developed architecture.

[3]SCAR - `https://github.com/grycap/scar`
[4]OSCAR - `https://github.com/grycap/oscar`

3.3.1 SCAR: Serverless Scientific Computing in Public Clouds

SCAR is an open-source framework that supports a High Throughput Computing model [151] to create embarrassingly parallel event-driven file-processing serverless applications on public FaaS platforms, currently supporting AWS Lambda. The applications can be packaged as Docker images that can be optionally stored in Docker Hub [68] (alternative means include Amazon S3). This allows to execute complex scientific applications in AWS Lambda, thus being able to spawn up to 3000 parallel invocations (depending on the region used). There are strict computing requirements per invocation in AWS Lambda, which are currently 10240 MB of RAM, 512 MB of ephemeral storage that is potentially shared across invocations and 15 minutes of execution time. Therefore, this typically requires using container minimization strategies in order to fit the Docker container within AWS Lambda's runtime environment, such as those available in tools like *minicon* [94], which analyses an application execution to obtain a filesystem that exclusively contains the dependencies detected.

For those applications that do not fit within AWS Lambda's computing requirements, SCAR provides a seamless integration with AWS Batch [14] an elastic-cluster as a service offering by AWS which dynamically deploys a cluster in charge of executing jobs packaged as a Docker images and which can grow and shrink depending on the number of jobs queued up at the Local Resource Management System (LRMS). This integration allows to delegate into AWS Batch functions invocations that require longer execution times, larger amount of memory or even GPU resources for accelerated execution, as described in the work by Risco et al. [167].

However, there are applications that can benefit from the event-driven behaviour of serverless platforms but that require strict privacy requirements and, therefore, cannot be run in a public Cloud provider. Also, there are organizations that are already operating an on-premises Cloud managed by a Cloud Management Platform such as OpenStack [146] and, therefore, do not want to spend additional economic cost from provisioning resources from a public Cloud. The OSCAR platform described in the following section was developed to support these scenarios.

3.3.2 OSCAR: Open-Source Serverless Computing for Data-Processing Applications

OSCAR is an open-source platform to support the Functions as a Service computing model for compute-intensive applications. OSCAR can be automatically deployed on multi-Clouds in order to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers than run on an elastic Kubernetes cluster that grows and shrinks depending on the usage of resources.

The automated deployment of an OSCAR cluster on multi-Clouds is achieved using:

- IM (Infrastructure Manager) [41], a TOSCA-compliant [142] Infrastructure as Code (IaC) tool to deploy complex customized virtualised infrastructures on the major on-premises and public Infrastructure as a Service providers.
- CLUES (CLUster Elasticity System) [5], an elasticity manager that allows virtual clusters to grow and shrink in terms of the number of nodes. It has plugins for popular systems such as Kubernetes, Apache Mesos, SLURM, etc.
- EC3 (Elastic Cloud Computing Cluster) [43], which combines the two developments above to deploy automated self-scaling clusters on multi-Clouds.

An OSCAR cluster features the integration of the following components:

- Kubernetes [39], a container orchestration platform, thus managing containerised applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications.
- OpenFaaS [144], an open-source FaaS framework to execute short-lived functions on top of a container orchestration platform.
- MinIO [128], an open-source object storage system with Amazon S3's API compatibility.
- OSCAR, the component in charge of creating a function together with the required resources to support event-driven batch-based GPU-aware executions on top of the Kubernetes cluster for serverless scientific computing.

The creation of an OSCAR function allows users to upload files to the object storage system which triggers the execution of the function to perform the data-processing, with automated elasticity if it is required, and the output data is

stored in any of the object storage systems supported. This is the case of Onedata [71] a global data management system that provides access to distributed storage resources for data-intensive scientific computations. This is used to support EGI DataHub, a federated data storage layer auspiced by EGI (European Grid Infrastructure), the largest federated Cloud in Europe. OSCAR is compatible with the EGI DataHub. Other object storage systems, such as Amazon S3, can be employed to store the output data, thus allowing to trigger the AWS Lambda functions.

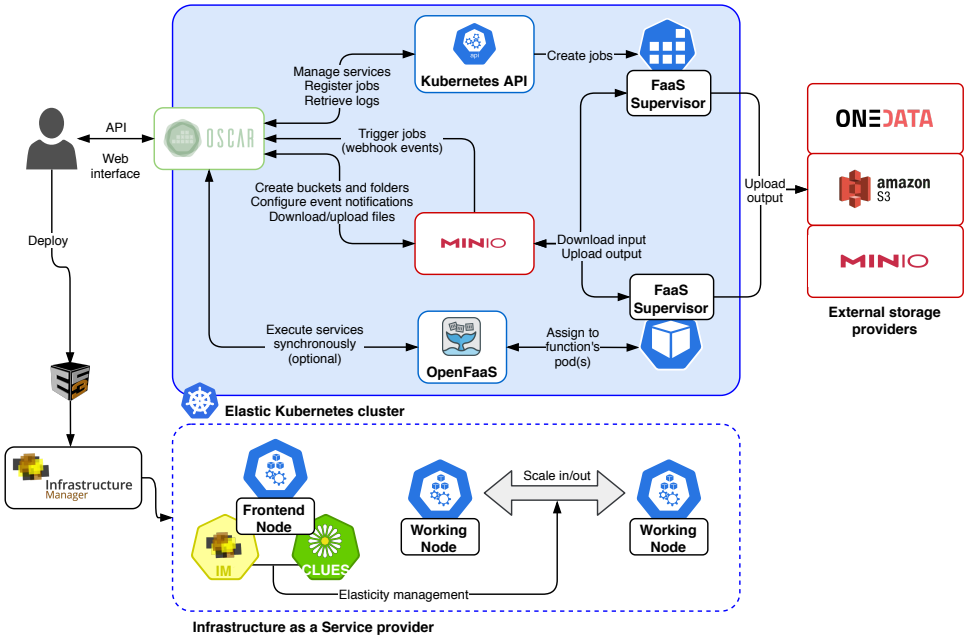


Figure 3.1: Architecture of the OSCAR platform and interactions among their services.

Fig. 3.1 describes the internal architecture of an OSCAR cluster. The bottom part depicts a horizontally elastic Kubernetes cluster that is deployed via EC3 from pre-defined TOSCA templates that are employed by the Infrastructure Manager (IM) to provision and configure the front-end node of the cluster. This node is configured with the required Kubernetes services, the CLUES elasticity manager, and a private instance of the IM server deployed in the aforementioned front-end node. This way, the clusters become autonomous in deciding whether to scale out (provision additional nodes from the underlying Cloud) or to scale in, depending on the number of pods that are pending to be executed.

The upper part of the figure shows the main components of the cluster together with a typical workflow. For this contribution, OSCAR was completely redesigned in order to support the computing model offered by SCAR. To this aim, OSCAR exposes a secure REST API that receives requests to create functions. It is responsible for creating the corresponding input and output buckets in MinIO, depending on the function configuration, and configure the event notifications in order to trigger the function execution upon a file upload to the input bucket. OpenFaaS is employed in order to perform synchronous executions of function invocations, typically short-lived, which is the most common use case of serverless computing. However, in order to support resource-intensive event-driven scientific computing, asynchronous executions are required. To this aim, OSCAR creates a Kubernetes job for each asynchronous invocation that are delegated into the Kubernetes workload scheduler for efficient execution. These jobs are wrapped with the FaaS supervisor[5], an Input/Output data manager especially created for multi-cloud settings, which allows to gather data from input data storages and upload output data into the corresponding data storages. The supported data storages are depicted in the right part of the picture.

Security has been addressed using best practices depending on the infrastructure being employed. For example, Lambda functions use pre-defined IAM (Identity and Access Management) Roles[6] that follow the Principle of Least Privilege (PoLP) so that they can only access the resources required, such as an Amazon S3 bucket to store the generated output data. The deployment via EC3 of the Kubernetes cluster dynamically generates a token for the user to connect to the OSCAR web-based user interface and tokens to access the Kubernetes dashboard and MinIO browser, in case the user wants to directly access them. Dynamic generation of secrets prevents from reusing passwords that would cause severe security implications such as unauthorized access breach that could be exploited for nefarious purposes. Moreover, the OSCAR API requires basic auth and is exposed through a Kubernetes ingress that supports SSL. Finally, Onedata leases tokens, which can be revoked at any time, in order to provide access to the space.

[5]FaaS Supervisor - `https://github.com/grycap/faas-supervisor`
[6]IAM Roles: `https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html`

3.3.3 Functions Definition Language for Data-Driven Serverless Workflows

To support the deployment of data-driven workflows of serverless functions that require complex data-processing, we opted for defining a YAML-based Functions Definition Language (FDL) that specifies the requirements for each function and how they are linked. Notice that, unlike the Serverless Workflow Specification (SWS) which provides a workflow definition out of existing FaaS functions, our language focuses on the definition of the functions to be dynamically created across the hybrid Cloud. Therefore, our proposal could be coupled at a later stage with the SWS language to provide a portable, interoperable description of workflows out of the dynamically created functions from our platform.

Two top-level resources are defined in a FDL (see a sample document in Fig. 3.2, used to support the case study described in section 3.4):

- Functions, which are created in a Cloud provider and they are assigned a name, a certain amount of computing resources together with a shell-script that will be executed, as part of the function invocation, inside a container created out of a specific Docker image that may available in Docker Hub. The function will be triggered whenever a file is uploaded to a specific folder within a storage provider and the shell-script will be in charge to perform the data processing on the file.
- Storage Providers, which become sources of events for input data processing and store the output data results from a function invocation. By using as output from a function the input storage provider from another function, a precedence relationship is established among them and a data-driven link is created.

Notice that our platform focuses on data-processing applications and, therefore, each function is linked to an input storage provider (so that the function is invoked upon a file upload) and also to one or more output storage providers (where the output file results of the processing will be stored). The choice of Docker allows to have complex execution environments that may be required by scientific applications, which typically rely on multiple libraries, require specific OS distributions, etc. This way we can have a consistent execution environment whether using a public Cloud or an on-premises one. Note that SCAR's ability to handle Docker images as runtime environments within both AWS Lambda and AWS Batch also supports the decision to use Docker images in this environment.

The choice of shell-scripts instead of providing bindings for specific programming languages responds to the goal of supporting scientific applications, which are

```
---
functions:
  aws:
  - lambda:
      name: scar-mask-detector
      memory: 1024
      init_script: mask-detector.sh
      container:
        image: grycap/mask-detector-yolo:mini
      input:
      - storage_provider: s3
        path: scar-mask-detector/intermediate
      output:
      - storage_provider: s3
        path: scar-mask-detector/result
  oscar:
  - my_oscar:
      name: oscar-anon-and-split
      memory: 2Gi
      cpu: '1.0'
      image: grycap/blurry-faces
      script: blurry-faces.sh
      input:
      - storage_provider: minio
        path: oscar-anon-and-split/input
      output:
      - storage_provider: s3.my_s3
        path: scar-mask-detector/intermediate
storage_providers:
  s3:
    my_s3:
      access_key: xxxxxx
      secret_key: xxxxxx
      region: us-east-1
```

Figure 3.2: Functions Definition Language file to deploy the workflow used in the following section.

typically legacy applications that are unfeasible to be adapted to other programming languages other than those initially used to code them. Also, this allows to execute *any* application that supports the command-line, thus broadening the scope of applications that can be supported, instead of forcing developers to adapt their legacy applications to a certain programming language.

The integration of SCAR and OSCAR tools, together with the adaptation of the FaaS Supervisor, has been carried out in order to support these new FDL files. For this purpose, as previously indicated, OSCAR has been redesigned to support the function definition model through its REST API, along with other improvements such as a refreshed web interface and the ability to retrieve execution logs for enhanced visibility. The SCAR tool has also been improved to allow communication with OSCAR endpoints, as well as a parser update to handle the new

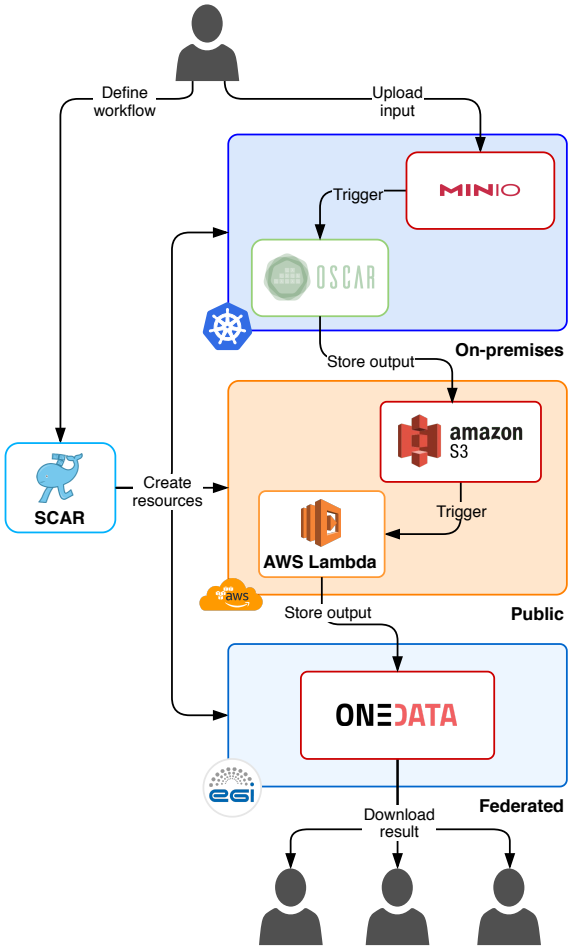


Figure 3.3: Simplified diagram of a hybrid serverless workflow that involves public, on-premises and federated cloud resources.

functions definition files. Finally, the FaaS Supervisor required minor changes to support the loading of storage providers' credentials from the new files. In this way, as shown in Fig. 3.3, SCAR has become a tool capable of orchestrating resources to support file-processing serverless computing along the Cloud continuum. Hence, SCAR manages the creation of the required resources both in AWS and in the OSCAR cluster together with the corresponding output folders in the Onedata space. The composition of the different steps of the workflow is achieved

by using the output bucket of one function as the input of the subsequent one. Therefore, to start the execution of the workflow, users only have to upload a file to the input bucket of the first function.

Note that infrastructure management is being left out from the FDL, in line with the serverless approach of delegating in the Cloud provider for this task. In our case, it is at deployment time of the OSCAR cluster when the user/administrator indicates the maximum number of nodes of the cluster, together with their computing requirements. This way, the user can focus on the definition of the application workflow and let the cluster auto-scale within the on-premises Cloud. In the case of functions created with SCAR, the concurrency limits can be specified by the user at creation time.

In order to demonstrate the benefits of the designed platform, the following section introduces a use case related to public health in smart cities.

3.4 Use Case: Mask Wearing Detection via Anonymised Deep Learning Video Processing

Data analysis in the context of smart cities is an active area of research and the role of data processing (Big Data) to extract knowledge from dense networks of sensors across a city is summarised in the review work by Nuaimi et al. [4], in the work of Camero et al. [44] and in the work by Chen et al. [51]. As an example, the work by Bello et al. [36] highlighted the importance of sound as a source of information about urban life, focusing on monitoring noise pollution and audio surveillance. Indeed, the work by Spadini et al. [183] focused on ambient sound processing across smart cities in order to detect abnormal events such as gunshots, sirens, etc.

Using a similar approach, we focus in this study on smart camera networks [165], which are distributed systems that perform computer vision tasks using multiple cameras. These have implications in activities such as surveillance with cameras that capture the natural movement of individuals and vehicles in everyday environments, as indicated in the work by Chen et al. [50].

This use case targets an scenario of video surveillance in which it is required to provide increased monitoring capabilities for the authorities to take better public health decisions. With the COVID-19 global pandemic that affected the entire world starting late 2019, many national authorities have regulated the mandatory use of face masks in order to minimize the spread of the virus across the population. To this aim, this use case introduces a workflow entirely based on

open-source components that allows to determine the people that are not wearing a mask out of sampled images from video recordings that could be obtained from a network of cameras distributed throughout a city. This may allow public health authorities to better devote resources to minimize this trend in the specific areas being monitored.

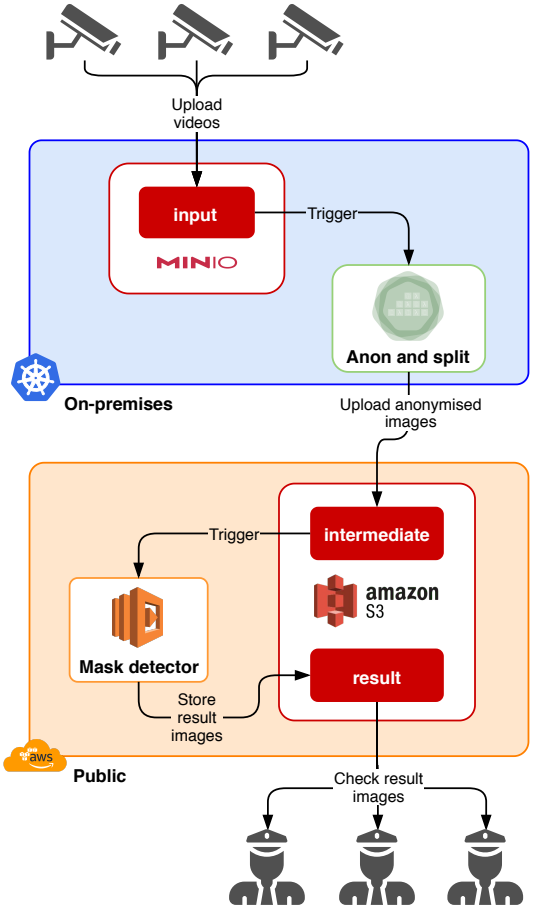


Figure 3.4: Workflow for the defined use case involving face mask detection on anonymised images on a hybrid Cloud.

However, according to the NIST Guide to Protecting the Confidentiality of Personally Identifiable Information (PII) [123], a person's face is considered a PII because it can unequivocally identify a human being. Therefore, in order to protect the privacy of the individuals, a pre-processing stage is performed in order

to blur the faces before applying a deep learning model to perform the face mask recognition. This is why a hybrid serverless workflow is required so that processing is performed along the Cloud continuum, where data is captured at the edge (camera devices), pre-processing is carried out in an on-premises Cloud for regulatory compliance purposes and, finally, processing and storing of final results is carried out in a public Cloud using a serverless platform for increased elasticity and long-term persistence.

The steps of the workflow can be shown in Fig. 3.4. A set of cameras from a smart camera network periodically take short videos that are automatically uploaded to an on-premises Cloud with a MinIO installation. Each uploaded video triggers an event that starts the "Anon and split" function within the OSCAR cluster in order to extract a frame every 5 seconds of video and perform the initial anonymisation phase on the extracted images to blur the faces using the BlurryFaces tool [129]. This phase takes an average of 65 seconds to chunk and anonymise 1 minute of video at a resolution of 1920x1080. Therefore, considering the computational requirements and the need to comply with the local regulations related to the use of PII, this phase can be performed in the on-premises Cloud.

The resulting anonymised images are then uploaded to an Amazon S3 bucket in order to start the inference process in the public Cloud. Each uploaded image triggers the "Mask detector" Lambda function responsible for using the *face-mask-detector* [156] Deep Learning model in order to compute the percentage of people in the picture that are not wearing a face mask. The output images are made available in another Amazon S3 bucket to guarantee long-term data persistence and for the responsible stakeholder to take actions upon the results obtained. We rely on Amazon S3 instead of EGI DataHub to store the output data for the sake of easier reproducibility. A sample image result of the processed workflow is shown in Fig. 3.5.

Notice that this technology can be applied by the local authorities to perform a quantitative and systematic evaluation of the fulfillment of the regulations to determine if further enforcement is required. The ability to safely outsource the embarrassingly parallel part of the workflow to a public Cloud supports the scalability of the designed approach.

The following subsections describe the approach employed to create the workflow, together with optimization techniques applied for increased cost-effectiveness.

Figure 3.5: Result image that differentiates people not wearing face masks.

3.4.1 Optimal Resource Allocation for the Lambda Function

The allocation of computing resources for an AWS Lambda function is linearly dependent on the amount of allocated RAM. Hence, increased amount of RAM may reduce the execution time but, at the same time, increases the cost, which is billed in milliseconds of execution time. Therefore, in order to choose the optimal amount of memory to achieve cost-efficient executions in a timely manner, we relied on AWS Lambda Power Tuning [47], an open-source tool to optimize Lambda functions for cost/performance using a data-driven approach. The tool performs the execution of the function with different memory allocations in order to compute both the execution time and the total cost.

Fig. 3.6 shows the output results obtained by the aforementioned tool showing average values ($N = 5$ repetitions) for the execution cost and the execution time for the mask detector function running with different RAM values starting at 256 MB, the least amount of RAM required to execute the deep learning model, until 4096 MB. Larger memory amounts have been omitted since the execution time remained at similar values, as can be seen in the line between 2048 MB and 4096 MB, while the cost kept increasing. Note that the first invocation for each memory amount is performed when the container image is not available in the AWS Lambda environment, which results in an increased execution time due to

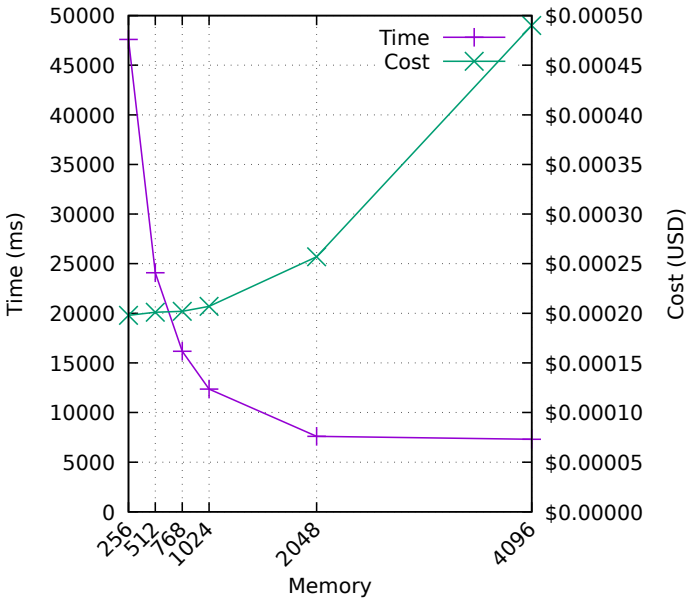


Figure 3.6: Time and cost analysis for the mask detector function running on AWS Lambda.

the cold start, while in the remaining ones (i.e. when the Docker image is already available for the function invocations) no variation is appreciated. The function was created using SCAR in the *us-east-1* region and outside of a VPC (Virtual Private Cloud), which provides faster execution times.

The figure shows that the best cost is obtained with the least amount of memory, at the expense of achieving the worse (maximum) execution time. Notice that the execution time reduces almost linearly when using up to 512 MB of RAM. From this point on, increasing the RAM, which proportionally affects the CPU allocation, provides moderate improvements in the execution time with an increased cost, which starts to grow considerably from 1024 MB upwards. Finally, allocating 4096 MB provides a marginal improvement with respect to 2048 MB at the expense of a substantial cost increment. For this particular application, the optimal amount of memory lies between 768 and 1024 MB of RAM, depending on the budget restrictions of the user.

Several key results are obtained from this analysis. On the one hand, allocating additional memory to a Lambda function tends to reduce the execution time, but this is not always the case (compare the execution time with 2048 and 4096 MB of RAM in the figure above). On the other hand, the execution time is

billed in milliseconds and, therefore, optimization is a mandatory strategy when creating a Lambda function. Marginal optimizations on a Lambda function that are executed a large amount of times end up in producing significant cost savings.

3.4.2 Case Study Design

Two scenarios were designed in order to prove the benefits of the designed platform. In the first one, the whole workflow is executed on the on-premises platform. In the second one, the proposed hybrid workflow has been used, which involves resources both from the on-premises Cloud and the public Cloud. This allows to better assess the benefits of adopting a hybrid approach along the Cloud continuum.

The on-premises platform employed to conduct the experiment consists of an elastic Kubernetes cluster deployed with the EC3 tool, since the use of an Infrastructure as Code (IAC) [132] approach allows to guarantee deterministic provisioning of customized virtual infrastructures. The cluster is composed of a front-end node and a maximum of 5 working nodes, each one with 4 vCPUs and 8 GB of memory. The underlying infrastructure is a physical cluster supported by OpenStack which includes 14 Intel Skylake Gold 6130 processors, with 14 cores each, 5.25 TB of RAM and 2 x 10GbE ports and 1 Infiniband port in each node.

For the "Anon and split" function, which always runs on the OSCAR platform, 1 vCPU and 2 GB of memory were set. Note that although the working node instances have 4 vCPU, the internal components of Kubernetes require a small amount of resources (0.2 vCPU and 250 MB approximately), which makes the 4 entire vCPUs unavailable for processing. Therefore, the Kubernetes scheduler can only assign for execution three function jobs simultaneously on the same working node. Moreover, the "Mask detector" function was executed on the public or on-premises Cloud depending on the scenario. Considering the results of the study carried out in the section 3.4.1, 1024 MB of RAM were chosen for this function in both scenarios, setting 1 vCPU when running in the on-premises OSCAR cluster.

The SCAR client was used to perform the workflow deployment, depicted in Fig. 3.4 across the hybrid infrastructure using the FDL file shown in Fig. 3.2. This file shows the definition of two functions, one in AWS Lambda and one in the OSCAR cluster, together with their computing requirements. It also indicates the Docker image from which a container will be created to execute the script that will process the file that triggers the execution of the function. Decoupling the infrastructure provisioning with the workflow deployment allows to reuse the

underlying provisioned infrastructure to support multiple hybrid workflows from different users, thus supporting a multi-tenant approach.

To perform the different tests, a sample video with a resolution of 1920x1080 pixels and a duration of 186 seconds was used. After the "Anon and split" phase this resulted in 37 images to be processed by the "Mask detector" function. The different experiments conducted on each scenario together with the results obtained are presented below.

3.4.3 Results and Discussion

This section analyses the results obtained from the execution of the different scenarios previously defined.

Video Processing Analysis

In order to prove the effectiveness of hybrid serverless workflows for processing data produced at the edge, the times obtained after 5 workflow runs to process a single video were measured. As mentioned above, the first function is responsible for extracting frames from the input video every 5 seconds and then applying an anonymisation strategy based on distorting the faces. This "Anon and split" function will always run on the on-premises platform (OSCAR), which will ideally be deployed on private Cloud infrastructures that comply with established data protection regulations, or even on intermediate devices located near the edge (fog computing). Afterwards, the "Mask detector" function will be triggered by each image resulting from the previous function, hence it will be possible to evaluate its performance when processing several images in parallel on both scenarios.

Fig. 3.7 shows the times obtained for each function after processing the sample video 5 times in the two defined scenarios. The first function required an average time of 206.4 seconds on the first scenario and 227.4 seconds on the second one. This increment is caused by the uploading of the images to the input bucket of the second function, which in the first case was located in the same cluster (due to the use of MinIO), while in the hybrid workflow is on Amazon S3, so the files must be uploaded via the Internet.

In the second function, a significant improvement can be seen due to the massive parallelism supported by AWS Lambda, which allows all the images to be processed in parallel. The average time obtained in the first scenario was 111.4 seconds, since the maximum level of parallelism was 3 processing jobs within a single node. Notice that to process a single video, the CLUES elasticity man-

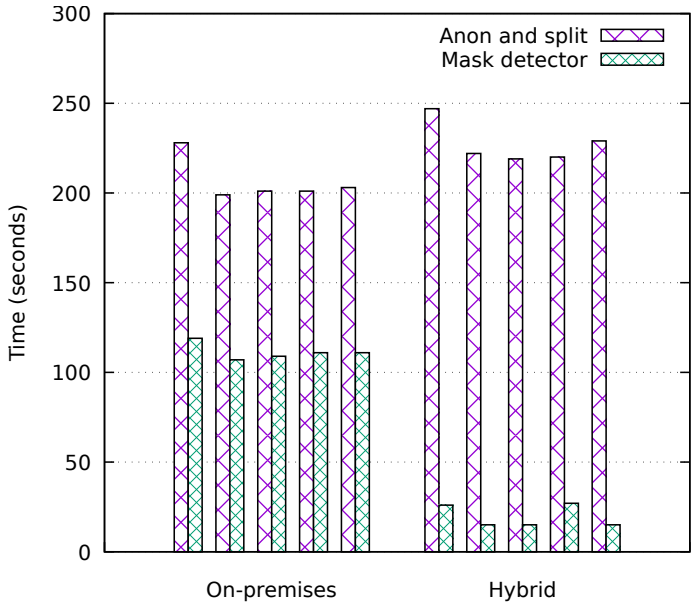


Figure 3.7: Execution times of the workflow functions in the two defined scenarios.

ager of OSCAR does not have enough time to scale out the cluster by deploying additional working nodes. In AWS Lambda, however, the processing of all the images completed in an average of 19.6 seconds, which supposes an improvement of 82.4%. As you may observe, the processing time of the 27 images in AWS Lambda is longer than that of a single image, i.e. not all are strictly performed in parallel. This is due to the fact that the time has been measured from the moment the first image starts to be processed until the result of the last image is saved. Delays in uploading files to S3 from the first function directly affect the total image processing time, which causes that not all the images are uploaded simultaneously. Therefore, good network connectivity between the two Clouds will considerably increase the performance of the second function.

Moreover, as observed in the figure, the first execution of each function has longer times than the rest. This can be explained by the fact that these values are measured when the function is triggered for the first time on the platform. On the one hand, in the OSCAR Kubernetes cluster, the first time a function is executed on a working node, the container image must be downloaded from Docker Hub, which generates a delay of up to 13% in the first function and 10% in the second one. Notice that the image used in the first function is considerably larger (1.51

GB) than that used in the second function (219 MB), as it has been reduced by *minicon* to fit on the constrained environment of AWS Lambda. On the other hand, the "Mask detector" function in the second scenario is performed on AWS Lambda and, as described in [153], if the function is "cold", the time taken to download the container image must be added to the time taken to start up the corresponding execution environment, increasing the gap from the average value.

Also, the execution cost of the "Mask detector" function in the second scenario has been analysed. As discussed in the section 3.4.1, the optimal performance point for the function in AWS Lambda was approximately 1024 MB of RAM. The average billed time after a cold start of the function is 12908 ms, which translates into an image processing cost of $0.00021556. On the other hand, the subsequent executions, having the image of the container available, will be much faster, obtaining an average billed time of just 2123 ms and a cost of $0.00003545. Summarizing, the processing cost of the 37 images generated by the video can range from $0.00131165, when the function is "warm", to $0.00797572 in the worst case, if all the executions were triggered exactly at the same time and none of them had the container image in the file system.

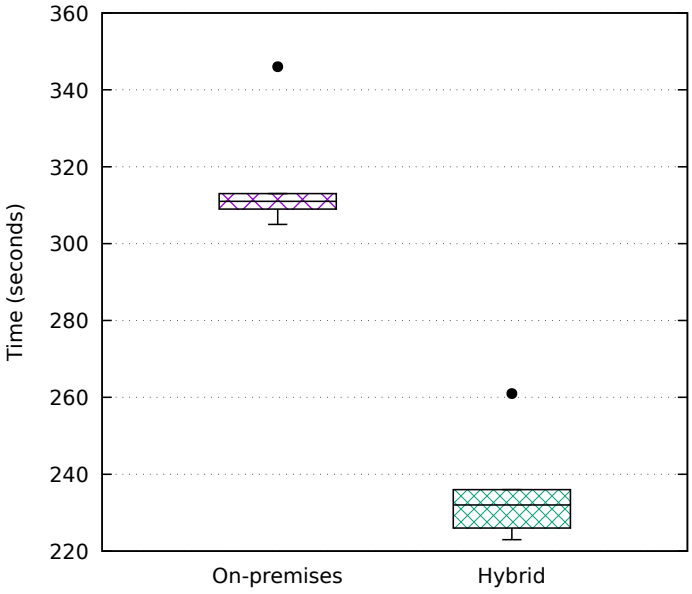


Figure 3.8: Total execution times of the workflow in the two defined scenarios.

The total times measured when performing a complete workflow execution are shown in Fig. 3.8. As mentioned above, the maximum values in each scenario, displayed in the box chart as outliers, match the first invocations of each function when the container images are not present in the working node and, in the case of AWS Lambda, the function is cold. Despite the fact that these situations considerably increase the time needed to complete the workflow, they can easily be avoided by downloading the images when each node in the cluster is started. For AWS Lambda several techniques exist to keep the functions "warm", such as invoking them periodically or activating the *Provisioned Concurrency* feature, at the expense of increasing the cost of the application.

In an edge computing scenario, on-premises clusters used by the sensors (the surveillance cameras in the experiment) will be of reduced scale so they cannot scale up to the sizes of AWS Lambda. Moreover, automatic scaling up the cluster through CLUES would require booting up resources and for this specific case will imply a prohibitive overhead. Although the figures of the experiment with a larger-scale on-premise cloud would have been more competitive with respect to the AWS Lambda, they would not be comparable in a large-scale production-level scenario.

Parallel Video Processing Analysis

In order to test the scalability of the designed system, we executed the workflow under the two scenarios with several number of videos to be processed. The videos were uploaded simultaneously to the input bucket thus simulating the data capturing process from several cameras at the edge. Fig. 3.9 shows the results obtained. Notice that the benefits of performing a hybrid approach appear since the beginning, as identified in the previous section, but the margin of improvement increases as the workload increases. Indeed, the impressive elasticity capabilities of AWS Lambda, that may perform up to 3000 parallel invocations greatly surpasses the bottlenecks that are typically found in on-premises Clouds where the parallel execution slots are limited to those available in the provisioned infrastructure.

At a more detailed level, the workflow execution for the processing of a single video on the first scenario has only been executed on one node. Despite the fact that CLUES triggers the scale-out order to the 4 remaining working nodes shortly after the creation of the 37 Kubernetes jobs for treating the images, all the jobs ended up on the active WN before the new ones completed their start up and configuration process. This means that resources are wasted when the load is low, as the platform is able to scale, but not in time. Thus, after a period of idle time the new nodes are shut down again. As can be seen, when the on-

premises platform receives more load (starting from 5 videos), the processing time is reduced in both scenarios as a result of the availability of more resources for parallel job processing.

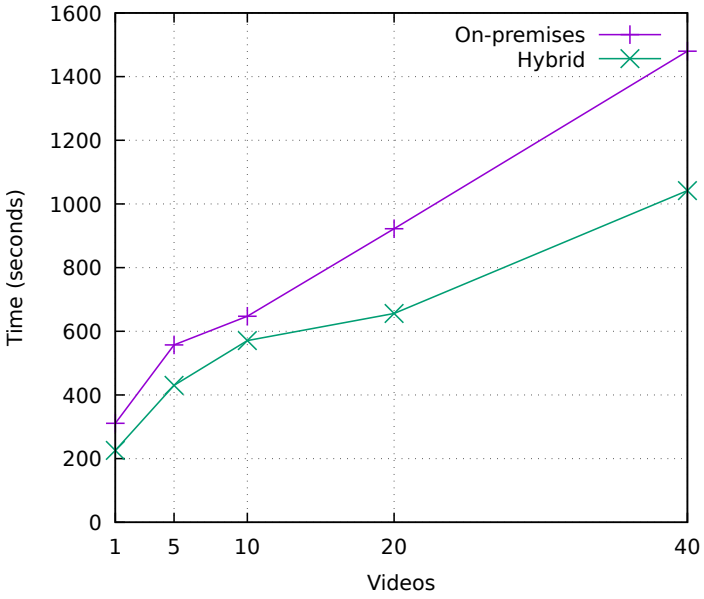


Figure 3.9: Measured times after processing different amounts of videos in parallel.

It is important to point out that the gap between the two lines in Fig. 3.9 can be closed by increasing the number of parallel execution slots and reducing the time until they are initially available for execution. The former is bounded by the underlying physical hardware available, which is fixed, and the allocation of resources to the deployed Virtual Machines, i.e., the instance types, which can be configured at deployment time. The latter depends on the elasticity rules employed in the cluster. In our case, the CLUES elasticity manager governing the rules to scale out (add additional nodes) and scale in (terminate the free nodes) was configured to only start the front-end of the cluster and a working node in charge of performing the job executions. This is a conservative strategy that aims to minimize energy consumption in an on-premises Cloud and only reactively provision additional resources whenever they are needed. Since CLUES rules can be configured, the user may prefer to have a pre-provisioned fleet of VMs that are immediately available upon moderate changes in the workload to be processed.

The use of an open-source stack that can be fully configured by the user in order to seamlessly perform both infrastructure provision along the Cloud continuum and data-driven workflow enactment using a serverless approach is an important step forward in widespreading the adoption of this techniques for scientific computing. Traditional serverless use cases focused on unpredictable bursts of short-lived requests, as is the case of web applications. However, we have demonstrated that compute-intensive, workflow-based applications can also benefit from the event-driven capabilities and automated resource management provided by serverless computing.

3.5 Conclusions

This paper has introduced an open-source platform that supports the definition of event-driven file-processing workflows that can execute across the Cloud computing continuum that features underlying elasticity in the provisioning of resources. The ability to Cloud burst into a public Cloud using a serverless approach introduces an unprecedented level of elasticity when compared to traditional approaches based exclusively on Virtual Machines.

The seamless integration between SCAR, which supports the execution of containers within AWS Lambda to bring serverless for scientific computing, and OSCAR, which provides the FaaS computing model for file-processing applications on Kubernetes clusters, has allowed to create hybrid data processing workflows across the Cloud continuum. These workflows can orchestrate automated provisioning of resources both in the on-premises Cloud, through elastic Kubernetes clusters and in the public Cloud, through the use of serverless services such as AWS Lambda.

A use case based on smart camera networks with applications in smart cities for video surveillance has been envisaged and assessed, in order to efficiently determine the usage of face masks across the population out of processed videos using Artificial Intelligence models. The use case has been made publicly available in GitHub[7] in order to guarantee its reproducibility. The experiments show that it is affordable and efficient to deviate a computing intensive part of the processing to AWS Lambda, rather than processing it on limited-scale, on-premises clusters, even if those clusters would have better network connectivity. This fact is more evident as the scale factor increases

Future works include dynamic resource orchestration across the Cloud-to-Things continuum, where the workflow can anticipate the expected incoming workload

[7]Mask detector workflow - `https://github.com/grycap/scar/tree/master/examples/mask-detector-workflow`

in order to further adapt the resources. This would minimize the amount of time invested in provisioning additional nodes within the on-premises Cloud and the cold-start incurred by the Lambda functions once they have scaled-to-zero. We also plan to adapt OSCAR to minimalistic Kubernetes distribution to move part of the event-driven functionality of OSCAR closer to the edge, by enabling it to run on IoT devices, allowing the composition of workflows that begin the processing on the data gathering device itself.

Chapter 4

A serverless gateway for event-driven machine learning inference in multiple clouds

Published as

Diana M. Naranjo, Sebastián Risco, Germán Moltó and Ignacio Blanquer. (2021) A serverless gateway for event-driven machine learning inference in multiple clouds. Concurrency and Computation: Practice and Experience.
https://doi.org/10.1002/cpe.6728

Abstract

Serverless computing and, in particular, the functions as a service model has become a convincing paradigm for the development and implementation of highly scalable applications in the cloud. This is due to the transparent management of three key functionalities: triggering of functions due to events, automatic provisioning and scalability of resources, and fine-grained pay-per-use. This article presents a serverless web-based scientific gateway to execute the inference phase of previously trained machine learning and artificial intelligence models. The execution of the models is performed both in Amazon Web Services and in on-premises clouds with the OSCAR framework for serverless scientific computing. In both cases, the computing infrastructure grows elastically according to the demand adopting scale-to-zero approaches to minimize costs. The web interface

provides an improved user experience by simplifying the use of the models. The usage of machine learning in a computing platform that can use both on-premises clouds and public clouds constitutes a step forward in the adoption of serverless computing for scientific applications.

4.1 Introduction

The development of cloud computing has introduced a series of service models that provide various abstraction layers with different levels of control. Common service models are IaaS (Infrastructure as a Service), PaaS (Platform as a Service), SaaS (Software as a Service), and FaaS (Functions as a Service).

The FaaS model is a part of serverless computing, which also includes the BaaS (Backend as a Service) category. It is considered an evolution of cloud programming models, with a higher level of abstraction, where the cloud provider dynamically manages the provisioning of resources. Serverless computing, and particularly the FaaS model, has become a paradigm for the deployment of applications in the Cloud, primarily because of the advantages it provides to developers with respect to the adoption of containers and microservices-based architectures [78]. Indeed, one of the fundamental challenges in the transition to serverless computing for a microservices-based architectures is that applications must be designed as a set of functions.

The FaaS model reduces infrastructure costs and developers' time, since they only have to focus on the functionalities of their application and not on the administration of the underlying infrastructure. In this model, applications run in stateless environments called functions that are triggered by certain events, such as the upload of a file to a storage system or an HTTP call, and are managed entirely by the cloud service provider.

The fine-grained pay-per-use model of serverless computing is one of the key elements that has led to its adoption by enterprises. This paradigm allows customers to pay only for the amount of resources used from the public cloud provider for the time they have been used. One of the most attractive potentialities is that the infrastructure provisioned by the public cloud provider dynamically resizes with the execution of multiple invocations of the function. This allows applications to run without worrying about over-provisioning and without the need to provision a specific amount of resources since these are flexible and entirely managed by the public cloud provider.

Large public cloud providers such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP) offer, through the pay-per-use model, storage services, network resources, databases and computational resources, among others. These providers have included support for FaaS services for the definition and execution of functions. This is the case of AWS Lambda, Azure Functions and Google Cloud Functions.

When using platform services from public cloud providers, there is a risk of being dependent on the services and products they offer. Indeed, FaaS APIs and formats strongly differ among providers. This dependency is often referred to as "vendor lock-in", since switching technologies and vendors may be costly. In order to mitigate this risk in the use of the resources for serverless computing from public cloud providers, several open-source frameworks have emerged, as is the case of OpenFaaS [144], Knative [112] and Apache OpenWhisk [25]. In this sense, the emergence of containers and the development of Container Orchestration Platforms (COPS), such as Kubernetes, facilitate the implementation of FaaS models in open-source platforms such as those mentioned above.

In parallel, the development of artificial intelligence and machine learning has led companies to include these types of services due to the wide range of benefits that can affect all aspects of human life, such as customer service, detection of fraud and business intelligence [193]. Artificial intelligence combined with cloud computing is seen as the next step in machine learning automation [176].

Deploying machine learning models on local servers has a certain complexity mainly due to the lack of high-end local computing power, which introduces significant delays in the inference and training processes. In fact it is a reality that regular maintenance and scaling is becoming increasingly complex [46]. In this sense, serverless computing emerges as a profitable and scalable solution that allows addressing the main challenges in terms of excessive resource provisioning and simplifies the implementation of the underlying infrastructure.

In order to address these challenges, this paper introduces a web-based serverless scientific gateway that supports the inference phase of machine learning and artificial intelligence models on dynamically scalable serverless platforms. Execution of the models can be done in public or in on-premises clouds, as specified in the web interface. This platform constitutes an extension of the work carried out by Naranjo et al. [137, 136] presented in Gateways 2020. In our previous work, only the deployment in a public cloud was supported. For this contribution, we have included an analysis of execution times and economic cost of the platform and the deployment in on-premises cloud with the OSCAR framework, thus supporting multi-cloud infrastructures. The fundamental objective of this

platform is to support the inference of machine learning and artificial intelligence models in multi-clouds by abstracting the configuration, management and scaling details of the underlying infrastructure by adopting serverless computing both from on-premises and public clouds.

After the introduction, the structure of the article is as follows. First, section 4.2 introduces the related works in the area of the execution of machine learning and artificial intelligence models on serverless platforms, both in public and on-premises clouds. Next, section 4.3 presents the components and the architecture of the platform. Then, section 4.4 introduces the models integrated in the platform. Later, section 4.5 presents the results of the use cases and section 4.6 discusses the results obtained. Finally, section 4.7 summarizes the main achievements and future works are presented.

4.2 Related Work

Previous research exposes the advantages of the serverless paradigm in scientific computing. This is the case of the work by Spillner et al. [184] which introduces the benefits of adopting the FaaS model for multiple scientific applications such as computer graphics, cryptology, mathematics, and meteorology. A study by Baldini et al. [31] analyzes existing serverless platforms by identifying key features, use cases, and describing technical challenges and open issues. The conclusions of this research indicates that the FaaS model appropriately adapts to a number of distributed applications, including event processing pipelines in compute-intensive applications.

One of the pioneers in the use of the FaaS model was Jonas et al. [108] who introduced the PyWren framework in order to perform Python-based distributed computing on AWS Lambda to support different distributed computing models efficiently. Later, this study was expanded in [177] presenting *numpywren*, a linear algebraic system built on a serverless platform. In addition, LAmbdaPACK is presented, a domain-specific language designed to implement highly parallel linear algebra algorithms in a serverless environment.

A more recent study by Eismann et al. [73] presents a guide for the design of new serverless approaches examining 89 use cases obtained from other scientific literatures. Each case is studied by analyzing different characteristics that include general aspects, but also workloads, applications and requirements. The work carried out by Jindal et al. [107] introduces an extension of FaaS to computing clusters, to support functions across a network of distributed heterogeneous target platforms, called Function Delivery Network (FDN). As a result, the varied

characteristics of the target platform, the possibility of collaborative execution between multiple target platforms, and the data localization provided by FDN are shown. The work done by Mahmoudi and Khazaei [121] introduces SimFaaS, an open-source tool written in Python that allows to simplify the validation process of a performance model developed on serverless public computing platforms. Through SimFaaS it is possible to predict various metrics related to service quality such as cold start, average response time and the probability of rejection of requests that helps to understand the limits of the system and measure the compliance with the Service Level Agreement (SLA) without the need for expensive experiments.

Serverless computing also covers other fields of computing such as the analysis of large amounts of data (Big Data). The work by Giménez-Alventosa et al. [82] presents MARLA (MapReduce on AWS Lambda)[1] a high performance open-source serverless architecture to run MapReduce jobs on AWS Lambda and Amazon S3, without the need for the user to pre-provision the computing infrastructure.

As stated in the introduction, an important element to consider in the adoption of serverless computing is the risk of vendor lock-in with the technologies and services of public cloud providers. In this scenario, it is difficult to migrate to a different provider without substantial cost due to the technical incompatibilities [143]. In order to mitigate this phenomenon, developers have focused on creating open-source solutions such as OpenFaaS [144], Knative [112], Fission [76], Nuclio [139], Apache OpenWhisk [25], and Oracle Cloud Fn [147], to name a few. These platforms support the definition and execution of functions in response to certain events. The difference between them is fundamentally in the programming language they support, the event sources and in the use of an orchestration platform such as Kubernetes.

The study conducted by Hendrickson et al. [100] presents OpenLambda, an open-source platform for running applications and web services based on a serverless architecture. In the work presented by Kaviani et al. [110] Knative compared with other serverless platforms in order to extract a minimal execution model with a common denominator that is close to a unified serverless platform. Palade et al. [149] performed an analysis of four open-source serverless frameworks: Kubeless, Apache OpenWhisk, OpenFaaS and Knative in some typical scenarios related to edge computing and IoT (Internet of Things) networks. The results of this research indicate that Kubeless surpasses the other frameworks in terms of response, time and performance.

[1] MARLA - `https://github.com/grycap/marla`

The work carried out by Li et al. [118] presents an analysis of open-source serverless frameworks taking into account platform design problems that affect performance. They determine that simple autoscaling based on resources or workloads is not adequate to meet the needs of serverless platforms. In the work developed by Benedetti et al. [37] the suitability of a local serverless platform for IoT applications, implemented through OpenFaas, is discussed and analyzed. A performance study is presented taking into account latency and resource consumption for the cold and warm boot deployment mode.

The rise in the development of machine learning and artificial intelligence applications has led to the adoption of the service models available in the cloud. Public cloud providers have included support for artificial intelligence and machine learning applications within their services. Amazon SageMaker [2] for example, is a fully managed platform in AWS that allows users to easily and rapidly create, deploy and train machine learning models.

The article conducted by Corral-Plaza et al. [58] presents an analysis of the main options for machine learning available in the cloud. The work is focused on the BigML [3] platform and Amazon Machine Learning. Another study by Ishakian et al. [101] evaluates the suitability of AWS Lambda to serve lightweight deep learning models. As a result, an analysis is made of how cold start influences processing performance and AWS Lambda storage limits restrict the implementation of larger models. In the work done by Kodandarama et al. [157] the feasibility of implementing the inference phase on a serverless platform using services provided by AWS is studied. Research results show that serverless platforms show promise for implementing the inference phase of machine learning models.

In the work by Bhattacharjee et al.[38] it is presented Barista, a local serverless platform for the implementation of machine learning models based on OpenStack to execute predictions, selecting the configuration of the virtual machine based on the objectives of service level, cost, and time of execution. For its implementation, this system requires a machine powerful enough to support the framework. The articles conducted by Christidis et al. [53, 54] propose a set of optimization techniques for the implementation of machine learning models on a serverless platform, without compromising capacity or performance. The results obtained indicate the feasibility of using serverless platforms in the implementation of machine learning and artificial intelligence models. A recent work presented by Kurz [116] analyzes the feasibility of implementing double machine learning, a method based on the estimation of primary and auxiliary predictive models [52], on AWS Lambda, taking advantage of the high level of parallelism that can be achieved

[2]Amazon SageMaker - `https://aws.amazon.com/sagemaker/`
[3]BigML - `https://bigml.com/`

with serverless computing. In the case study analyzed in the research, an implementation written in Python called *DoubleML-Serverless* is presented, where its usefulness is demonstrated by analyzing the execution times and estimating the costs.

In the work by Ishakian et al. [101] a series of experiments are performed to run Amazon MXNet machine learning models on AWS Lambda. The objective in this research is to measure efficiency in terms of processing time, scalability, and memory used. The results obtained demonstrate that the use of a serverless platform is adequate to obtain the prediction of the models, as long as they are integrated into the AWS platform and that they comply with the limitations of AWS Lambda. Other research presents SerFer [157] as an inference system for machine learning applications in the AWS cloud. In this system the inference is restricted to AlexNet, a convolutional neural network (CNN) [113], and the implementation is based on a system that executes the inference phase in an EC2 instance.

The challenges in the execution of machine learning models are limited memory and compute capacity, together with long execution times. Serverless computing allows the implementation of these applications in a more cost-effective way, especially in the inference phase where large amounts of resources are required in a short execution time. In order to address the main challenges, this document presents a serverless architecture integrated with these type of applications, where the inference phase of machine learning models can be executed in a public cloud or on-premises clouds using serverless computing strategies. Access to the models is implemented through a web-based scientific gateway, which facilitates their use by users without experience in this type of technology. The models implemented in the use cases and the tools used to design the platform are open-source and publicly available in GitHub: models [62], SCAR [97], DEEPaaS [67] and web-based scientific gateway [96].

4.3 Components and architecture

This section introduces the main components used to create the web-based scientific gateway to support the inference phase of machine learning models from multi-clouds based on the serverless model (both public and on-premises clouds). Figure 4.1 shows the components used in this development. Two fundamental deployment methods are identified in the developed platform, a public cloud and an on-premises cloud. DEEPaaS API and the user interface are common components to both deployment methods. On the one hand, SCAR allows the implementation

of the FaaS model in AWS and, on the other hand, OSCAR allows supporting the FaaS model in an on-premises cloud. The following subsections provide further details on the components involved in the scientific gateway.

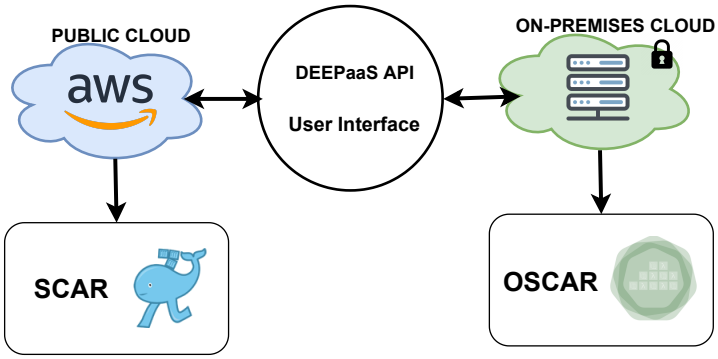


Figure 4.1: Components of the designed architecture.

4.3.1 DEEPaaS API

DEEPaaS API[4] is a software developed in the European DEEP Hybrid-DataCloud[5] project. It is a REST API written in Python which provides simplified access to machine learning, deep learning, and artificial intelligence models. Through HTTP calls, the user has access to the functionalities of the implemented model. The requirements and changes to integrate the applications with the DEEPaaS API are minimal, and this allows an easier interaction with the training and validation functionalities of the models [120].

The inference and training of the models integrated with the DEEPaaS API is done through its REST API. In order to obtain the prediction of the models in the designed platform a new functionality was added to obtain the prediction of the models from the command-line interface. This functionality is a command written in Python where the user specifies certain input values in order to obtain the prediction result through the command-line. This enables support for batch execution of ML models packaged in DEEPaaS API to execute on both high-end HPC supercomputers and batch-based computing installations such as virtual clusters [136].

The command to be executed is *deepaas-predict*. The required options are: `--input-file` and `--output-file` for the input and output files respectively,

[4]DEEPaaS API - `https://github.com/indigo-dc/DEEPaaS`
[5]DEEP Hybrid-Datacloud - `https://deep-hybrid-datacloud.eu/`

and `--content-type` to specify the type of file that is returned in the execution of the command. By default, a JSON file is returned with the result of the prediction, but depending on how the user has integrated the model, other types of files such as JPG or ZIP files can be obtained. Furthermore, as optional elements, users can define `--model-name`, to select a specific model in case of having several models installed in the same environment, and `--url` to define the input file from a URL.

Developing command-line-based tools that connect to a REST API that implement a particular service is a common approach in distributed systems [171]. The functionality incorporated into DEEPaaS API allows extending the field of use to scenarios where both a REST API and the command line can be used.

4.3.2 Web-based Scientific Gateway

The development of a scientific gateway contributes to improve the user experience by allowing the efficient use of the tool and reducing the learning curve. The implemented web interface allows users to perform the inference of files with the machine learning models integrated in the platform, which contributes to the application being used by users who have no experience in the use of those models.

In the web programming environment there are many frameworks and languages that facilitate the work of developers. The development of this web interface is based on the Javascript frameworks VueJS[6] and Vuetify[7]. VueJS is a popular open-source Javascript front-end framework aimed at organizing and simplifying web development, mainly in the development of user interfaces. The use of components is one of the most powerful features of Vue. In large applications it is more efficient to divide the application into small, autonomous, and often reusable components so that development is more adaptable [117].Vuetify is a component library made for VueJS that makes web interface development easy, with each component designed to be modular, responsive, and high-performance. The web interface is compiled as a static web site that is served from an Amazon S3 *bucket* and made publicly available[8].

Authentication to the web can be done through two methods, as Figure 4.2 shows: Amazon Cognito and DEEP IAM. Amazon Cognito is a service offered by AWS that allows user registration, login, and access control in web and mobile applications. In this service, through the *User Pools* [28], a group of users is created where access credentials are assigned. DEEP IAM is an identity provider based

[6]VueJS - `https://vuejs.org/`
[7]Vuetify - `https://vuetifyjs.com/`
[8]Web Interface - `https://scar-deepaas-ui.grycap.net/`

on the OpenID Connect standard that allows existing users in that community to log in into our service without having to register. Both authentication methods are integrated with Amazon Cognito *Identity Pools* [27] (Federated Identities) to obtain temporary credentials from AWS that allow access to other services such as Amazon S3, AWS Lambda, among others.

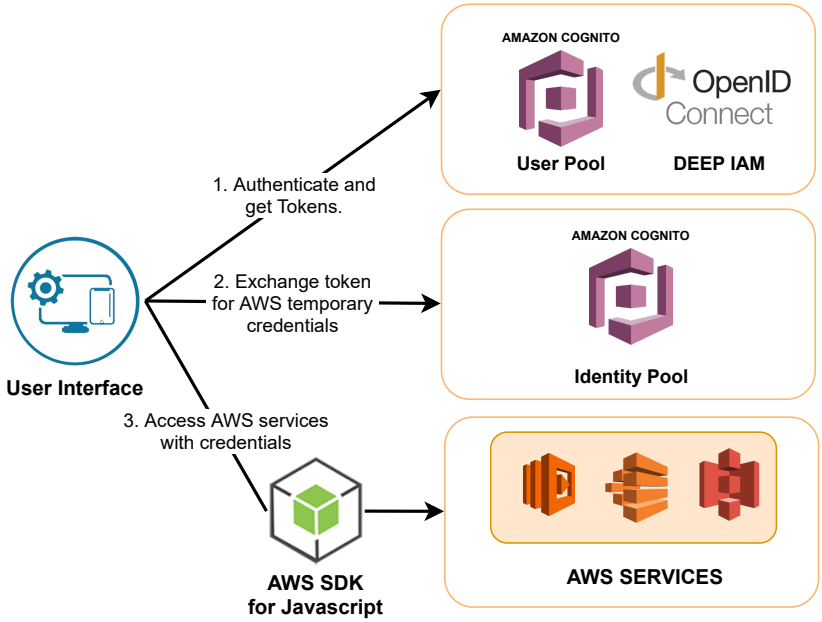


Figure 4.2: High-level authentication and authorization flow in the web interface.

Once authenticated, the user can interact with the different integrated models by uploading, downloading, listing, and deleting the files to be processed or those that are the result of the inference phase. In addition, from the gateway itself, it is possible to check the status of the jobs that are being processed, in case they execute for a long amount of time. All this process is possible through the OSCAR API and AWS services such as Amazon S3, AWS Lambda, AWS Batch and the use of *AWS SDK for JavaScript* that allows access to AWS services from a web interface.

4.3.3 Serverless Frameworks

This section refers to the frameworks that allow to implement the execution of machine learning and artificial intelligence models on AWS and in the on-premises cloud. SCAR is used for deployment in AWS, while OSCAR is used for on-premises clouds. Both tools are developed by our research group and allow us to implement the FaaS model across multi-clouds.

SCAR (Serverless Container-aware ARchitectures)

AWS Lambda is the serverless computing service provided by AWS for the implementation of the FaaS model. This service has certain limitations that restrict the customization of the applications to run. For example, it is not possible to install external packages at runtime because the functions are not executed with root privileges.

One of the solutions would be the use of Docker containers, though Docker requires root access for its installation. To solve this limitation, a container engine able to run Docker containers in the user space is needed. In this sense, tools such as *udocker* [84], *Singularity* [115], *CharlieCloud* [155], *Shifter* [103] or *Podman* [79] play a fundamental role since they precisely allow the execution of Docker images in spaces where there are no root privileges.

SCAR[9] [153] is a tool that uses *udocker* to transparently run container out of Docker images in AWS Lambda as event-driven applications, such as in response to uploading a file to a S3 *bucket* or via an HTTP call to API Gateway[10]. SCAR started to be implemented in 2016 and, in late 2020, AWS Lambda announced native support for running applications packaged on Docker containers. However, this feature does not support images from Docker Hub, the largest repository of Docker images. This shows that as users need more customizable environments, as is the case for scientific applications, cloud providers adapt their services to the new requirements.

The functions in SCAR are created from a YAML [195] file which describes, among other features, the Docker image of the application, the script to be executed in the container, the input and output storage provider, and the execution mode. SCAR allows the implementation of several execution modes: *lambda*, *batch*, and *lambda-batch*. These execution modes determine the service to be executed based on computational requirements.

[9] SCAR - `https://github.com/grycap/scar`
[10] API Gateway - `https://aws.amazon.com/api-gateway/`

In the *lambda* mode, all executions are performed as Lambda function invocations. In addition to the AWS Lambda limitations mentioned above, it is important to add: maximum execution time of 15 minutes, 512MB of ephemeral, potentially shared, storage space, and 10GB of RAM, which affects linearly to the computing capacity. These computing requirements of AWS Lambda led to the emergence of other execution modes in SCAR, as is the case of the integration of AWS Batch into SCAR, as described in the work by Risco et al. [167].

In the *batch* mode, the executions are delegated to AWS Batch, a service that allows the execution of jobs based on Docker containers in an elastic computing cluster of automatically provisioned virtual machines, that grow and shrink according to the execution needs and enable GPU support, a feature not yet available in Lambda. These clusters also have the ability to automatically scale down to zero nodes and provide a perfect fit to the serverless computing model. In the *lambda-batch* mode the execution is carried out in AWS Lambda and in case of a timeout, the job is automatically delegated to AWS Batch.

OSCAR (Open Source Serverless Computing for Data-Processing Applications)

OSCAR[11] is an open-source platform that deploys and integrates several services in order to support event-driven long-running executions within an elastic Kubernetes cluster, accessed through a web interface [152], REST API or CLI. The graphical interface of OSCAR is a static web site for users to view, create, edit, and delete the functions implemented in the platform. From the web interface itself, you can access the storage system, which allows to download and view the input and output files. It is also possible to check the status of the functions and the logs generated in the execution.

In the process of defining the function there are certain parameters, such as the name of the function, the Docker image that contains the application code and the shell-script to be executed to perform the processing, which are required. Other parameters such as environment variables are optional. The function is executed once a file is uploaded into the input storage system, which is processed in an ephemeral container that contains the application code and the configuration specified in the function definition. Once the processing is completed, the result is stored in the output storage system.

A previous research by Naranjo et al. [135] achieved the integration of acceleration devices, such as GPUs, into the OSCAR platform. For this, the rCUDA[12]

[11] OSCAR - `https://github.com/grycap/oscar`
[12] rCUDA - `http://www.rcuda.net/`

[160, 159] tool was used, which allows virtualizing GPU devices that represent physical GPUs in a remote machine. In addition, rCUDA allows the same GPU to be shared by multiple applications accessing them simultaneously. The use of acceleration devices on a serverless platform allows expanding the field of action and inclusion of applications that require intensive computing, such as machine learning and artificial intelligence models.

4.3.4 Architecture

Figure 4.3 shows the proposed architecture for the web-based scientific gateway that supports the inference from machine learning models executed on serverless platforms in multi-clouds.

The integration of the models in AWS is done through SCAR, with functions that are activated once a file is uploaded into the input storage system. In the case of the on-premises cloud, the integration of the models is done through the OSCAR framework. Both deployment methods support three types of storage providers: Amazon S3, MinIO[13], and EGI DataHub [72]. Amazon S3 is the storage system provided by AWS, MinIO is a server-side storage system compatible with the Amazon S3 API, and EGI DataHub is one of the storage systems supported by the EGI Federated Cloud [93], an IaaS-type cloud made up of on-premises and academic clouds that provide computing resources to the scientific research community.

For the integration of the models in the platform, the system administrator must first create the functions through the SCAR client in the case of the deployment method in AWS, and through the OSCAR graphical interface, REST API or CLI, in the case of selecting this deployment method. In both cases, it is necessary to specify the name of the function, the Docker image with the code of the models and the script to be executed in the container. In the case of SCAR, it is also necessary to specify the execution mode, taking into account if the application complies with AWS Lambda's execution time and storage limitations (*lambda* execution mode) or if it does not comply with them (*batch* execution mode). If the duration of the function is unknown, the user can select the *lambda-batch* execution mode, which will execute the job in AWS Lambda and, if a timeout is obtained, a job is automatically delegated to AWS Batch. From this moment on, the functions with the models will be available from AWS and from the OSCAR platform. These functions will be executed every time a file is uploaded to the input storage system.

[13] MinIO - `https://min.io/`

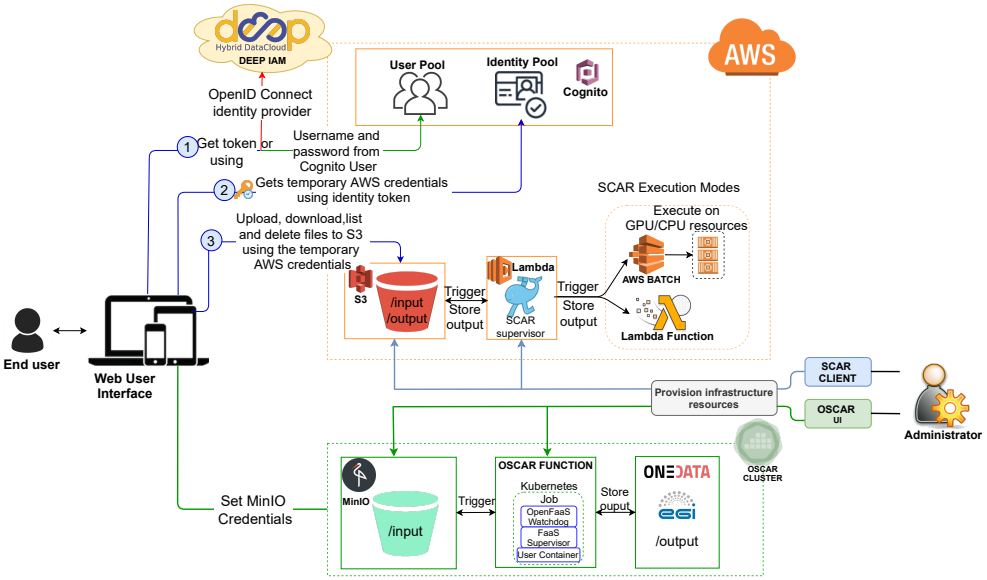


Figure 4.3: Architecture for the integration of Machine Learning models in AWS and an on-premises cloud.

The development of this type of architecture allows the integration of machine learning and artificial intelligence models in a serverless platform that allows execution in a public cloud (AWS) and in an on-premises cloud. The tools used for these deployments, SCAR and OSCAR, are open-source tools that enable the creation of highly parallel event-based file processing serverless applications in environments such as AWS Lambda, AWS Batch, and an on-premises cloud via a dynamically provisioned elastic Kubernetes cluster. The implemented serverless service grows elastically, based on execution needs, and terminates provisioned resources (scale to zero) when they are no longer needed, thus saving costs.

4.4 Use Cases

To evaluate the advantages of the proposed architecture in terms of jobs processed/time unit, several case studies of machine learning models are proposed. In order to be able to integrate other models into the platform, a detailed study of the use of the platform in the inference process of models that are pre-trained and publicly accessible is provided.

Three models from the DEEP Open Catalog and the Darknet model were integrated:

- Audio Classifier [65]: This model allows to perform audio classification with deep learning. It allows to classify through a model previously trained with the AudioSet [85] dataset of 527 high-level classes. To implement the prediction, the model expects as input a URL or an audio file and as output a JSON file with the top 5 predictions is returned.
- Plants species classifier [64]: This model allows to classify plant images among 10 thousand species from the iNaturalist [1] dataset. For the inference phase the model expects as input a URL or an RGB image and returns a JSON file with the top 5 predictions.
- Body Pose Detection [63]: This model allows real-time detection of body poses using deep neural networks. It can be used to estimate single or multiple poses in images or videos. In our case it is used to detect body poses in images. To obtain the prediction the model expects a URL or a RGB image and as a result returns as output the different key points of the body with the corresponding coordinates. This case study obtains an image identifying each of the key points in addition to a JSON file with the result of the classification.
- Darknet [162]: Darknet is an open-source neural network framework written in C and CUDA that supports CPU and GPU computation. This example uses the YOLO (you only look once) library for real-time object detection, such as people, cars, animals, etc.

Figure 4.4 shows, in a simplified way, the processing flow of the files. In the following points a more detailed explanation of the process is made from when a file is uploaded until the prediction result is obtained.

- **Authentication:** The first step to access the classification models is to authenticate on the web. To do this, users can authenticate through their Amazon Cognito credentials or through DEEP IAM, if they have credentials from this identity provider.
- **Method of deployment:** Once the authentication process is completed, the user selects one of the available deployment methods, AWS or OSCAR. In case OSCAR is selected, it is necessary to configure the MinIO credentials in the SETTINGS tab, Figure 4.5.

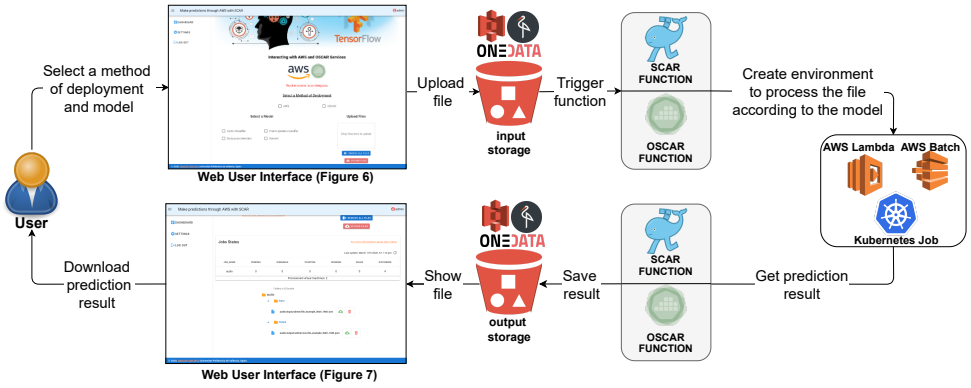


Figure 4.4: Simplified file processing workflow. Functions with different deployment methods selected, either on AWS or in a local cluster with OSCAR.

- **Select Model**: After selecting the deployment method, the user can select one of the available models. When a model is selected, two links of interest to the user are displayed, *Input example for models* which shows an example input file and *Link to the model in the Catalog* link where more information about the model can be obtained. This information allows the user to have specific information about the selected model.

- **Upload Files**: At this point, the user can now upload files from the web interface, in order to trigger the execution of the function corresponding to the selected model, to perform the processing of the file(s). The input and output files are stored in the storage system specified in the function definition. In the storage process, a directory structure has been created where the files are stored in folders named after the selected model and the user, allowing each user to access only her information. This allows the development of a multi-tenant environment and the addition of an activation event for each model independently.

- **Job Status**: The models that are executed in AWS Batch are generally long-running. Therefore, the web interface allows the user to query the status of the jobs that are being processed and, thus, know when the result of the prediction has been obtained.

- **Download Result**: Once the inference process has been performed, the prediction result is stored in the output storage system. In the case of AWS the result is accessible from the web interface and in the case of OSCAR the

result is stored either in MinIO or in the EGI DataHub user space, accessible through a link from the web interface.

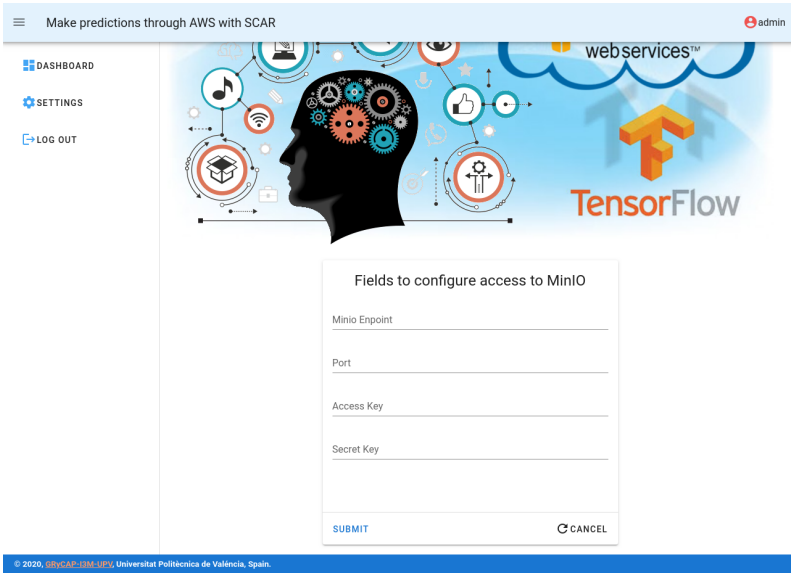


Figure 4.5: Settings tab to configure access to MinIO.

4.5 Results

In order to test the different models integrated in the platform, different experiments were developed. This section analyzes the results obtained in each of the deployment methods and execution modes. Remember that in the case of AWS, with SCAR, the execution can be performed in AWS Lambda and AWS Batch, while in the case of the on-premises cloud OSCAR is used.

Concerning the models that are part of the DEEP Open Catalog, when using AWS the inference process is carried out with the *batch* execution mode, since the size of the images is greater than the limit allowed by AWS Lambda (512 MB). The Darknet model, which complies with AWS Lambda restrictions, runs in the *lambda* execution mode. Functions in OSCAR run as Kubernetes jobs in an on-premises cloud, so they do not have any of these limitations.

Figure 4.6 shows the panel for selecting a deployment method, one of the available models, and the section for uploading the files to the input storage system.

Loading the file generates an event that automatically triggers the function corresponding to the selected model. As mentioned before, the web interface automatically creates the directory structure taking into account the selected model and the authenticated user on the web.

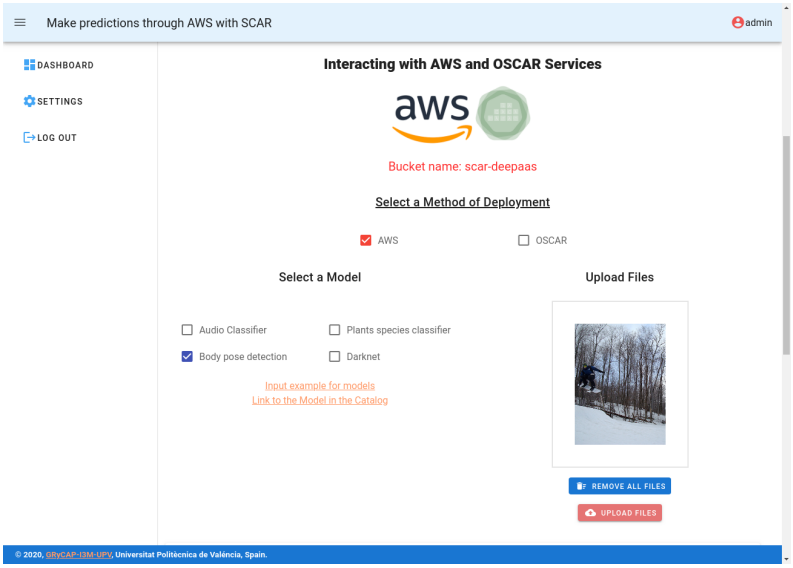


Figure 4.6: *Select Method of Deployment*, *Select Model* and *Upload Files* panels of the Web Interface.

From the web interface it is possible to check the status of the jobs running in AWS Batch, as shown in Figure 4.7. It is important to note that in AWS Batch the deployment of a compute environment can take several minutes because the EC2 instances need to be provisioned and configured; hence the importance of querying the status of the jobs running in this service. The status of the jobs is queried through a Lambda function that communicates with the AWS Batch API, as shown in Figure 4.8. This function is triggered every time the job status is updated from the web interface. In the case of OSCAR, the job status is queried through its API, for jobs in status: *PENDING*, *RUNNABLE*, *STARTING*, *RUNNING*, *FAILED* and *SUCCEDED*. This process allows the user to monitor the life cycle of long-running jobs.

The prediction result is stored in the output storage system specified in the function definition. Like the input files, the prediction result is stored taking into account the user and the selected model. Figure 4.7 also shows the section for interacting with the input and output files of the selected deployment method and

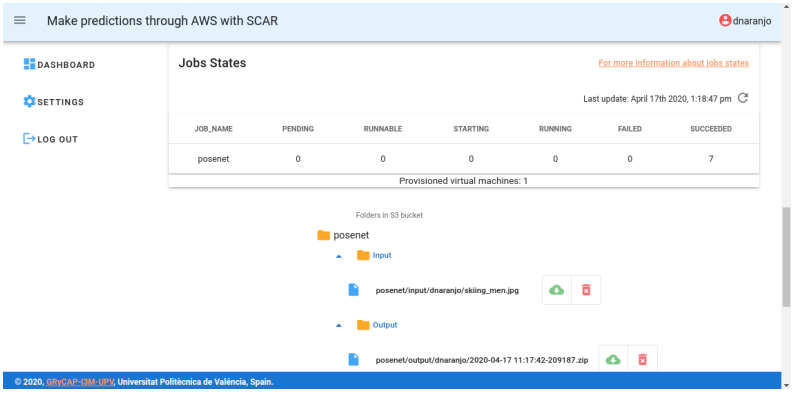


Figure 4.7: Panels to check the status of jobs and stored files.

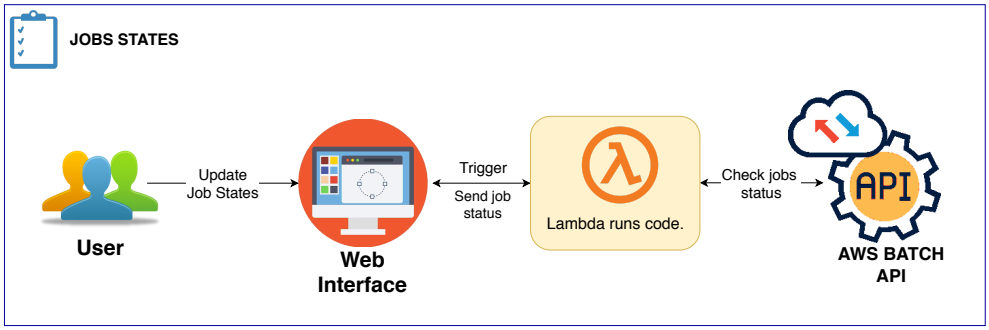


Figure 4.8: Check the status of jobs through a Lambda function.

model. From this section it is possible to download the files or delete them if they are no longer needed. Amazon S3 and EGI DataHub provide high availability, long-term preservation and remote accessibility from anywhere. Alternatively, since MinIO is installed inside the Kubernetes cluster, it provides certain storage capabilities limited to the lifespan of the cluster.

As an example, Figure 4.9 shows the prediction result for the case of the plant species classifier model. On the left (a) the original image is shown and on the right (b) the result of the prediction in JSON format. Also in (c) an example of the search result of the link indicated in red in (b) is shown.

An experiment was carried out that consisted of calculating the processing times for 10 images executed simultaneously in both deployment methods (AWS and OSCAR) and using the *batch* and *lambda* execution modes in the case of AWS.

a) Original Image.

c) Search result of the link marked in red in figure b).


```
{
    "labels":[
        "Aphanostephus ramosissimus",
        "Erigeron karvinskianus",
        "Bellis sylvestris",
        "Bellis perennis",
        "Celmisia discolor"
    ],
    "probabilities":[
        0.3199771046638489,
        0.26108187437057495,
        0.0821433812379837,
        0.07331397384405136,
        0.03924986347556114
    ],
    "links":{
        "Google Images":[
            "https://www.google.es/search?tbm=isch&q=Aphanostephus+ramosissimus",
            "https://www.google.es/search?tbm=isch&q=Erigeron+karvinskianus",
            "https://www.google.es/search?tbm=isch&q=Bellis+sylvestris",
            "https://www.google.es/search?tbm=isch&q=Bellis+perennis",
            "https://www.google.es/search?tbm=isch&q=Celmisia+discolor"
        ],
        "Wikipedia":[
            "https://en.wikipedia.org/wiki/Aphanostephus_ramosissimus",
            "https://en.wikipedia.org/wiki/Erigeron_karvinskianus",
            "https://en.wikipedia.org/wiki/Bellis_sylvestris",
            "https://en.wikipedia.org/wiki/Bellis_perennis",
            "https://en.wikipedia.org/wiki/Celmisia_discolor"
        ]
    }
}
```


b) Prediction result in JSON format.

Figure 4.9: Example of the result obtained using the Plant Species Classifier Model. On the left (a) the original image, on the right (b) the result of the prediction in JSON format, and (c) an example of the search result.

One of the fundamental elements to take into account in a serverless platform with scale to zero is *cold start*. In the case of AWS Batch, it is important to analyze the startup time of the instances.

Figure 4.10 shows the processing times obtained for the case of the Darknet model in AWS Lambda. In this case, it can be seen that the first execution is the one with the longest processing time, because the platform implements scale to zero, which refers to the fact that while the platform is not used, there is no active function, which allows to save costs. Scale to zero introduces the phenomenon of *cold start*, where in the first execution of the function, for SCAR, it is necessary to download the Docker image that contains the application code, start a new execution environment, execute the initialization code and execute the function. Once these steps have been carried out in the first execution, the following ones run faster since the unpacked Docker image may be reused from the ephemeral, potentially shared */tmp* space. It is important to note that cold start can be mitigated by keeping the function always hot at a higher cost. After this first

invocation where the function is already initialized, the rest of the executions typically reuse the configuration mentioned above, which causes them to be processed in a similar time, around 13 seconds.

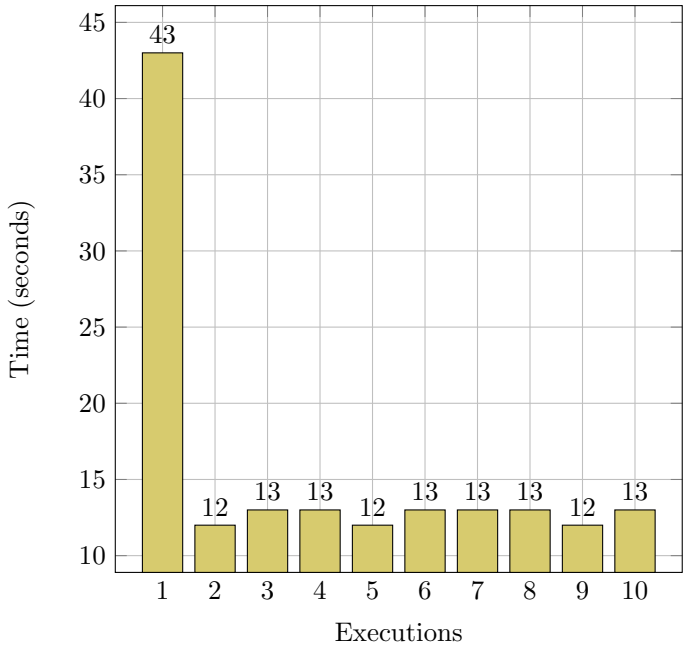


Figure 4.10: Execution times for 10 images with the Darknet model in AWS Lambda.

Figure 4.11 and Table 4.1 show the same experiment performed in AWS Lambda, but in this case the 10 executions are executed taking into account the plant species classifier model, which has to be executed in AWS Batch (*batch* execution mode). In the case of AWS Batch, the compute environment was defined with a maximum of 2 instances of 1CPU and 4GB of RAM. Jobs are sent simultaneously and queued until the scheduler detects that there are resources available and sends them to be processed. Therefore, the processing time is divided into execution time and waiting time. In the graph, the blue bars represent the processing time of the file without taking into account the waiting time in the job queue that is displayed in the yellow bars. The processing time is approximately equal to 14 seconds for each of the invocations.

In the first execution, it can be seen that the waiting time is considerably greater than in the rest of the executions. This behavior is due to the fact that in the

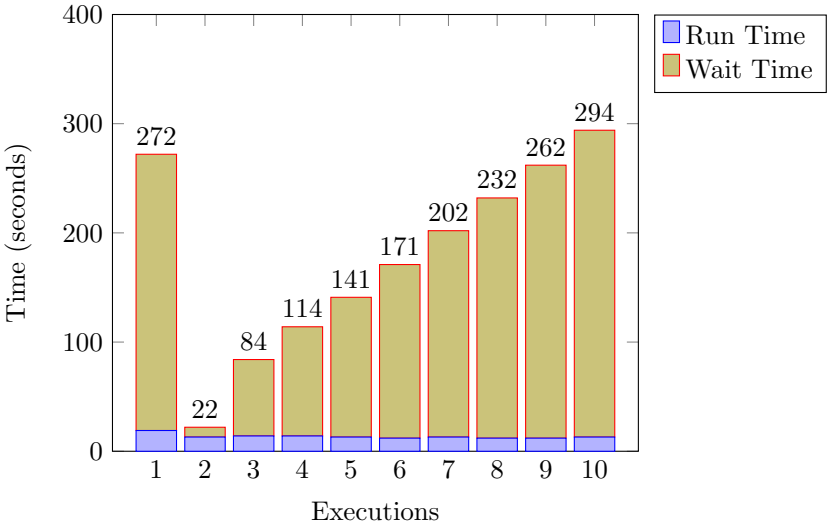


Figure 4.11: Execution times for 10 images with the plant species classifier model in AWS Batch.

first invocation, there is no an active compute environment and, in the same way as the case of AWS Lambda, it is necessary to configure the environment with the selected model. From this point on, the two virtual machines are deployed to serve the workload and the jobs are queued until there are resources are available, hence the waiting times increase. From the moment the jobs to be processed are sent, they go through various states (mentioned in previous sections).

Figure 4.12 and Table 4.2 show the times obtained for the same experiment performed in AWS Lambda and AWS Batch with the plant species classifier model, but in this case implemented in an elastic Kubernetes cluster with the OSCAR framework. The components used for the deployment of this cluster allow it to grow and decrease according to the number of nodes and the workload. In order to have the same environment configured in AWS Batch, a maximum of 2 nodes with 1CPU and 4GB of memory were defined. By default, only one of the nodes is active at startup, so there is one execution slot.

As in the case of AWS Batch the blue bar corresponds to the processing time and the yellow bar to the waiting time in the job queue. The processing time includes the time to download the input file, the processing of the file and the time to upload the result into the output storage system. In all executions, the

Table 4.1: Execution times in AWS Batch, for 10 images using the plant species classifier model.

Execution	Run Time (s)	Wait Time (s)	Total (s)
1	19	253	272
2	13	9	22
3	14	70	84
4	14	100	114
5	13	128	141
6	12	159	171
7	13	189	202
8	12	220	232
9	12	250	262
10	13	281	294

Table 4.2: Execution times in OSCAR, for 10 images using the plant species classifier model.

Execution	Run Time (s)	Wait Time (s)	Total (s)
1	24	44	68
2	24	66	90
3	24	88	112
4	23	109	132
5	23	131	154
6	24	150	174
7	23	173	196
8	23	192	215
9	24	214	238
10	24	237	261

processing time (blue bar) is approximately the same, since the images to be processed have the same characteristics.

The deployment of a new node in OSCAR is done through CLUES [91], an open-source modular elasticity system that allows the introduction of horizontal elasticity capabilities (increase/reduce the number of compute nodes) for cluster-based computing. Once CLUES detects that the workload increases, the new node is deployed (it takes 5 minutes approximately to configure the node), to have more resources available for processing the jobs. Once the workload decreases (around 3 minutes later with no workload), the system itself takes care of shutting down the node that is no longer needed, thus saving electricity. In a cluster with OSCAR by default there is always a node ready.

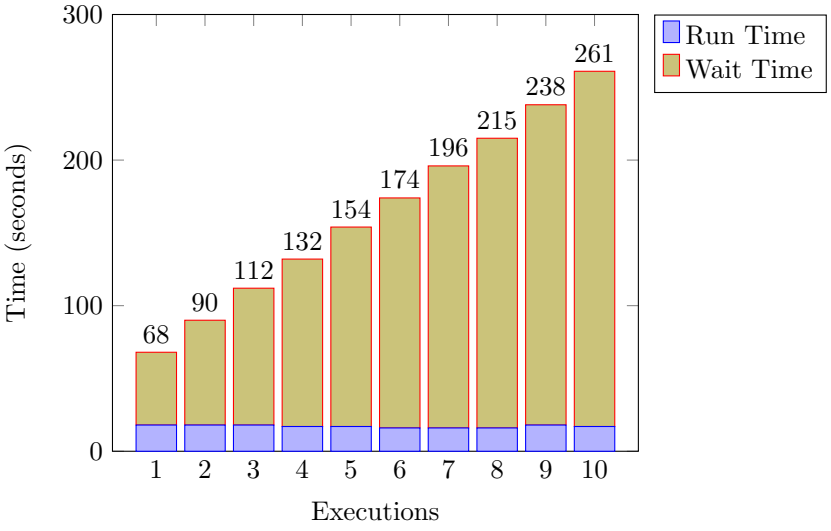


Figure 4.12: Execution times for 10 images with the plant species classifier model in an OSCAR cluster.

In the example implemented in this article, CLUES detects the increase in workload and powers on a new node. Due to the time it takes CLUES to configure the new one, none of the executions run on the new node. When all jobs are submitted simultaneously, they are queued and processed one by one as there is only one active execution slot until CLUES configures the new node, so the waiting time increases from one execution to another. The wait time (yellow bar) of the first execution is due to the pull of the Docker image used in the function. Therefore, between one execution and the next, the wait time increases by the run time (blue bar) of the previous execution. The order of execution of each of the invocations depends on the Kubernetes scheduler. The total execution time of the 10 jobs is similar to those obtained in Batch (300 seconds in AWS Batch and 261 seconds in OSCAR approximately).

The results obtained in each of the environments are different. In the case of AWS Lambda, the effect of the cold start can be seen in the first execution and, from this moment on, the processing time is approximately the same in each of the invocations. In the case of AWS Batch, the first invocation takes longer because there is no compute environment created and it needs to be configured. From this point on the jobs are queued until compute resources are available. The processing

time (not including the time spent waiting in the queue) is approximately the same for each of the executions but AWS Batch provides the ability to run jobs on GPUs. Deploying in an on-premises cloud with OSCAR, saves budget by obtaining execution times similar to public cloud platforms such as AWS and without the restrictions of certain environments such as Lambda. However, the parallelism depends on the underlying computing capacity of said on-premises cloud.

Along with these experiments, an analysis of the costs generated in each of the environments: AWS Lambda, AWS Batch and OSCAR is shown in Table 4.3. The table refers to all services used in AWS and the on-premises cloud. Amazon CloudWatch is used for monitoring, providing the storage for log files. The prices indicated for Amazon S3 and Amazon CloudWatch are general as they depend largely on the size of the files to be processed and the size of the logs generated. For example, to process 1000 images of an average size of 150KB, which are the ones used in this case study, the cost of Amazon S3 would be $0.0034 per month.

It is important to note that most services on AWS have a free usage tier, which would allow costs to be reduced to practically zero, depending on the use of the platform. For example, in the case of AWS Lambda, the free tier includes one million free requests per month and 400,000 GB per seconds of computing time per month, which would be sufficient for this use case.

Costs in an on-premises cloud are highly dependent on the volume and capacity of the platform. This analysis takes into account the costs generated by electricity and the personnel required for the maintenance of the platform. The infrastructure contemplates two nodes where the storage required for incoming and outgoing files is highly dependent on the expected usage of the platform. Using the Azure Total Cost of Ownership (TCO) Calculator [127] it is possible to determine the cost of an infrastructure deployed in the Azure cloud or on an on-premises platform. In this case we use the values for an on-premises platform, and for a computational environment with the same characteristics as those referenced above and taking as reference the 386 seconds of the maximum execution of the 10 invocations (Figure 4.12) we obtain a cost of $0.048 per execution. As shown in Table 4.3 the costs per execution in AWS Lambda are lower, but the limitations introduced by this environment have already been discussed. In the case of AWS Batch there are no limitations as in the case of AWS Lambda, and it also allows the use of acceleration devices like GPUs. In the local cloud we obtain the highest execution costs. However, we can avoid using a commercial public cloud platform if we have access to computational resources such as those provided by EGI Federated Cloud for scientific computing.

Table 4.3: Cost of the platform in a public (AWS) and an on-premises cloud.

	Public Cloud (AWS)				On-premises Cloud
Services	**AWS Lambda**	**AWS Batch**	**Amazon S3**	**Amazon CloudWatch**	**OSCAR**
Resources Provided	2048 MB RAM	m3.medium (1vCPU 4GB)	First 50TB	Store and access log files	2 nodes (1vCPU 4GB)
Prices	$0,0000000333/ms	$0,000018611/s	$0,023 per GB	$0,50 per GB	-
Free Tier	1M free requests per month. 400,000 GB-seconds of compute time per month.	No additional charge for AWS Batch. EC2 resources 12 months free. 750 hours per month	12 months free. 5 GB of standard storage.	CloudWatch stores logs for free for most AWS services (EC2, S3, Lambda, etc.)	-
Execution Time (s)	15.7	294	-	-	261
Cost (per execution)	$0,000000523	$0,005471667	-	-	$0,048

4.6 Discussion

The designed platform is based on several tools to perform the inference phase of machine learning and artificial intelligence models from a web interface on multiple Clouds. From AWS, the deployment of the models is done through SCAR, which allows the execution of Docker images as serverless functions, triggered by events such as uploading a file to an S3 bucket. The execution modes supported by SCAR allow the execution of functions in AWS Lambda or AWS Batch according to the execution characteristics of the application.

Through the *lambda* execution mode the workload can be handled by executing short-lived asynchronous functions. Requests are queued until AWS Lambda provides the on-demand computing capabilities necessary to process invocations in parallel. The *batch* execution mode also handles the processing of long-running resource-intensive tasks. Jobs are stored in the queue until resources are available for processing. Computing environments are created from EC2 instances where applications have access to accelerated resources such as GPUs.

The implementation of machine learning and artificial intelligence models on a serverless on-premises platform, with OSCAR, bring some benefits. On the one hand, it avoids vendor lock-in from the use of resources and technologies from public Cloud providers and, on the other hand, no hardware costs are incurred. The configuration in OSCAR allows infrastructures that grow and shrink elastically, in addition to scaling the worker nodes to zero when not in use, which translates into energy efficiency. Another interesting aspect of OSCAR is the integration with EGI Federated Cloud that provides computational services to researchers under open standards.

The designed platform has included models that are not previously integrated into AWS. Even though they do not comply with the limitations of AWS Lambda, they have been integrated with event-driven scalable services that provide access to computing resources such as GPUs thanks to the *batch* execution mode. The

use of the models is done in a simple way from a web interface, so that users can obtain the result of the prediction without requiring previous skills in the use of AWS or machine learning and artificial intelligence models. One of the main advantages of the proposed platform is the scaling to zero that allows you to pay for services only when they are in use, in addition to automatic scaling when demand increases. The solution proposed in this research facilitates the inference of previously trained machine learning models in the public and on-premises clouds at a reduced cost.

4.7 Conclusions

This paper has focused on the development of a web-based scientific gateway for the inference of machine learning and artificial intelligence models on serverless platforms, using the AWS public cloud and on-premises clouds with OSCAR, by using elastic Kubernetes clusters. For deployment on AWS, SCAR is used, which runs applications packaged in Docker containers, such as functions in AWS Lambda that are triggered in response to certain events. Models whose execution characteristics exceeded AWS Lambda's limits were integrated into AWS Batch. This allowed the use of accelerated devices such as GPUs, a feature not yet available in Lambda.

The implemented development is a step forward in the adoption of the serverless model in the machine learning and artificial intelligence environment. The platform, through the web interface, facilitates the use of the models by users, without the need to define complex jobs. The level of abstraction introduced in this platform allows users with no experience in the AWS cloud and machine learning models to interact without the complexity required.

The processing times obtained for this type of applications compared to other systems are acceptable. Depending on the available resources, the user can select the deployment in the AWS cloud or in an on-premises cloud with the OSCAR framework. The inferences are obtained through serverless services, which implies cost reduction since costs are only generated when resources are used. The designed system constitutes a step forward in the simplification and adoption of machine learning models in serverless systems.

In the availability of machine learning and artificial intelligence models on serverless platforms, there are three fundamental lines of action, in which we intend to continue our research. First, additional models will be incorporated. Second, adaptation to other public Cloud providers will be included. Finally, we will address including GPU support in AWS Lambda by means of remote GPU ac-

celeration. These will allow a more thorough adoption of serverless technology in machine learning and artificial intelligence applications.

Chapter 5

Rescheduling Serverless Workloads Across the Cloud-to-Edge Continuum

Submitted as

Sebastián Risco, Caterina Alarcón, Sergio Langarita, Miguel Caballer and Germán Moltó. (2023) Rescheduling Serverless Workloads Across the Cloud-to-Edge Continuum. Pre-print submitted to: Future Generation Computer Systems.

Abstract

Serverless computing was a breakthrough in Cloud computing due to its high elasticity capabilities and fine-grained pay-per-use model offered by the main public Cloud providers. Meanwhile, open-source serverless platforms supporting the FaaS (Function-as-a-Service) model allow users to take advantage of many of their benefits while operating on on-premises platforms of organizations. This opens the possibility to deploy and exploit them on the different layers of the cloud-to-edge continuum, either on IoT devices located at the Edge (i.e. next to data acquisition devices), in on-premises clusters closer to the data sources (i.e. Fog computing) or directly on the Cloud.

This paper presents two different strategies to mitigate the overload that disparate data ingestion rates may cause in low-powered devices located at the Edge or Fog

layers. To this end, it is proposed the delegation and rescheduling of serverless jobs between the different stages of the cloud-to-edge continuum using an open-source platform for event-driven file processing. To demonstrate the performance of these strategies, a use case for fire detection is proposed that includes processing in the Fog via minified Kubernetes clusters located near the Edge, in the private Cloud via on-premises elastic clusters, and finally in the public Cloud by using the AWS Lambda FaaS service. The results indicate that these strategies can mitigate overloads in use cases involving processing across the cloud-to-edge continuum by coordinating the usage of multiple types of computing resources.

5.1 Introduction

The Cloud-to-Edge Continuum (or Computing Continuum) [35] encompasses a wide variety of components that may include low-powered devices with limited computer resources, on-premises servers with moderate resources, expensive high-performance computers and public cloud platforms. This is in line with the definition by the OpenFog Reference Architecture for Fog Computing, stating that it is a system-level architecture that distributes computing, storage, control and networking functions closer to the users along a continuum [145].

Indeed, the SPEC-RG reference architecture for the edge continuum [104] proposes an architecture for task offloading in the edge continuum according to five computing models: Mist computing, edge computing, multi-access edge computing, fog computing and mobile cloud computing. Mist computing is sometimes used interchangeably with fog computing, even if some authors point to subtle differences [111]. This distributed computing paradigm extends cloud computing capacities into the edge of the network to bring computation closer to the data source and the end devices such as sensors and other IoT devices. In this paradigm, the edge devices collect data that is locally processed at the edge of the network, to the extent that it is possible due to the computing capacity constraints of such devices. Workload is offloaded into the Cloud when additional computing power is required, thus effectively using the cloud-to-edge continuum. This approach offers several benefits:

- Reduced latency: By processing data locally, mist computing reduces the time it takes to transmit data to the cloud and receive a response. This is particularly important for real-time applications that require immediate decision-making.
- Bandwidth optimization: Sending large volumes of data to the cloud can strain network bandwidth. Mist computing filters and processes data locally,

reducing the amount of data that needs to be transmitted to the cloud. Only relevant or summarized data is sent, optimizing bandwidth usage.

- Enhanced privacy and security: Some applications, such as those involving sensitive data or strict privacy requirements, can benefit from keeping data locally and reducing the need for data transfer over public networks. Mist computing allows sensitive data to be processed and analyzed closer to its source, improving privacy and security.

- Offline operation: In scenarios where intermittent connectivity to the cloud is common, mist computing enables devices to continue operating and processing data locally even when disconnected from the cloud. This ensures uninterrupted functionality and allows for offline data analysis, if the computing capacity of the devices is not exceeded.

However, the execution along the cloud-to-edge continuum involves several challenges that need to be addressed, as identified by the work of Mouradian et al. [133]. This work highlights "task scheduling" and "offloading and load redistribution" as key features for computing in scenarios related to fog computing.

In this scenario, serverless has risen in recent years as an event-driven computing paradigm involving services where the service provider manages the underlying computational infrastructure entirely. This has paved the way for the surge of open-source serverless platforms to be deployed on on-premises resources that mimic this abstraction layer for the developers while typically involving Container Orchestration Platforms, such as Kubernetes, which provide seamless resource allocation. This is the case of KNative [112], OpenFaaS [144] and, as addressed in this paper, OSCAR [95]. These platforms provide the required abstractions to execute functions or applications, packaged as Docker images, with dynamic provisioning of resources.

To this aim, this work presents a novel approach for rescheduling workloads on a serverless platform that can run along the cloud-to-edge continuum. This attempts to mitigate the disparate workload distribution across the multiple layers of this continuum to profit from additional computing resources, especially when involving devices with constrained computing resources.

An implementation of the proposed approach is done in the OSCAR[1] open-source serverless platform, together with an assessment of the functionality on a realistic use case on wildfire detection. To the best of the authors' knowledge, this provides the first implementation of a job rescheduling system for serverless computing

[1]OSCAR - `https://oscar.grycap.net`

across the cloud-to-edge continuum, provided as a ready-to-use implementation in an existing open-source framework.

The remainder of the paper is structured as follows. First, section 5.2 discusses the related works. Next, section 5.3 introduces an architecture to support job delegation and rescheduling across event-driven serverless platforms. Later, section 5.4 introduces a use case on serverless fire detection along the cloud-to-edge continuum to assess the benefits of the proposed approach. Finally, section 5.5 summarizes the main achievements and discusses future work.

5.2 Related Work

Several works in the state-of-the-art focus on the scheduling of serverless workloads. For example, the work by Zhang et al. [196] introduces the cost of execution as a requirement for scheduling serverless analytics tasks. They introduce a task scheduler that minimizes execution cost while being Pareto-optimal between cost and job completion time.

Kaffes et al. [109] discuss the limitations of existing scheduling mechanisms for serverless platforms when considering the diverse requirements of applications in terms of burstiness, different execution times and statelessness. They propose a centralized and core-granular scheduler for serverless functions with a global view of the cluster resources.

The usage of serverless computing along the cloud-to-edge continuum has also increased recently. This way, Rausch et al. [158] proposed a serverless platform for building and deploying edge AI applications, thus integrating concepts from AI lifecycle management into the serverless computing model. Based on OpenWhisk composer for workflow composition, they unveiled the lack of support for ARM-based architectures for OpenWhisk.

The cloud-to-edge continuum embraces a diverse plethora of heterogeneous platforms and computer architectures. In this regard, the work by Jindal et al. [107] introduces an extension of FaaS to heterogeneous clusters and to support heterogeneous functions via a network of distributed heterogeneous platforms (Function Delivery Networks). They focus on SLO (Service Level Objective) requirements and energy efficiency, deploying functions on Edge platforms to reduce overall energy consumption. The authors use OpenWhisk, OpenFaaS and Google Cloud Functions.

Sicari et al. [180] build on the concept of scientific workflows using the FaaS computational paradigm to create Serverless workflow-based applications based on a customized Domain-specific Language (DSL) to federate the Cloud-Fog-Edge layers to profit from each computing tier. This is exemplified in the open-source OpenWolf platform, a Serverless workflow engine for native cloud-to-edge continuum, based on OpenFaaS, for function execution and Redis to store the workflow manifests and the execution information for the workflows.

Smirnov et al. [182] introduce Apollo, an orchestration framework for Serverless function compositions that can run across the cloud-to-edge continuum. The framework leverages data locality to benefit cost and performance optimization. It also includes a decentralized orchestration approach where multiple instances can cooperatively orchestrate the application while balancing the workload between the spare resources.

The work by Ferry et al. [74] introduce the SERVERLEss4I0T platform to perform the deployment and maintenance of applications over the cloud-to-edge continuum, but no open-source software is provided.

Unlike previous works, our contribution provides an open-source implementation of the methods described in the paper to support job rescheduling and distribution among multiple service replicas that can execute along the cloud-to-edge continuum. An evaluation and assessment of the benefits of the implementation is done through a use case on wildfire detection run on disparate computing infrastructures on this continuum, involving serverless computing at the edge, on-premises clusters and public cloud infrastructures.

5.3 Proposed Architecture

The work carried out is focused on the extension of the OSCAR [95, 152] platform, an open-source framework for serverless data processing through container-based applications. OSCAR is a cloud-native framework that runs on the Kubernetes [114] container orchestration system, to define serverless services for data processing. As shown in Figure 5.1, it allows the scheduling of Kubernetes jobs for the asynchronous processing of files uploaded to a predefined bucket of the MinIO [128] storage system. These jobs are executed as containers, created out of user-defined Docker images, that run on an elastic Kubernetes cluster that can grow and shrink in terms of the number of nodes depending on the current workload and the limits defined at deployment time.

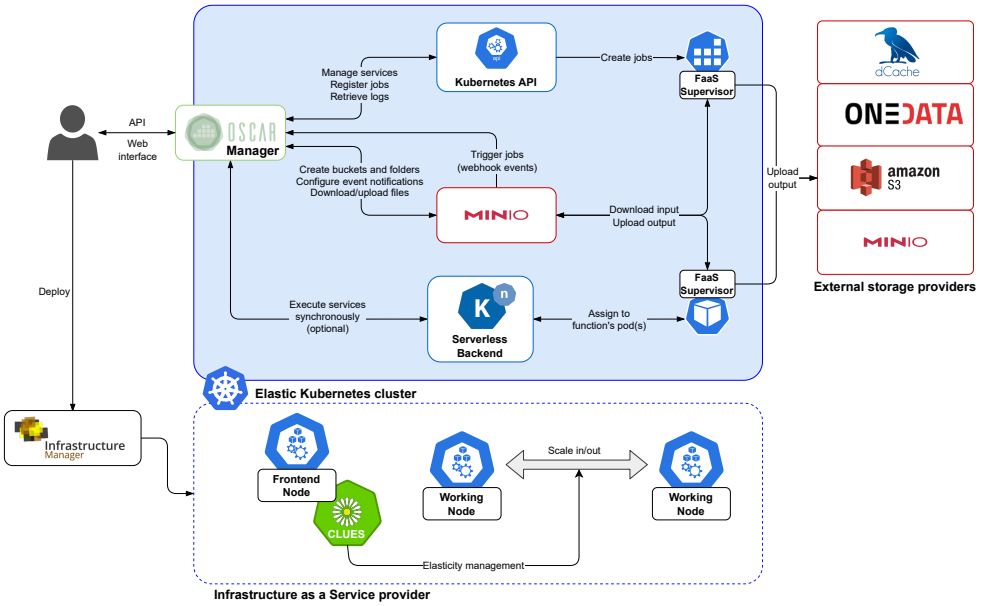


Figure 5.1: Overall architecture of the OSCAR serverless platform

Output files are likewise uploaded to MinIO so users can easily retrieve them or to any supported data storage systems such as Amazon S3, Onedata or dCache. OSCAR also supports the synchronous processing of invocations performed via HTTP requests. For this purpose, the platform is integrated with the Knative [112] Serving framework. However, this study focuses on the asynchronous feature of OSCAR, considering that it is more appropriate for compute-intensive batch tasks, such as inference processes using Artificial Intelligence / Machine Learning (AI/ML) models, as is the use case described in section 5.4.

OSCAR allows the definition of services via a web-based interface or through the Functions Definition Language (FDL)[2] files using the command-line interface. An OSCAR service is mainly characterized by:

- A Docker image available in a container image registry (e.g. Docker Hub or GitHub Container Registry)
- A shell-script that will be executed inside the container created out of the Docker image in order to perform the data processing on the customized execution environment provided by the Docker image.

[2]Functions Definition Language (FDL) - `https://docs.oscar.grycap.net/fdl/`

- A set of computing requirements in terms of vCPUs, RAM and GPUs.
- An input storage bucket that will trigger the execution of the OSCAR service and one or more output storage back-ends on which the output data generated by the service will be stored.

These services can be run on an OSCAR cluster or in AWS Lambda via our development SCAR[3] [153]. AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) to support the Functions as a Service (FaaS) computing paradigm. It allows users to run code in response to certain events (file upload, HTTP request, etc.) without provisioning or managing servers, which is responsibility of AWS. Its highly-elastic features (up to 3000 parallel invocations) and fine-grained billing model (in 1 ms blocks) turned the AWS Lambda into a popular option to develop microservices-based architectures. In turn, SCAR is an open-source tool that pioneered in 2017 the deployment of container-based applications in AWS Lambda when this service still had no native container support (introduced in late 2020). SCAR facilitates the execution of general-purpose applications in AWS Lambda and it provides an automated delegation of jobs into AWS Batch, a managed service to provide automated elastic compute clusters as a service. This allows to use AWS Lambda to execute spiky bursts of short jobs with moderated computing requirements (AWS Lambda invocations cannot run beyond 15 minutes or use more than 10 GiB of RAM) while delegating into AWS Batch jobs that require larger memory or specialized hardware, such as GPUs.

The advantage of using a common Functions Definition Language is the ability to compose serverless workflows across the different layers of the cloud-to-edge-continuum. For example, as described in our previous work by Risco et al. [168], workflows can be composed by services defined on OSCAR platforms configured on lightweight clusters (i.e. on ARM-based devices such as Raspberry Pi) located on the Edge or Fog, on OSCAR clusters in on-premises clouds or Lambda functions in the public Cloud.

A well-known drawback of the cloud-to-edge continuum is the limited computational capacity at the edge. Usually, the devices employed have scarce computing resources, and this can represent a bottleneck in several use cases where the input data ingestion rate may fluctuate depending on external factors. The main goal of this contribution is to mitigate overload problems in these low-powered devices.

Replication and distribution are features required to achieve high availability in a distributed system. Applying this approach in the cloud-to-edge continuum allows to use resources from disparate computing infrastructures, coordinated by a

[3]SCAR - `http://github.com/grycap/scar`

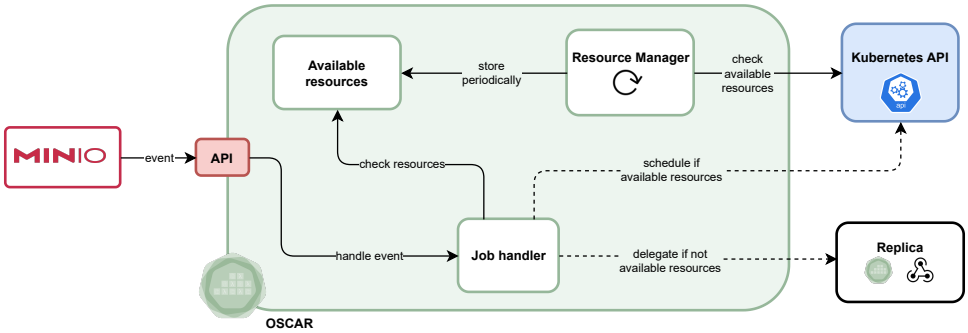


Figure 5.2: Simplified diagram of the Resource Manager component.

distributed control plane that mediates access and resource distribution. Therefore, we introduce the ability to create replicas of serverless services for this work. An OSCAR cluster has the OSCAR Manager component (shown in Figure 5.1), which provides the entry point to trigger the execution of an OSCAR service. The cluster can be deployed on a wide variety of computing infrastructures supported, such as Raspberry Pis, IaaS Clouds and public Clouds. The dynamic deployment on multiple Clouds is achieved thanks to the Infrastructure Manager (IM)[4] [41], an open-source Infrastructure as Code (IaC) tool to provision and configure virtualized computing resources from multiple cloud back-ends. An OSCAR service can have multiple replicas, each one potentially running on a different cluster, with a similar configuration (but each service replica can specify a different number of computational resources).

To this end, two strategies are proposed to reschedule jobs among OSCAR service replicas: *Resource Manager*, described in section 5.3.1, and *Rescheduler*, described in section 5.3.2. Furthermore, section 5.3.3 defines the extension of the Functions Definition Language (FDL) used in SCAR and OSCAR to support this new functionality, as well as details the mechanism for delegating the events that trigger the execution of the jobs.

5.3.1 Resource Manager

Given the capabilities for resource discovery on the nodes of a Kubernetes cluster, a resource manager has been implemented in OSCAR to bypass job scheduling on a cluster that does not have available resources. For this purpose, the Kubernetes core API has been exploited to obtain the status of all active working nodes capa-

[4]Infrastructure Manager - `https://im.egi.eu`

ble of scheduling jobs, and an environment variable `RESOURCE_MANAGER_INTERVAL` has been added to configure the time interval for updating the available resources.

As shown in Figure 5.2, the available resources are stored in a memory variable accessible by the job handler, which is responsible for scheduling jobs in the cluster upon the arrival of new events to the OSCAR API. This handler, therefore, checks if there are available resources in the cluster to schedule the job upon the arrival of a new event. If there are no available resources and the OSCAR service has a replica defined in its specification, it will delegate the event to the replica. If resources are available, the job handler will schedule the job in the current cluster.

It is essential to mention that the Resource Manager is an optional feature in OSCAR and will only work if the `RESOURCE_MANAGER_ENABLE` configuration variable is enabled and there are replicas defined for the active OSCAR service.

5.3.2 Rescheduler

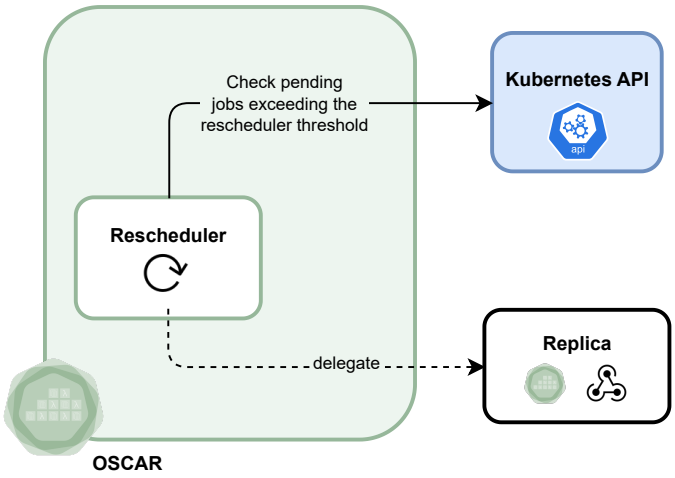


Figure 5.3: Simplified diagram of the Rescheduler component.

Although the Resource Manager allows preventing jobs from being scheduled once a cluster is overloaded, it is possible that during a peak of service invocations, the cluster schedules a large number of jobs in the cluster before the resources available in the cluster are updated. These spikes can generate significant amounts of jobs queued in the Kubernetes scheduler for further processing as resources become available.

To solve this situation, an additional mechanism named Rescheduler has been developed. The Rescheduler is another component to mitigate cluster overloads that is in charge of checking the jobs in "Pending" status in the Kubernetes scheduler. For this purpose, it uses the Kubernetes core API to list the jobs scheduled in the system. It automatically filters them by their status as well as by several labels automatically defined by the OSCAR backend itself.

Figure 5.3 shows how the Rescheduler periodically checks the pending jobs in the cluster that exceed a given threshold. This interval is configurable by means of the `RESCHEDULER_INTERVAL` environment variable. This threshold has a default value configurable at the cluster level through the `RESCHEDULER_THRESHOLD` environment variable. However, as detailed in Figure 5.4, it can be configured for each service via the `rescheduler_threshold` parameter in the FDL. Jobs that exceed the defined threshold will be automatically delegated to a replica by the Rescheduler and, once scheduling is achieved on the replica, will be removed from the current cluster queue.

Just like the Resource Manager, the Rescheduler is an optional feature for OSCAR services and can be enabled or disabled globally through the `RESCHEDULER_ENABLE` environment variable. Furthermore, suppose a service does not have replicas in its definition. In that case, the OSCAR backend will not add the required labels for the Rescheduler to filter the jobs so they can remain in the Kubernetes scheduler queue as long as necessary until free resources are available.

5.3.3 Delegation Mechanism

To support the delegation of events to external clusters or endpoints, the Functions Definition Language (FDL) has been extended to include the concept of *replicas*, as introduced earlier. Multiple replicas can be defined for the same service, so if delegation fails on one replica, there are other replicas to which service invocation can be delegated. The definition of replicas can be done in the FDL through the `replicas` parameter, which is a list of OSCAR service replicas. A priority system has been implemented to choose the replica to delegate in the first place. Users can indicate each replica's priority, with the number 0 as the highest priority and larger integers having a lower priority.

As shown in Figure 5.4, two different types of replicas can be specified. On the one hand, the "oscar" type of replicas are services defined in another OSCAR cluster, being required to indicate the cluster identifier (`cluster_id` parameter) where such service is deployed, as well as its name. The OSCAR-CLI client automatically takes care of embedding the access credentials to the clusters of

the replicas in the configuration of the services so that users do not have to worry about managing them. On the other hand, we also support the delegation of events to HTTP endpoints, which will be sent via POST requests. Support for these endpoints makes it possible to use any FaaS service (such as AWS Lambda) where function invocation via REST APIs can be enabled. Thanks to this support, jobs can be rescheduled between OSCAR clusters, which can run on the edge, on-premises and public Clouds, and self-managed services in the public Cloud such as AWS Lambda functions, which can be exposed via HTTP APIs (using *function URLs* or via API Gateway, as done with the SCAR framework).

Algorithm 1 shows the simplified pseudocode of the delegation mechanism. The first step is to ensure that the list of replicas is sorted by priority to consequently wrap the original event (that triggered the service, such as file upload to MinIO) by adding the identifier of the source cluster. This wrapping is necessary for the replica to know where the event comes from and, in this way, to download the input file, which usually comes from the MinIO storage provider of the source cluster. Then the algorithm proceeds as follows: if the replica type is "oscar", it just checks that the cluster identifier is defined in the configuration (i.e. the cluster's credentials exist under that identifier) and, consequently, the request is prepared with the replica configuration. In the case of "endpoint" type replicas, the HTTP headers defined by the user are added to the request. Finally, the request is sent, and the response is checked. If the response is valid, the algorithm finalises; if not, it continues the loop to try to delegate to another replica in the list.

Algorithm 1 Delegation algorithm pseudocode.

```
Require: replicaList is sorted by priority
  event ← WrapEvent(originalEvent, clusterID)
  for each: replica ∈ replicaList do
      if replica.type = "oscar" then
          if not isClusterDefined(replica) then
              continue
      req ← prepareDelegationRequest(replica, event)
      response ← delegate(req)
      if isValidResponse(response) then
          break
```

Regarding security, all events delegated to other OSCAR clusters are performed using authorisation tokens obtained from the OSCAR configuration API via the basic-auth credentials embedded in the services configuration. Moreover, different authorisation mechanisms can be provided thanks to the support of user-defined

```
---
functions:
  oscar:
  - fog:
      name: fire-detection
      cpu: 1.0
      memory: 1Gi
      image: ghcr.io/grycap/fire-detection
      script: script.sh
      rescheduler_threshold: 15
      replicas:
      - type: oscar
        cluster_id: on-premises
        service_name: fire-detection-replica
        priority: 0
      input:
      - storage_provider: minio.default
        path: fire-detect/input
      output:
      - storage_provider: minio.default
        path: fire-detect/output
      environment:
        Variables:
          AWS_ACCESS_KEY_ID: xxxxxx
          AWS_SECRET_ACCESS_KEY: xxxxxx
          TOPIC_ARN: xxxxxx
  - on-premises:
      name: fire-detection-replica
      cpu: 1.0
      memory: 1Gi
      image: ghcr.io/grycap/fire-detection
      script: script.sh
      rescheduler_threshold: 15
      replicas:
      - type: endpoint
        url: https://lambda-function.example
        headers:
          Authorization: Bearer xxxxxx
        priority: 0
      output:
      - storage_provider: minio.edge
        path: fire-detect/output
      environment:
        Variables:
          AWS_ACCESS_KEY_ID: xxxxxx
          AWS_SECRET_ACCESS_KEY: xxxxxx
          TOPIC_ARN: xxxxxx
```

Figure 5.4: Support for replicas in the Functions Definition Language file.

custom headers in the "endpoint" replica type. In addition, all invocations support the HTTPS protocol, so the traffic between the client and server will be encrypted.

Notice that this approach takes into account the peculiarities of event-driven serverless systems regarding the event delegation across replicas to avoid unnecessary data transfers and the ability to invoke remote HTTP endpoints as the entry point for public serverless services. To assess the benefits of this approach for automated serverless workload redistribution along the cloud-to-edge continuum, we carried out the use case described in the next section.

5.4 Use Case: Serverless Fire Detection Across the cloud-to-edge continuum

Increased wildfires due to rising temperatures are one of the most alarming impacts of global warming [131]. Detecting fires in their early stages is essential to act quickly and minimise the damage caused to forests. However, it is not easy to anticipate these events. While they are often correlated with several meteorological factors, external factors can also provoke them. Surveillance data analysis is an active field of research to prevent this type of situation. Advances in image processing and artificial intelligence enable the development of models capable of detecting fires from images taken from surveillance systems.

This section proposes a use case for processing surveillance images across the cloud-to-edge continuum. For this purpose, an architecture is presented in which the data capture devices would be located at the Edge. These devices would be composed of thermal sensors capable of analysing different meteorological metrics such as temperature or relative humidity and cameras capable of obtaining images periodically. The information obtained by the thermal sensors will be used to detect the level of fire risk at a given time, thus increasing or decreasing the rate of obtaining the images to be processed. To process the images, Minified Kubernetes clusters (using the *k3s* [55] distribution) composed of Raspberry Pis located in the Fog, i.e. near the capture devices, will be used. Each cluster will be in charge of processing images from several cameras. In the experiment described in section 5.4.1, a cluster in the Fog will process images from three cameras. Moreover, the Amazon SNS service will notify the firefighters in case of fire detection.

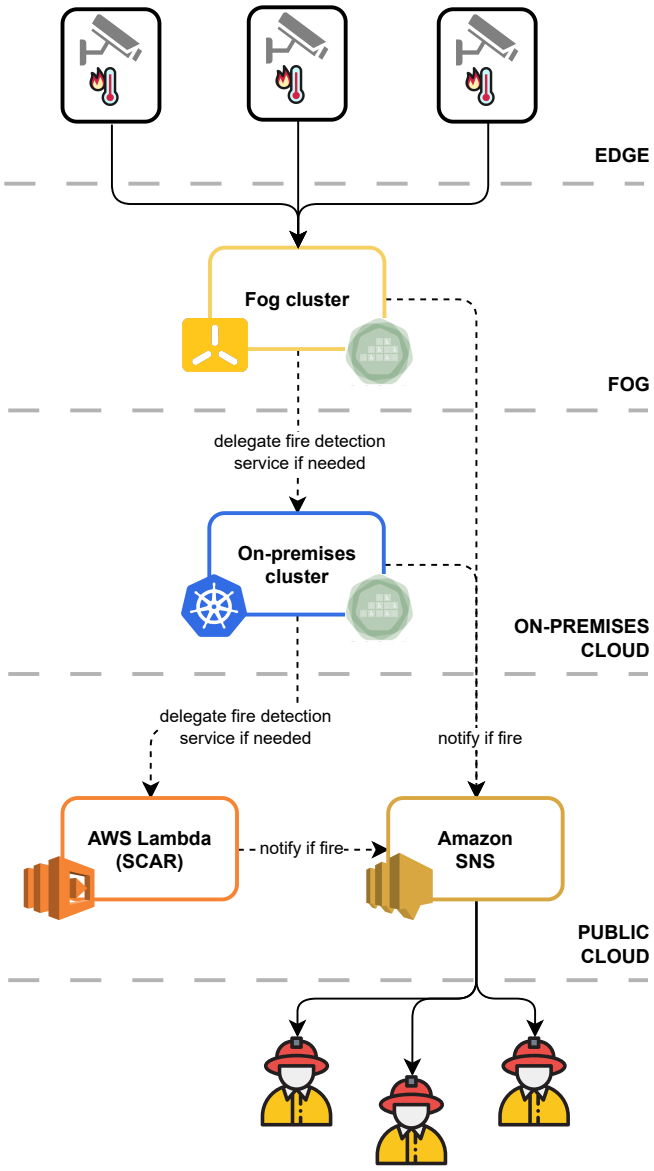


Figure 5.5: Architecture of the use case for fire detection across the cloud-to-edge continuum.

5.4.1 Case Study Design

To evaluate the operability of the new serverless job delegation mechanisms an experiment based on the use case described above has been designed. Although in a real scenario there would be multiple devices at the Edge to capture information, i.e. cameras with thermal sensors in a forest, and multiple Fog clusters to process the data, in the experiment we simulated the ingestion of images from only three cameras to a single Fog cluster. The on-premises cluster has been configured with a single working node to become overloaded quickly. However, it is essential to mention that in a real case, this cluster could have more nodes to process the jobs delegated from multiple Fog clusters. Moreover, OSCAR's deployment can be configured to be elastic, i.e. the number of working nodes can be increased or decreased depending on the existing workload.

The specifications of both the Fog and On-premises clusters are as follows:

- *Fog cluster:* composed of four Raspberry Pi 4 model B, each with 4GB of RAM and a Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz. The Kubernetes minified distribution *k3s* has been used to deploy the components, running one node as the frontend, with the remaining three Raspberry Pi set as working nodes.
- *On-premises cluster:* deployed on an OpenStack-based Cloud, whose underlying infrastructure is composed of 14 Intel Skylake Gold 6130 processors, with 14 cores each, 5.25 TB of RAM and 2 x 10GbE ports and 1 Infiniband port in each node. The virtualized Kubernetes-based OSCAR cluster is configured with one frontend and one working node with eight vCPUs and 32 GB of RAM each, dynamically deployed and configured using the Infrastructure Manager (IM).

The fire detection service is based on the application[5] from the study conducted by Thompson et al. [190], in which a compact convolutional neural network model for non-temporal real-time fire detection was developed and trained. The implementation consists of a simplified ShuffleNetV2 architecture for full-frame binary fire detection and an in-frame classification using superpixel segmentation. The application has been modified to provide a text file with the words "FIRE" or "NOT FIRE" as output. Meanwhile, the script employed for the service generates a compressed (*zip*) file with the text file and the image of the superpixel segmentation.

[5] `https://github.com/NeelBhowmik/efficient-compact-fire-detection-cnn`

Notifications when a fire is detected are sent via the Amazon SNS service [21], whose SDK (Software Development Kit) client has been included in the software container built for the service. AWS credentials can be specified in the services' definition so that notifications can be sent regardless of the cluster in which they are deployed.

The configuration of the services in OSCAR is shown in Figure 5.4. After profiling the application, the services were set up to ensure that each OSCAR service sets 1 CPU and 1 GB of RAM for the jobs created when the service is invoked, both in the Fog and On-premises clusters. Therefore, the number of jobs that can be executed concurrently will be 9 in the Fog cluster and 7 in the On-premises cluster, since the services involved in the OSCAR control plane also use RAM from the underlying virtual infrastructure.

Data ingestion was initially designed using Apache NiFi, a scalable tool for directed graphs of data routing, transformation, and system mediation logic, by creating a dataflow that controls the data ingestion into a MinIO bucket to trigger the OSCAR service. Since NiFi has no available processors to take pictures from the webcam, the GetWebCamera plugin was included[6]. However, we found limitations in the data capture rate by this plugin. Therefore, we decided for the use case to emulate the data ingestion through a Python script that reproduces all the data flow. It gets the image from the virtual web camera and uploads it into the MinIO bucket. The ingestion rate has two phases with a duration of 30 minutes. The first phase ingests three images every 30 seconds. The second one has an ingestion rate of three images every 5 seconds.

To validate the operation of the delegation mechanisms and to benchmark the performance of the developments, the experiment has been carried out in two different scenarios:

- *Scenario 1:* There is the Fog cluster, to which the images that trigger the execution of the fire detection service are uploaded, and the On-premises cluster with the service configured as a replica. When the image ingestion rate increases, the Fog cluster will be overloaded and start delegating jobs to the On-premises cluster. This scenario has been designed to exemplify the use case using on-premises resources, except the SNS service for fire notifications, so there is no need to rely on public Cloud serverless platforms (such as AWS Lambda).
- *Scenario 2:* Same as the previous scenario but with the addition of a replica deployed as a function in AWS Lambda created through SCAR. The function

[6] https://github.com/tspannhw/GetWebCamera

has been made accessible via HTTP requests through the API Gateway service. Therefore, the FDL specifies an additional replica of type "endpoint" with 1 GB of RAM. This scenario has been developed to demonstrate how delegating jobs to higher levels of the cloud-to-edge continuum can be appropriate to profit from the scalability of managed serverless services, specially in time-constrained use cases.

5.4.2 Results and Discussions

This section presents the results obtained after conducting the previously described experiment for the two proposed scenarios. After running the experiment in both scenarios, the average processing time of the fire detection jobs on the three platforms used, i.e. Fog cluster, on-premises cluster and AWS Lambda, has been analysed. Figure 5.6 shows that the Fog cluster is noticeably slower than the other platforms due to the lower computational capacity of the cluster's lightweight devices (Raspberry Pis). Meanwhile, the on-premises cluster is the one that has offered the best performance, followed by AWS Lambda, in which the infrastructure is abstracted from the users, so it is not possible to know precisely the instance type used. It is important to point out that AWS Lambda allocates computational power (e.g. CPU) proportionally to the amount of memory allocated (up to 10 GBs). For the sake of cost-effectiveness, the memory allocated to the Lambda function was only 1 GB, thus resulting in lower performance when compared to the execution in the on-premises cluster.

The worst execution times for all three platforms correspond to the first runs when the software image has not yet been downloaded to the cluster nodes, in the case of OSCAR, and when the functions are not started in AWS Lambda (cold start). This cold start can be mitigated in OSCAR by pre-caching the Docker image in all the nodes of the Kubernetes cluster.

The first phase of image ingestion resulted in a total of 180 jobs being processed in the Fog cluster for both scenarios. In contrast, the second phase generated 1005 images in the first scenario and 1028 images in the second. It is essential to mention that the script employed to simulate the use case waits for the time indicated in the ingestion rate between file uploads but does not take into account the time incurred in uploading images as such, i.e. if any image takes longer to be uploaded due to latency or bandwidth this may affect the total number of images uploaded in the experiment, as it has been the case. However, this does not affect the overall results of the experiment, whose main objective is to analyse the behaviour of the two job delegation mechanisms.

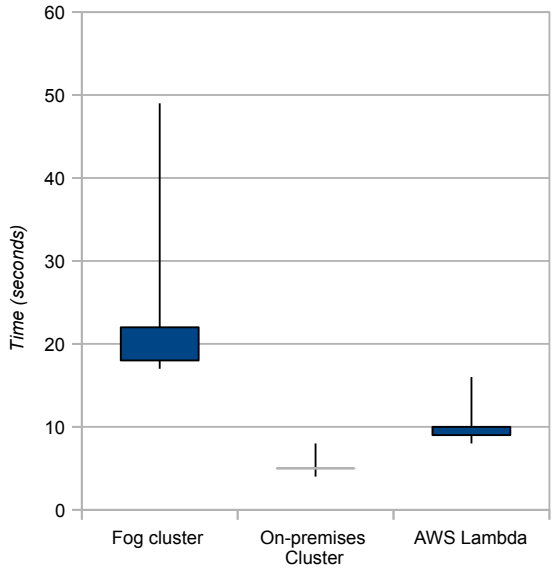


Figure 5.6: Average execution time of the fire detection service on the three platforms employed.

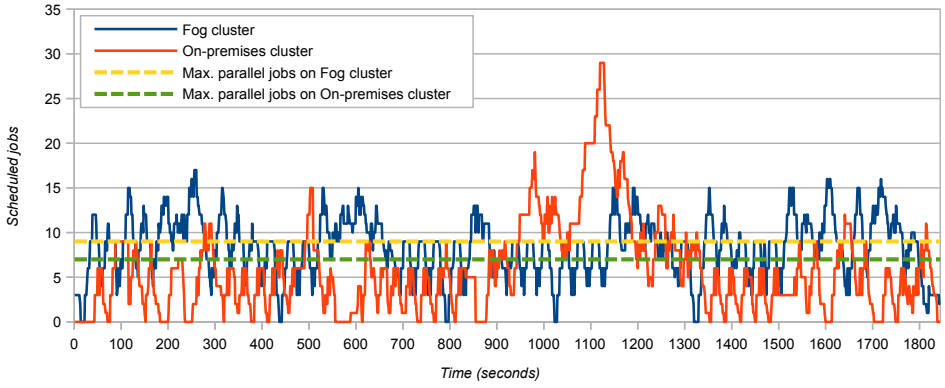


Figure 5.7: Number of scheduled jobs on the fog and the on-premises cluster, and the maximum number of jobs that each cluster can simultaneously execute.

Since the ingestion rate in the first phase is three images every 30 seconds, all the jobs could be processed in the Fog cluster without the rescheduling mechanisms having to delegate any of them. The second phase, however, is where the

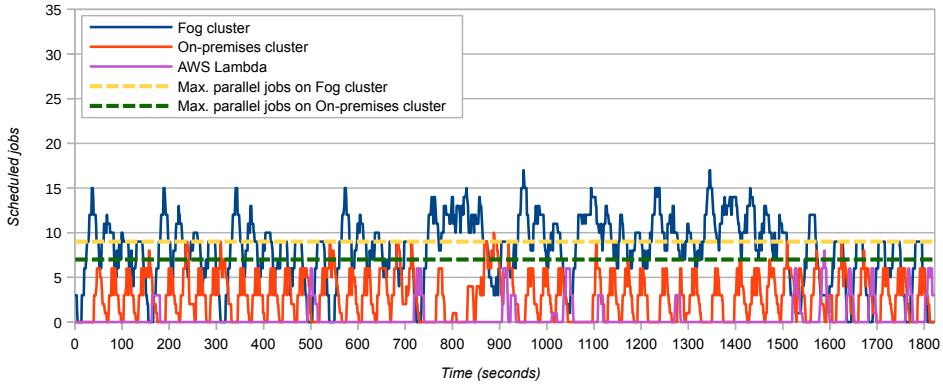


Figure 5.8: Number of scheduled jobs on the fog, the on-premises cluster and AWS Lambda together with the maximum number of jobs that each cluster can simultaneously execute.

behaviour of the delegation systems could be examined due to the large number of images to be processed:

- In scenario 1, a total of 477 jobs have been delegated to the on-premises cluster, being 455 of them delegated via the Resource Manager mechanism and 22 via the Rescheduler. Figure 5.7 details the job scheduling of the second image ingestion phase for the first scenario. As can be seen, load peaks appear when the clusters become saturated; these spikes displayed above the lines of maximum parallel jobs for each cluster mean that the jobs cannot be processed and are kept in the queue until there are free resources available. The peak that occurs at approximately the 1090th second in the on-premises cluster is worth mentioning, in which the cluster is fully saturated as many jobs are scheduled.

- In scenario 2, 538 jobs have been delegated from the Fog cluster to the on-premises cluster, 510 delegated by the Resource Manager and 28 by the Rescheduler. Likewise, the on-premises cluster has delegated 85 jobs to AWS Lambda, 76 by the Resource Manager and 9 by the Rescheduler. As seen in figure 5.8, thanks to the delegation from the on-premises cluster to the public Cloud, the saturation of the on-premises cluster has almost disappeared. Unlike the previous scenario, most load peaks appear only in the Fog cluster. After analysing these results, it can be concluded that reducing the Resource Manager update interval could have further mitigated these workload spikes in the Fog cluster.

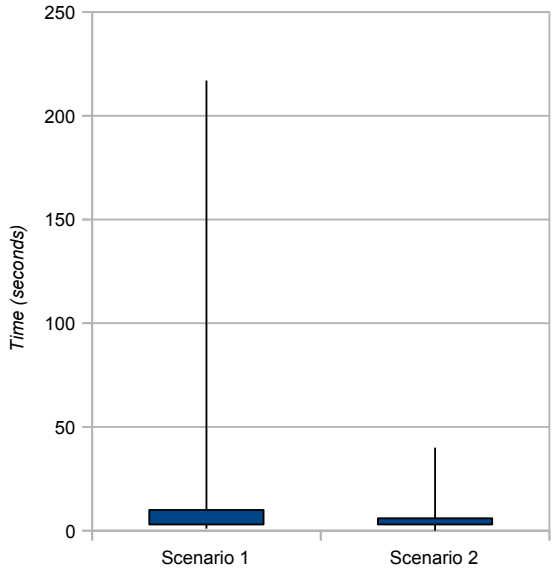


Figure 5.9: Average time that jobs have been queued for each scenario.

Furthermore, an unusual behaviour was found after experimentation: repeated output files were obtained in the second scenario. After analysing the results, it was discovered that the repeated files only appeared in some jobs delegated by the Rescheduler from the on-premises cluster to AWS Lambda. Due to the shorter processing time in this cluster and the default configuration of the Rescheduler, a non-negligible percentage of the jobs delegated to Lambda were also processed in the on-premises cluster. Remarkably, the Rescheduler has been configured in both OSCAR clusters (Fog and On-premises) with the default values, which are 15 seconds for the time interval between checking the jobs in pending state and 30 seconds for the threshold that indicates the maximum time a job can be queued. It is crucial to understand that these times are configurable and should ideally be adjusted according to the job processing time for each use case. Notice that this issue has caused an additional waste of computing resources but it does not affect the main objective, which is to perform the automated delegation of computing when the workload exceeds a certain threshold, along the cloud-to-edge continuum.

To summarise, Figure 5.9 shows the average time jobs have queued in the two scenarios. As it can be appreciated, in scenario 2 this time has decreased notably, proving that combining serverless computing with such strategies to delegate jobs

to replicas along the different layers of the computing continuum can considerably benefit several use cases of near real-time processing where the workload may vary in a non-predictable way.

5.5 Conclusions and Future Work

This paper has presented different strategies for delegating jobs on an open-source serverless file-processing platform that runs on top of Kubernetes. To exemplify the operation of the two delegation mechanisms implemented, a use case was developed based on a pre-existing fire detection AI model and then adapted to the OSCAR platform. The experimentation carried out has allowed, in addition to testing the operation of the rescheduler and the resource manager, the benefits of delegating Serverless jobs to a different on-premises cluster, but also to FaaS services on public cloud providers, thus making use of the different layers of the cloud-to-edge continuum. The results indicate that such systems can be beneficial for several use cases where the workload is unpredictable, and relying only on edge processing devices can significantly limit the ability to handle information quickly.

Future work involves fine-tuning the implementation of the Rescheduler component in order to minimize the execution of duplicate jobs. Also, adapting the Resource Manager mechanism to support additional workload scheduling systems on top of Kubernetes, such as Apache Yunikorn, which is currently being used as a mechanism to limit the amount of resources per-service within an OSCAR cluster. Finally, we plan to introduce support for dynamically changing the replicas of an OSCAR service in order to reflect changes in the underlying infrastructure with the dynamic addition and removal of virtualized computing resources.

Chapter 6

Discussion of the Results

This chapter aims to present an overview of the results achieved during this dissertation. Although the previous four chapters have described the results of each article individually, this chapter summarises all the results achieved, including those that have not been published in peer-reviewed journals. For this purpose, the chapter will be divided into four sections. First, section 6.1 summarises the primary outcomes of the research and developments made. Next, section 6.2 describes two successful cases of the developments that have been carried out as part of this thesis. Then, section 6.3 enumerates the different publications in JCR-indexed research journals, as well as the different conference presentations. Finally, the research projects in which the author has participated during the development of the PhD are presented in section 6.4.

6.1 Summary

All the products developed during this PhD pursue the common goal of exploiting the capabilities of Serverless computing to facilitate the deployment of data processing applications and workflows in the computing continuum. To this end, as presented in the previous chapters, different software applications and tools have been developed and published as open source under the Apache 2.0 licence so that the community can benefit from them.

The SCAR tool for defining container-based data processing functions on top of the FaaS AWS Lambda service previously developed in the GRyCAP[1] research group served as the basis for building all subsequent developments. Initially, the different options for exploiting GPUs as acceleration devices were studied, as these devices are widely used to improve inference times in artificial intelligence models, and since SCAR was integrated with AWS Lambda, it was decided to leverage another service of the company that would allow the use of such devices: AWS Batch. The integration also meant starting to work on the goal of defining a Function Definition Language (FDL) that would support different configurations, which was eventually extended to also support the description of services on the OSCAR platform. Another fundamental point of the integration was improving the *faas-supervisor* component, which manages the input and output of data in the software containers. In this case, improvements were made so that the component could be automatically downloaded within the instances employed by AWS Batch and initialised appropriately in response to the new event types triggered by this platform. Chapter 2 concisely describes the entire development process and the solution's architecture. The tool's behaviour was also demonstrated and analysed using a multimedia processing use case, which consisted of a workflow to process video using artificial intelligence applications to recognise objects and automatically generate subtitles. Results reveal considerable improvements in processing times in the video processing application through GPUs, so this development opens up the spectrum of use cases that can exploit the SCAR tool while retaining all its advantages and ease of use.

Furthermore, this dissertation's central area of research and development has been the OSCAR platform. It started from an initial prototype that was developed as part of the author's master's thesis [166], as well as Alfonso Pérez's doctoral thesis [6]. However, due to the challenging objectives of the present work, it was decided to redesign the application completely following the next steps:

1. A REST API was implemented for creating, reading, updating and deleting (CRUD) Serverless *services* built from containerised software applications. These services are asynchronously triggered by default upon file uploads to the MinIO storage system. Nevertheless, OSCAR also supports both synchronous and asynchronous processing via requests to its API.

2. Integration with the on-premises FaaS platform OpenFaaS [144] as runtime for the execution of synchronous invocations was the starting point. However, it was decided to extend the platform with support for Knative [112] for enhanced scalability. Knative is a native Kubernetes extension for running

[1] `https://www.grycap.upv.es`

Serverless deployments using synchronous event-driven processing, significantly improving resource management in the cluster.

3. Multiple recipes (e.g. Ansible Roles and TOSCA templates) were developed for automating the deployment and configuration of clusters. It is worth mentioning that these recipes include the installation and configuration of the CLUES [91] elasticity system for automatic resource management in the cluster, which is responsible for starting or shutting down instances within the IaaS platform used according to the load present in the system. Thanks to this tool, as well as the scheduling capabilities of the Kubernetes orchestrator system, the fundamental premise of the Serverless paradigm can be fulfilled, in which users do not have to worry about infrastructure management, relying on elasticity systems to support highly parallel processing in a transparent manner.

The most important milestone achieved during the course of this work was the integration between SCAR and OSCAR for the definition of hybrid workflows, where some steps of the flow can be executed in AWS through Lambda functions deployed with SCAR and others in on-premises clusters by means of OSCAR services. This integration was possible thanks to the common development of the *faas-supervisor* component, in which different event formats from the multiple supported data storage systems were defined. In addition, the SCAR tool was modified to communicate with the OSCAR API so that hybrid workflows can be created from the command line interface from a single FDL file that describes the services to be deployed. The results of this integration have been described in the article corresponding to chapter 3, and the use case of mask detection prior to image anonymisation has been used as a reference in the European research project AI-SPRINT, presented in section 6.2.1.

Subsequently, and in connection with the purpose of leveraging the Serverless paradigm on lightweight devices at the Edge of the Continuum, it was determined to address the study of schemes for load balancing the processing in workflows at different levels. For this purpose, it was decided to extend OSCAR and the FDL to support the concept of service replicas. Two mechanisms have been implemented for this purpose: Resource Manager and Rescheduler. The Resource Manager periodically monitors the available resources in the cluster. When the system receives a new event to be processed and does not have the resources to process it, it delegates the job to the replica defined in the FDL file. Meanwhile, the Rescheduler provides a system to establish the maximum time a Serverless job can be queued. The job is automatically delegated to the replica when this threshold is exceeded. It is worth noting that OSCAR services can delegate executions to external services defined via HTTP webhooks, so Lambda functions can be defined

with SCAR to be used as replicas. The details of the design and implementation of these techniques are described in the article from chapter 5, where a use case for fire detection has been developed that relies on replicas in different layers of the Computing Continuum.

Regarding the usability of OSCAR, particular emphasis has been dedicated to developing a graphical interface that facilitates the deployment of generalist container-based applications by less experienced users. This interface allows the creation of services, uploading and downloading files on the MinIO system that triggers the processing, visualisation of logs, and even performing synchronous executions. Moreover, the OSCAR API has been documented[2] to facilitate the integration of external tools. An example of this is the platform defined in the article corresponding to chapter 4, where a gateway has been created to serve inference applications based on artificial intelligence models using SCAR and OSCAR. As a result, a web interface has been achieved from which registered users can efficiently perform inference processes on their files. In this paper, in addition, different applications generated in the framework of the European DEEP-Hybrid DataCloud project have been adapted through an expansion of the DEEPaaS API, detailed in section 4.3.1. This adaptation fulfils the main objective of tailoring scientific use cases to exploit the high parallelism capabilities of Serverless platforms in research projects.

In summary, this thesis has generated an open-source[3] framework for the execution of Serverless container-based applications and workflows across the Computing Continuum. Different challenges have been addressed, always seeking solutions that facilitate the adoption of the Serverless paradigm by non-expert users, trying to optimise and abstract the management of computational resources. Several scientific articles have been published, presentations at different international conferences have been given, and the products developed have been made available to the community so they can be utilised and adapted to different use cases. Notably, the OSCAR platform has been integrated into the EGI Applications on Demand portal and the EOSC Marketplace[4], which increases its visibility and allows the scientific community to benefit from the platform. The following section showcases two successful cases of the outcomes of this work.

[2] `https://docs.oscar.grycap.net/api/`
[3] GRyCAP's GitHub: `https://github.com/grycap`
[4] `https://marketplace.eosc-portal.eu/services/eosc.grycap.oscar`

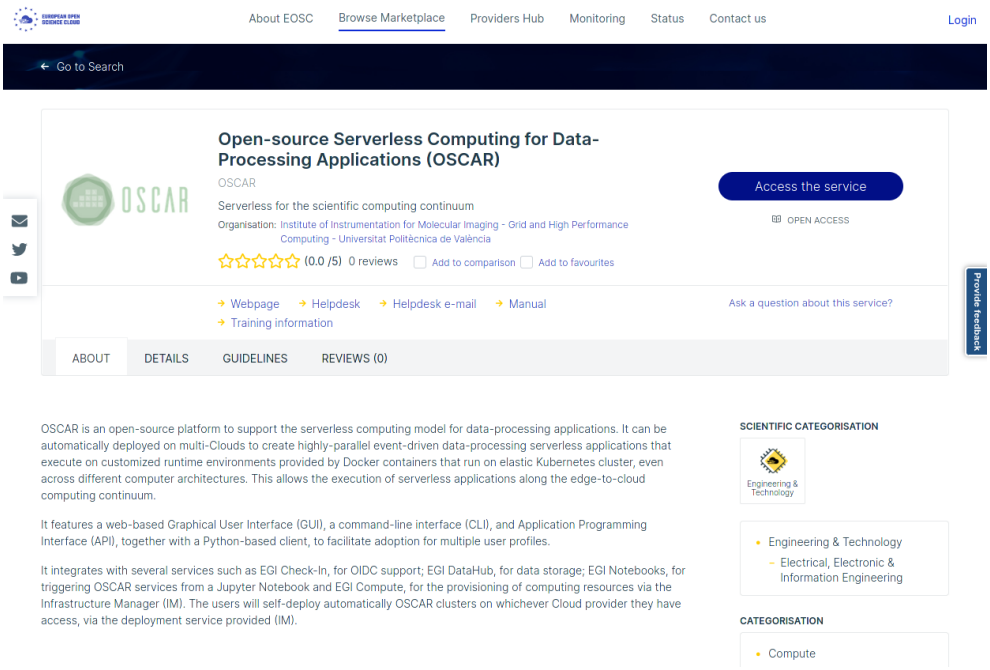


Figure 6.1: OSCAR in the EOSC Marketplace.

6.2 Successful use cases

Although several use cases have been described in the previous chapters to demonstrate the performance of the developed tools and support the experimental work, this section aims to highlight other success cases related to this thesis.

6.2.1 AI-SPRINT project

AI-SPRINT[5] (Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum) is a European research project whose main objective is developing a secure platform to support the creation and serving of artificial intelligence applications in the Computing Continuum. The development of this thesis has been aligned with the project, where SCAR and OSCAR are part of the platform architecture to serve the developed inference applications, as seen in the general diagram of the project shown in Figure 6.2 [130].

[5] `https://www.ai-sprint-project.eu/`

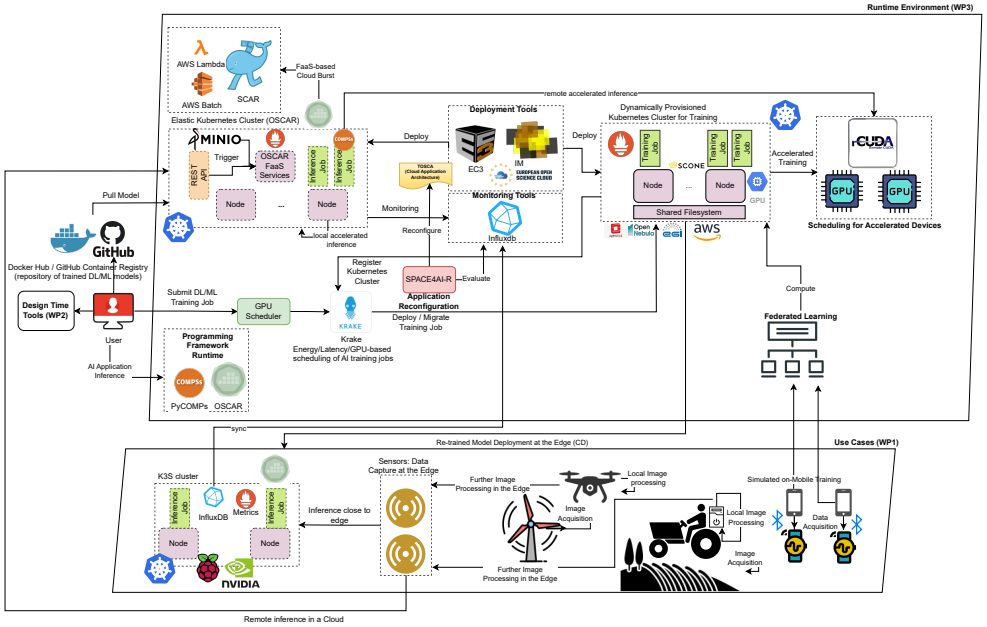


Figure 6.2: General architecture of the AI-SPRINT platform.

Infrastructure Manager is the tool in charge of deploying and configuring the different components of the platform on public cloud providers or on-premises clusters. In the case of OSCAR, IM launches elastic Kubernetes clusters with all the requirements to support the platform. It is essential to mention that different integrations have been made in both the OSCAR application and *faas-supervisor* to support the AI-SPRINT monitoring subsystem. In addition, due to the unique characteristics of some of the project's use cases, OSCAR's ability to run on lightweight devices has also been exploited. An example is the Raspberry PI-based k3s [55] micro-cluster shown in figure 6.3, which was assembled and configured by the author and for which specific configuration recipes were written.

In addition, the SCAR tool has also been explored to delegate executions to AWS Lambda when these cannot be performed on OSCAR clusters due to a lack of resources or specific use case requirements.

The three main use cases of the project are detailed below:

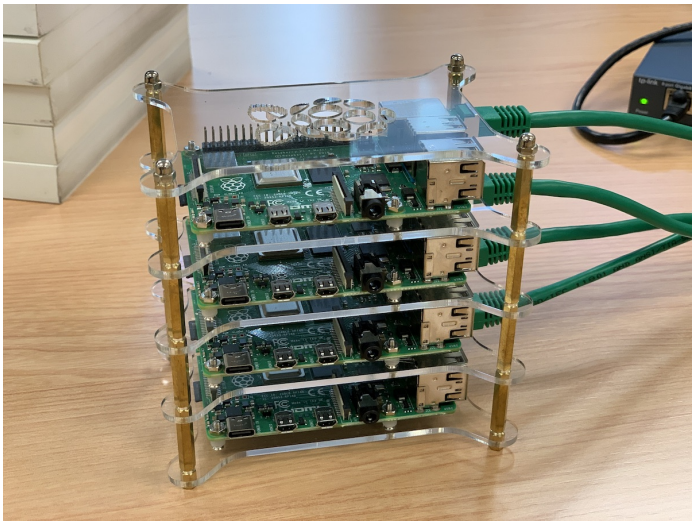


Figure 6.3: Raspberry PI-based micro-cluster to support Serverless computing at the Edge.

- *Personalised Healthcare:* this use case led by the Barcelona Supercomputing Center[6] focuses on developing applications to prevent heart strokes. It uses machine learning techniques on medical data from wearable technology, processed using the COMPSs programming models [30], which have been adapted to run on the OSCAR platform through a software container.

- *Maintenance & Inspection:* this is a use case to automate inspection and predict maintenance of wind turbines. Inference is performed from pictures of the wind turbine propellers taken by drones. These pictures are sent to Edge devices, which are processed further to predict whether maintenance work is needed.

- *Farming 4.0:* this use case aims to exploit digital technologies (internet of things, big data, artificial intelligence) to optimise agricultural processes. Specifically, this scenario involves a computer system and different sensors assembled inside tractors. The sensors capture images as the tractor moves through vineyards, and the artificial intelligence model processes these in near real-time to decide the optimal amount of phytosanitary treatment to be automatically released by the sprayers.

[6] `https://www.bsc.es`

6.2.2 OSCAR as a backend for RDataFrame

RDataFrame [154] is a high-level interface for the ROOT software [40] developed by CERN[7] (Centre Europeen pour la Recherce Nucleaire). ROOT is widely used by the scientific community, especially for the processing of High Energy Physics (HEP) data, such as obtained from experiments performed by the Large Hadron Collider. The decoupling of RDataFrame as a processing interface for ROOT allows custom implementations to support new backends for processing large data sets.

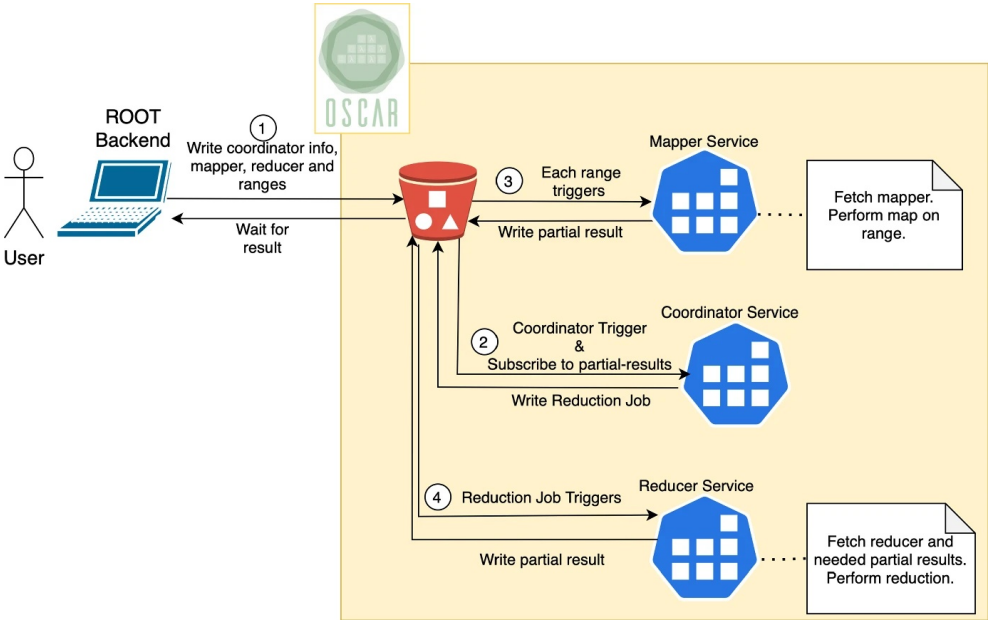


Figure 6.4: Integration of OSCAR as a backend for ROOT for the coordinated reduction process.

Indeed, for the implementation of OSCAR as a backend for RDataFrame, a collaboration has been carried out in the study conducted by Padulano et al. [148], which aims to analyse the use of Serverless tools to support the distributed processing of large amounts of data through ROOT. The implementation has been undertaken by defining the objects needed to implement the RDataFrame interface through the appropriate calls to the OSCAR API and uploading and downloading files to the MinIO storage system. The three main objects to be implemented correspond

[7] `https://home.cern`

to the usual operations of the MapReduce programming model: Mapper, Ranges and Reducer.

Essentially, two software container images have been developed to support the map and reduce operations to be defined as OSCAR services. In addition, the coordinated reduction process has been analysed to optimise the processing. Figure 6.4 [148] shows the interaction between the different components implemented. As can be seen, MinIO is used as the storage system for the initial, partial and final data, as well as the triggering of the three implemented services.

Results suggest that using OSCAR as a Serverless engine for ROOT can be a suitable solution for physicists to deploy their distributed processing environments easily. Although there are points for improvement in the integration, mainly related to the behaviour of the Kubernetes job scheduler, the use of the Serverless paradigm can be one of the best options to achieve the necessary scalability for future HEP experiments, in which the amount of data to be processed will increase considerably.

6.3 Scientific contributions

Several scientific papers have been published, or are in the process of being published, in indexed journals during the execution of this doctoral thesis. The author has also made various presentations at international congresses to disseminate the knowledge obtained. All contributions directly related to the development of this doctoral thesis are listed below.

Presentations at international conferences:

- Sebastián Risco, Alfonso Pérez, Miguel Caballer and Germán Moltó. "Serverless Computing for Data-Processing Across Public and Federated Clouds". In: *10th Iberian Grid Infrastructure Conference (IBERGRID 2019)*.
- Sebastián Risco, Germán Moltó, Diana M. Naranjo and Miguel Caballer. "OSCAR 2.0: Serverless Scientific Computing". In: *EGI Conference 2020*.
- Sebastián Risco, Diana M. Naranjo, Miguel Caballer and Germán Moltó. "Serverless computing across the Cloud continuum for Deep Learning Inference with OSCAR". In: *EGI Conference 2021*.
- Sebastián Risco, Germán Moltó, Miguel Caballer, Caterina Alarcón and Sergio Langarita. "Serverless workflows along the computing continuum

with OSCAR/SCAR: Use cases from AI/ML inference". In: *EGI Conference 2022*.

Journal papers:

- Diana M. Naranjo et al. "Accelerated serverless computing based on GPU virtualization". In: *Journal of Parallel and Distributed Computing* 139 (2020), pp. 32–42. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.01.004
- Sebastián Risco and Germán Moltó. "GPU-Enabled Serverless Workflows for Efficient Multimedia Processing". In: *Applied Sciences* 11.4 (Feb. 2021), p. 1438. ISSN: 2076-3417. DOI: 10.3390/app11041438
- Sebastián Risco et al. "Serverless Workflows for Containerised Applications in the Cloud Continuum". In: *Journal of Grid Computing* (2021). ISSN: 1572-9184. DOI: 10.1007/s10723-021-09570-2
- Diana M. Naranjo et al. "A serverless gateway for event-driven machine learning inference in multiple clouds". In: *Concurrency and Computation: Practice and Experience* 35.18 (2021), e6728. ISSN: 1532-0634. DOI: 10.1002/cpe.6728
- Sebastián Risco et al. "Rescheduling Serverless Workloads Across the Cloud-to-Edge Continuum." Pre-print submitted to: *Future Generation Computer Systems* (2023).

6.4 Research projects

This thesis has been performed in the framework of different national and European projects where the author has contributed, and the developments undertaken have been reflected. All the related research projects are listed below:

National research projects:

- BigCLOE. *Computación Big Data y de Altas Prestaciones sobre Multi-Clouds Elásticos.* Grant number: TIN2016-79951-R. "Retos de Investigación" Funded by Ministerio de Economía, Industria y Competitividad of Spain. Duration: 30/12/2016 to 29/12/2020.
- SERCLOCO. *Serverless Scientific Computing Across the Hybrid Cloud Continuum.* Grant number: PID2020-113126RB-I00. "Retos de Investigación" Funded by Ministerio de Ciencia e Innovación of Spain. Duration: 01/09/2021 to 01/09/2025.

- OSCARISER. *Open Serverless Computing for the Adoption of Rapid Innovation on Secure Enterprise-ready Resources.* Grant number: PDC2021-120844-I00. "Prueba de Concepto" Funded by Ministerio de Ciencia e Innovación of Spain and by "European Union NextGenerationEU". Duration: 01/12/2023 to 31/05/2024.

European research projects:

- *Elastic Serverless Platform for High Throughput Computing Scientific Applications.* Funded by EGI Strategic and Innovation Fund. Duration: 13/03/2019 to 12/09/2019.
- AI-SPRINT. *AI in Secure Privacy-Preserving Computing Continuum.* Grant number: 101016577. "Horizon 2020" Funded by the European Commission. Duration: 01/01/2021 to 31/12/2023.
- AI4EOSC. *Artificial Intelligence for the European Open Science Cloud.* Grant number: 101058593. "Horizon Europe" Funded by the European Commission. Duration: 01/09/2022 to 31/08/2025.
- InterTwin. *An interdisciplinary Digital Twin Engine for science.* Grant number: 101058386. "Horizon Europe" Funded by the European Commission. Duration: 01/09/2022 to 31/08/2025.

Chapter 7

Conclusions

Abstracting infrastructure management and maintenance through the Serverless paradigm has excellent advantages for agile application development. This thesis has presented a set of strategies and tools for users to quickly implement highly scalable event-driven containerised applications in public and private clouds. As a result of the integration effort, it has been possible to combine the different tools developed to compose highly parallelisable workflows for file processing across the different layers of the Computing Continuum.

First, the OSCAR platform has been completely redesigned to be natively integrated with the Kubernetes container orchestrator system. The Go programming language has been used to achieve this, which features interfaces to quickly interact with the Kubernetes API and the rest of the components. In addition, the main application has been packaged in a lightweight, easily distributable software container, which can be run seamlessly on lightweight devices such as Raspberry PIs. In terms of the web interface, it has also been drastically improved. As a result, users can easily define services for Serverless file processing simply and without extensive programming knowledge. Furthermore, multiple recipes for automating the platform deployment have been developed and integrated into the web interface of the Infrastructure Manager service itself. Another important aspect to highlight is the native GPU support provided by the Kubernetes job interface, which allows users to take advantage of GPUs in OSCAR services by simply enabling a parameter through the web interface or in the FDL file. All

these refinements allow the scientific community to deploy their own OSCAR cluster and benefit from the platform to serve their Serverless workflows.

Second, new functionalities have been introduced to the SCAR tool in order to support acceleration devices such as GPUs through the integration with the AWS Batch service. The application has also been modified to support the creation of hybrid workflows (i.e. using Lambda functions deployed on the AWS service or OSCAR services, which clusters can be deployed on different IaaS providers, either public or on-premises). For this purpose, a Serverless Function Definition Language (FDL) has been created and gradually extended, which allows the composition of functions simply using YAML files. This FDL is not only supported by SCAR, but a command line client for OSCAR, called OSCAR-CLI[1], has also been developed.

Third, techniques for balancing Serverless workloads across the different layers of the Computing Continuum have been investigated to improve processing times. As a result, new functionalities have been added to the OSCAR platform to delegate jobs between different clusters or to FaaS services from public Cloud providers. This extension allows to speed up processing in the face of significant workload peaks, which can occur especially in workflows that involve computational load on lightweight devices located at the Edge of the Continuum.

Fourth, multiple use cases have been developed, and others created by the community have been adapted to exemplify the possibilities of all the software developed during this doctoral thesis. Extensive documentation has been generated, and we have participated in several national and European research projects. This participation has allowed the continuous improvement of the tools developed and the dissemination of the results obtained during this research. In addition, the OSCAR platform has been integrated into the EOSC Marketplace and the EGI Applications on Demand portal, which ensures the continuity of the projects and opens the door to carrying out new studies on them.

Finally, it can be stated that all the objectives have been successfully achieved. From a personal point of view, the author has been able to participate in numerous projects and attend multiple international conferences, collaborating with great researchers and learning cutting-edge technologies. Furthermore, all the software resulting from this thesis has been freely published so the community can use and contribute to it.

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

7.1 Future work

Future work involves the design and implementation of a centralised Serverless control panel from which users can deploy their clusters along the Continuum, define and orchestrate workflows on top of them, and benefit from traceability systems to monitor the status of executions better and thus improve development and troubleshooting processes for the applications deployed. To this end, the tools described in Chapter 4 could be adapted to create the web interface, which could be integrated with the Infrastructure Manager to automatically deploy and configure the clusters. This control plane would improve the platform's usability, primarily when used to define hybrid workflows by less experienced users.

A further enhancement to the SCAR tool would be the extension to support other cloud providers. Since there are already Serverless services to provision container-based applications in the leading public cloud providers, SCAR would only have to be updated to communicate with their APIs and add these new configurations in the Function Definition Language. In addition, the *faas-supervisor* component could also be extended to support new event sources and storage backends, such as dCache [60].

To conclude, another potential breakthrough for the OSCAR platform is the study of different scheduling algorithms for workloads. To this end, we could explore recent alternate schedulers to add granularity to resource management on Kubernetes, examples of which are Apache YuniKorn [24] and Kueue [56]. Integrating such schedulers would mean greater control in managing the resources assigned to each service and allow the prioritisation of some services over others, adding the concept of *queues* to the platform. Furthermore, multi-tenant support is considered a key goal to allow the shared use of OSCAR clusters by different users, isolating their configuration and adding the concept of cluster administrators. This would allow institutions to deploy clusters to be shared among their staff, implying improvements in resource management and infrastructure maintenance.

Bibliography

[1] *A Community for Naturalists · iNaturalist*. URL: https://www.inaturalist.org/.

[2] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] Istemi Ekin Akkus et al. "SAND: Towards high-performance serverless computing". In: *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*. 2020, pp. 923–935. ISBN: 978-1-939133-02-1. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[4] Eiman Al Nuaimi et al. "Applications of big data to smart cities". In: *Journal of Internet Services and Applications* 6.1 (Aug. 2015). Publisher: Springer-Verlag London Ltd, p. 25. ISSN: 1867-4828. DOI: 10.1186/s13174-015-0041-5.

[5] Carlos de Alfonso et al. "Multi-elastic Datacenters: Auto-scaled Virtual Clusters on Energy-Aware Physical Infrastructures". In: *Journal of Grid Computing* 17.1 (Mar. 2019), pp. 191–204. ISSN: 1570-7873. DOI: 10.1007/s10723-018-9449-z.

[6] Alfonso Pérez. "Advanced Elastic Platforms for High Throughput Computing on Container-based and Serverless Infrastructures". PhD thesis. Universitat Politècnica de València. URL: https://riunet.upv.es/handle/10251/146365.

[7] Alibaba. *What Is Edge Computing? Advantages of Edge Computing - Alibaba Cloud Knowledge Base*. URL: https://www.alibabacloud.com/knowledge/what-is-edge-computing.

[8] Amazon. *Amazon Simple Storage Service (Amazon S3)*. URL: http://aws.amazon.com/s3/.

[9] Amazon Web Services. *Amazon API Gateway*. Publication Title: Amazon Web Services, Inc. URL: https://aws.amazon.com/api-gateway/.

[10] Amazon Web Services. *Amazon CloudWatch - Application and Infrastructure Monitoring*. URL: https://aws.amazon.com/cloudwatch/.

[11] Amazon Web Services. *Amazon ECS - Run containerized applications in production*. URL: https://aws.amazon.com/ecs/.

[12] Amazon Web Services. *Amazon Elastic Compute Cloud (EC2)*. URL: https://aws.amazon.com/ec2/.

[13] Amazon Web Services. *Amazon Simple Queue Service*. URL: https://aws.amazon.com/sqs/.

[14] Amazon Web Services. *AWS Batch — Easy and Efficient Batch Computing Capabilities*. URL: https://aws.amazon.com/batch/.

[15] Amazon Web Services. *AWS Lambda - Serverless Compute*. URL: https://aws.amazon.com/lambda/.

[16] Amazon Web Services. *AWS SDK for Python*. URL: https://aws.amazon.com/sdk-for-python/.

[17] Amazon Web Services. *Fast NoSQL Key-Value Database – Amazon DynamoDB*. URL: https://aws.amazon.com/dynamodb/.

[18] Amazon Web Services. *Firecracker*. URL: https://firecracker-microvm.github.io/.

[19] Amazon Web Services. *Lambda function scaling - AWS Lambda*. URL: https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

[20] Amazon Web Services. *Managed Kubernetes Service – Amazon EKS*. URL: https://aws.amazon.com/eks/.

[21] Amazon Web Services. *Push Notification Service - Amazon Simple Notification Service (SNS)*. URL: https://aws.amazon.com/sns/.

[22] Amazon Web Services. *Serverless Compute Engine – AWS Fargate*. URL: https://aws.amazon.com/fargate/.

[23] Apache. *Apache License, Version 2.0*. URL: https://www.apache.org/licenses/LICENSE-2.0.html.

[24] Apache. *Apache YuniKorn - Unleash the power of resource scheduling for running Big Data & ML on Kubernetes!* URL: https://yunikorn.apache.org/.

[25] Apache. *OpenWhisk - Open Source Serverless Cloud Platform*. URL: https://openwhisk.apache.org/.

[26] Argo. *Workflows & Pipelines*. URL: https://argoproj.github.io/projects/argo/.

[27] AWS. *Amazon Cognito Idenity Pools*. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html.

[28] AWS. *Amazon Cognito User Pools*. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html.

[29] AWS. *New for AWS Lambda – Container Image Support*. URL: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/.

[30] Rosa M. Badia et al. “COMP Superscalar, an interoperable programming framework”. In: *SoftwareX* 3 (Dec. 2015). Publisher: Elsevier. ISSN: 2352-7110. DOI: 10.1016/j.softx.2015.10.004.

[31] Ioana Baldini et al. “Serverless computing: Current trends and open problems”. In: *Research Advances in Cloud Computing*. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1.

[32] Ioana Baldini et al. “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*. ISSN: 15232867. New York, New York, USA: ACM Press, 2017, pp. 89–103. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133855.

[33] Daniel Balouek-Thomert et al. “Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows”. In: *International Journal of High Performance Computing Applications* 33.6 (Nov. 2019). Publisher: SAGE Publications Inc., pp. 1159–1174. ISSN: 17412846. DOI: 10.1177/1094342019877383.

[34] L. Baresi et al. “A unified model for the mobile-edge-cloud continuum”. In: *ACM Transactions on Internet Technology* 19.2 (Apr. 2019). Publisher: Association for Computing Machinery, pp. 1–21. ISSN: 15576051. DOI: 10.1145/3226644.

[35] Pete Beckman et al. “Harnessing the Computing Continuum for Programming Our World”. In: *Fog Computing*. Wiley, May 2020, pp. 215–230. DOI: 10.1002/9781119551713.ch7.

[36] Juan Pablo Bello, Charlie Mydlarz, and Justin Salamon. “Sound Analysis in Smart Cities”. In: *Computational Analysis of Sound Scenes and Events*. Cham: Springer International Publishing, Sept. 2018, pp. 373–397. ISBN: 978-3-319-63450-0. DOI: 10.1007/978-3-319-63450-0_13.

[37] Priscilla Benedetti et al. “Experimental Analysis of the Application of Serverless Computing to IoT Platforms”. In: *Sensors* 21.3 (2021). ISSN: 1424-8220. DOI: 10.3390/s21030928.

[38] A. Bhattacharjee et al. “BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services”. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019 IEEE International Conference on Cloud Engineering (IC2E), 2019, pp. 23–33. URL: https://arXiv.org/abs/1904.01576.

[39] Eric A. Brewer. “Kubernetes and the path to cloud native”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC ’15*. New York, New York, USA: Association for Computing Machinery (ACM), 2015, pp. 167–167. DOI: 10.1145/2806777.2809955.

[40] Rene Brun and Fons Rademakers. “ROOT — An object oriented data analysis framework”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. New Computing Techniques in Physics Research V 389.1 (Apr. 1997), pp. 81–86. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(97)00048-X.

[41] Miguel Caballer et al. “Dynamic Management of Virtual Infrastructures”. In: *Journal of Grid Computing* 13.1 (Mar. 2015), pp. 53–70. ISSN: 1570-7873, 1572-9184. DOI: 10.1007/s10723-014-9296-5.

[42] Miguel Caballer et al. “EC3: Elastic Cloud Computing Cluster”. In: *Journal of Computer and System Sciences* 79.8 (Dec. 2013), pp. 1341–1351. ISSN: 00220000. DOI: 10.1016/j.jcss.2013.06.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022000013001141.

[43] Amanda Calatrava et al. “Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures”. In: *Future Generation Computer Systems* 61 (Aug. 2016), pp. 13–25. ISSN: 0167739X. DOI: 10.1016/j.future.2016.01.018.

[44] Andrés Camero and Enrique Alba. “Smart City and information technology: A review”. In: *Cities* 93 (Oct. 2019). Publisher: Elsevier Ltd, pp. 84–94. ISSN: 02642751. DOI: 10.1016/j.cities.2019.04.014.

[45] Canonical. *cloud-init: The standard for customising cloud instances*. URL: https://cloud-init.io/.

[46] Joao Carreira et al. “A case for serverless machine learning”. In: *Workshop on Systems for ML and Open Source Software at NeurIPS*. Backup Publisher: Workshop on Systems for ML and Open Source Software at NeurIPS. Semantic Scholar, 2018.

[47] Alex Casalboni. *AWS Lambda Power Tuning*. URL: https://github.com/alexcasalboni/aws-lambda-power-tuning.

[48] Ryan Chard et al. “funcX: A Federated Function Serving Fabric for Science”. In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA: ACM, 2020, pp. 65–76. ISBN: 978-1-4503-7052-3. DOI: 10.1145/3369583.3392683.

[49] Ryan Chard et al. “Serverless Supercomputing: High Performance Function as a Service for Science”. In: *CoRR* abs/1908.04907 (2019). URL: http://arxiv.org/abs/1908.04907.

[50] Ching Hui Chen et al. “Camera Networks for Healthcare, Teleimmersion, and Surveillance”. In: *Computer* 47.5 (May 2014). Publisher: IEEE Computer Society, pp. 26–36. ISSN: 0018-9162. DOI: 10.1109/MC.2014.112.

[51] Qi Chen et al. “A Survey on an Emerging Area: Deep Learning for Smart City Data”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 3.5 (Oct. 2019). Publisher: Institute of Electrical and Electronics Engineers Inc., pp. 392–410. ISSN: 2471-285X. DOI: 10.1109/TETCI.2019.2907718.

[52] Victor Chernozhukov et al. “Double/Debiased Machine Learning for Treatment and Causal Parameters”. In: *arXiv* (July 2016). URL: http://arXiv.org/abs/1608.00060.

[53] Angelos Christidis, Roy Davies, and Sotiris Moschoyiannis. “Serving machine learning workloads in resource constrained environments: A serverless deployment example”. In: *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019*. Institute of Electrical and Electronics Engineers Inc., Nov. 2019, pp. 55–63. ISBN: 978-1-72815-411-4. DOI: 10.1109/SOCA.2019.00016.

[54] Angelos Christidis et al. “Enabling Serverless Deployment of Large-Scale AI Workloads”. In: *IEEE Access* 8 (2020). Publisher: Institute of Electrical

and Electronics Engineers Inc., pp. 70150–70161. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2985282.

[55] Cloud Native Computing Foundation. *K3s*. URL: https://k3s.io/.

[56] CNCF. *Kueue - Kubernetes-native Job Queueing*. URL: https://kueue.sigs.k8s.io/.

[57] CNCF. *Serverless Workflow: A specification for defining declarative workflow models that orchestrate Event-driven, Serverless applications*. URL: https://serverlessworkflow.io.

[58] David Corral-Plaza, Juan Boubeta-Puig, and Manuel Resinas. "Un Recorrido por los Principales Proveedores de Servicios de Machine Learning y Predicción en la Nube". In: *Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2018)*. Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2018), 2018, pp. 1–10. URL: http://hdl.handle.net/11705/JCIS/2018/013.

[59] Raphaël Couturier. *Designing scientific applications on GPUs*. Publication Title: Designing Scientific Applications on GPUs. Chapman and Hall/CRC, 2013. 1-473. ISBN: 978-1-4665-7164-8. DOI: 10.1201/b16051.

[60] dCache. *Distributed storage for scientific data*. URL: https://dcache.org/.

[61] Ewa Deelman et al. "Pegasus: a Workflow Management System for Science Automation". In: *Future Generation Computer Systems* 46 (2015), pp. 17–35. DOI: 10.1016/j.future.2014.10.008.

[62] *DEEP Open Catalog*. URL: https://marketplace.deep-hybrid-datacloud.eu/.

[63] *DEEP Open Catalog - Model "Body Pose Detection"*. URL: https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-posenet-tf.html.

[64] *DEEP Open Catalog - Model "Plants species classifier"*. URL: https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-plants-classification-tf.html.

[65] *DEEP Open Catalog - Model "Train an audio classifier"*. URL: https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-audio-classification-tf.html.

[66] DEEP-Hybrid DataCloud. *Deep learning and machine learning for the EOSC*. URL: https://deep-hybrid-datacloud.eu/.

[67] *DEEPaaS: A REST API to serve machine learning and deep learning models*. URL: https://github.com/indigo-dc/DEEPaaS.

[68] Docker. *Docker Hub*. URL: https://hub.docker.com/.

[69] Docker. *Enterprise Container Platform*. URL: https://www.docker.com/.

[70] Dropbox. *Dropbox.com*. URL: https://www.dropbox.com/.

[71] Łukasz Dutka et al. "Onedata - A step forward towards globalization of data access for computing Infrastructures". In: *Procedia Computer Science*. Vol. 51. ISSN: 18770509. 2015, pp. 2843–2847. DOI: 10.1016/j.procs.2015.05.445.

[72] *EGI DataHub*. URL: https://www.egi.eu/services/datahub/.

[73] Simon Eismann et al. "A Review of Serverless Use Cases and their Characteristics". In: *arXiv* (Aug. 2020). Publisher: arXiv. URL: http://arXiv.org/abs/2008.11110.

[74] Nicolas Ferry, Rustem Dautov, and Hui Song. "Towards a Model-Based Serverless Platform for the Cloud-Edge-IoT Continuum". In: *Proceedings - 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022* (2022), pp. 851–858. DOI: 10.1109/CCGRID54584.2022.00101.

[75] FFmpeg. *FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video*. URL: https://www.ffmpeg.org/.

[76] Fission. *Open source Kubernetes-native Serverless Framework*. URL: https://fission.io/.

[77] Sadjad Fouladi et al. “From laptop to Lambda: Outsourcing everyday jobs to thousands of transient functional containers”. In: *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019*. 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: https://dl.acm.org/doi/10.5555/3358807.3358848.

[78] Geoffrey Charles Fox et al. “Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research”. In: *arXiv* abs/1708.08028 (2017). URL: https://arXiv.org/abs/1708.08028.

[79] Holger Gantikow, Steffen Walter, and Christoph Reich. “Rootless Containers with Podman for HPC”. In: *High Performance Computing*. Ed. by Heike Jagode et al. Backup Publisher: High Performance Computing. Cham: Springer International Publishing, 2020, pp. 343–354. ISBN: 978-3-030-59851-8.

[80] Gartner. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $500 Billion in 2022*. URL: https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022.

[81] Yolanda Gil et al. “Wings: Intelligent Workflow-Based Design of Computational Experiments”. In: *IEEE Intelligent Systems* 26.1 (2011). URL: http://www.isi.edu/~gil/papers/gil-etal-ieee-is-11.pdf.

[82] V. Giménez-Alventosa, Germán Moltó, and Miguel Caballer. “A framework and a performance assessment for serverless MapReduce on AWS Lambda”. In: *Future Generation Computer Systems* (Mar. 2019). ISSN: 0167739X. DOI: 10.1016/j.future.2019.02.057.

[83] V. Giménez-Alventosa, Germán Moltó, and J. Damián Segrelles. “RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments”. In: *International Symposium on Cloud Computing and Services for High Performance Computing Systems (as part of the 18th International Conference on High Performance Computing & Simulation (HPCS 2020)*. 2020.

[84] Jorge Gomes et al. “Enabling rootless Linux Containers in multi-user environments: The udocker tool”. In: *Computer Physics Communications*

232 (Nov. 2018). Publisher: Elsevier B.V., pp. 84–97. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.05.021.

[85] Google. *Audioset*. URL: https://research.google.com/audioset/.

[86] Google Cloud. *Cloud Computing Services*. URL: https://cloud.google.com/.

[87] Google Cloud. *Kubernetes - Google Kubernetes Engine (GKE)*. URL: https://cloud.google.com/kubernetes-engine.

[88] GoogleContainerTools. *Build Container Images In Kubernetes.* URL: https://github.com/GoogleContainerTools/kaniko.

[89] Gorka Madariaga. *Building serverless microservices in Azure - sample architecture*. URL: https://azure.microsoft.com/en-us/blog/building-serverless-microservices-in-azure-sample-architecture/.

[90] Grace O'Halloran. *An Introduction to Serverless Computing and Function-as-a-Service*. URL: https://www.advancinganalytics.co.uk/blog/2021/9/2/an-introduction-to-serverless-computing-and-function-as-a-service.

[91] GRyCAP. *CLUES - cluster energy saving (for hpc and cloud computing)*. URL: https://www.grycap.upv.es/clues/es/index.ph.

[92] GRyCAP. *FaaS Supervisor - Input / Output Data Manager for FaaS Infrastructures*. URL: https://github.com/grycap/faas-supervisor.

[93] GRyCAP. *Integration with the EGI Federated Cloud - OSCAR documentation*. URL: https://docs.oscar.grycap.net/egi-integration/.

[94] GRyCAP. *minicon: minimization containers*. URL: https://github.com/grycap/minicon.

[95] GRyCAP. *OSCAR: Open Source Serverless Computing for Data-Processing Applications*. URL: https://github.com/grycap/oscar.

[96] GRyCAP. *scar-deepaas-ui*. URL: https://github.com/grycap/scar-deepaas-ui.

[97] GRyCAP. *SCAR: Serverless Container-aware ARchitectures (e.g. Docker in AWS Lambda)*. URL: https://github.com/grycap/scar.

[98] Quentin Hardy. *Cloud Computing Giants Add to Open Source Credentials With Kubernetes*. en. URL: https://archive.nytimes.com/bits.blogs.nytimes.com/2014/07/10/cloud-computing-giants-add-to-open-source-credentials-with-kubernetes/.

[99] Michael T. Heath. *Scientific Computing: : An Introductory Survey, Revised Second Edition*. Philadelphia, PA: Society for Industrial and Applied Mathematics, Nov. 2018. 1-567. ISBN: 978-1-61197-557-4. DOI: 10.1137/1.9781611975581.

[100] Scott Hendrickson et al. "Serverless Computation with openLambda". In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. Backup Publisher: 8th USENIX Conference on Hot Topics in Cloud Computing. USENIX Association, 2016, pp. 33–39. ISBN: 00384941. DOI: 10.1111/j.1540-6237.2011.00758.x.

[101] V. Ishakian, V. Muthusamy, and A. Slominski. "Serving Deep Learning Models in a Serverless Platform". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Apr. 2018, pp. 257–262. DOI: 10.1109/IC2E.2018.00052.

[102] Peter Ivie and Douglas Thain. *Reproducibility in scientific computing*. ISSN: 15577341 Issue: 3 Publication Title: ACM Computing Surveys Volume: 51. Apr. 2018. DOI: 10.1145/3186266.

[103] Douglas M Jacobsen and Richard Shane Canon. *Contain This, Unleashing Docker for HPC*. United States: NERSC, 2015. URL: https://www.nersc.gov/assets/Uploads/cug2015udi.pdf.

[104] Matthijs Jansen et al. "The SPEC-RG Reference Architecture for the Edge Continuum". In: (July 2022). URL: http://arxiv.org/abs/2207.04159.

[105] Jeff Barr. *AWS Lambda – Run Code in the Cloud*. Section: Amazon DynamoDB. Nov. 2014. URL: https://aws.amazon.com/blogs/aws/run-code-cloud/.

[106] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. "Serverless execution of scientific workflows". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10601 LNCS. ISSN: 16113349. Springer Verlag, 2017, pp. 706–721. ISBN: 978-3-319-69034-6. DOI: 10.1007/978-3-319-69035-3_51.

[107] Anshul Jindal et al. "Function delivery network: Extending serverless computing for heterogeneous platforms". In: *Software: Practice and Experience* (Mar. 2021). Publisher: John Wiley and Sons Ltd, spe.2966. ISSN: 0038-0644. DOI: 10.1002/spe.2966.

[108] Eric Jonas et al. "Occupy the cloud: distributed computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing - SoCC '17*. Backup Publisher: 2017 Symposium on Cloud Computing - SoCC '17. New York, New York, USA: ACM Press, 2017, pp. 445–451. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128601.

[109] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. "Centralized core-granular scheduling for serverless functions". In: *Proceedings of the ACM symposium on cloud computing*. 2019, pp. 158–164.

[110] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. "Towards serverless as commodity: A case of Knative". In: *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*. Backup Publisher: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019. Association for Computing Machinery, Inc, Dec. 2019, pp. 13–18. ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368135.

[111] Shwet Ketu and Pramod Kumar Mishra. "Cloud, fog and mist computing in IoT: an indication of emerging opportunities". In: *IETE Technical Review* 39.3 (2022), pp. 713–724.

[112] Knative. *Kubernetes-based platform to deploy and manage modern serverless workloads*. URL: https://knative.dev/.

[113] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM*

60.6 (May 2017). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[114] Kubernetes. *Production-Grade Container Orchestration*. URL: https://kubernetes.io/.

[115] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (May 2017). Ed. by Attila Gursoy. Publisher: Public Library of Science, e0177459. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459.

[116] Malte S. Kurz. "Distributed Double Machine Learning with a Serverless Architecture". In: *arXiv* (Jan. 2021). URL: http://arXiv.org/abs/2101.04025.

[117] Katina Kyoreva. "State of the Art JavaScript Application Development with Vue. js". In: *Proceedings of International Conference on Application of Information and Communication Technology and Statistics in Economy and Education (ICAICTSEE)*. International Conference on Application of Information and Communication, 2017, pp. 567–572.

[118] Junfeng Li et al. "Understanding open source serverless platforms: Design considerations and performance". In: *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*. Backup Publisher: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019. Association for Computing Machinery, Inc, Dec. 2019, pp. 37–42. ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368139.

[119] Linux Containers. *LXC*. URL: https://linuxcontainers.org/lxc/introduction/.

[120] Álvaro López García. "DEEPaaS API: a REST API for Machine Learning and Deep Learning models". In: *Journal of Open Source Software* 4.42 (Oct. 25, 2019). Publisher: The Open Journal, p. 1517. ISSN: 2475-9066. DOI: 10.21105/joss.01517.

[121] Nima Mahmoudi and Hamzeh Khazaei. *SimFaaS: A Performance Simulator for Serverless Computing Platforms*. 2020. URL: https://arXiv.org/abs/2102.08904.

[122] Maciej Malawski et al. “Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions”. In: *Future Generation Computer Systems* (2017). Publisher: Elsevier B.V. ISSN: 0167739X. DOI: 10.1016/j.future.2017.10.029.

[123] E McCallister, Tim Grance, and K Kent. “Guide to protecting the confidentiality of personally identifiable information (PII)”. In: *Special Publication 800-122 Guide* (2010). ISBN: 9781437934885, pp. 1–59. DOI: 10.5555/2206206.

[124] Microsoft Azure. *Azure Functions—Develop Faster With Serverless Compute*. URL: https://azure.microsoft.com/en-us/services/functions/.

[125] Microsoft Azure. *Cloud Computing Services*. URL: https://azure.microsoft.com/en-us/.

[126] Microsoft Azure. *Managed Kubernetes Service (AKS)*. URL: https://azure.microsoft.com/en-us/products/kubernetes-service/.

[127] Microsoft Azure. *Total Cost of Ownership (TCO) Calculator*. URL: https://azure.microsoft.com/en-us/pricing/tco/calculator/.

[128] MinIO. *High Performance, Kubernetes Native Object Storage*. URL: https://min.io/.

[129] Asmaa Mirkhan. *BlurryFaces: A tool to blur faces or other regions in photos and videos*. URL: https://github.com/asmaamirkhan/BlurryFaces.

[130] Germán Moltó. *AI-SPRINT D3.1 - First release and evaluation of the runtime environment*. Dec. 2021. DOI: 10.5281/zenodo.6372963.

[131] Max A Moritz. “Wildfires ignite debate on global warming”. In: *Nature* 487.7407 (2012), pp. 273–273.

[132] Kief Morris. *Infrastructure as code: managing servers in the cloud*. Publication Title: O’Reilly Media, Inc. 2016. 362 pp. ISBN: 978-1-4919-2435-8. URL: https://www.oreilly.com/library/view/infrastructure-as-code/9781491924334/.

[133] Carla Mouradian et al. "A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 416–464. DOI: 10.1109/COMST.2017.2771153.

[134] Diana M. Naranjo et al. "A serverless gateway for event-driven machine learning inference in multiple clouds". In: *Concurrency and Computation: Practice and Experience* 35.18 (2021), e6728. ISSN: 1532-0634. DOI: 10.1002/cpe.6728.

[135] Diana M. Naranjo et al. "Accelerated serverless computing based on GPU virtualization". In: *Journal of Parallel and Distributed Computing* 139 (2020), pp. 32–42. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.01.004.

[136] Diana María Naranjo Delgado. "Serverless Computing Strategies on Cloud Platforms". PhD thesis. Valencia, Spain: Universitat Politècnica de València, 2021. URL: https://riunet.upv.es/handle/10251/160916.

[137] Naranjo Diana M., Risco Sebastián, Moltó Germán, Blanquer Ignacio. "A Serverless Gateway for the Execution of Open Machine Learning Models on AWS". In: Backup Publisher: Gateways 2020. OSF, 2020. URL: https://osf.io/g4dwy/.

[138] Netflix. *Watch TV Shows Online, Watch Movies Online*. URL: https://www.netflix.com.

[139] Nuclio. *Nuclio: Serverless Platform for Automated Data Science*. URL: https://nuclio.io/.

[140] NVIDIA. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone.

[141] NVIDIA. *NVIDIA container runtime*. URL: https://github.com/NVIDIA/nvidia-container-runtime.

[142] OASIS. *TOSCA Simple Profile in YAML version 1.3*. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html.

[143] Justice Opara-Martins, Reza Sahandi, and Feng Tian. "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective". In: *Journal of Cloud Computing* 5.1 (Dec. 2016). Publisher: Springer Verlag, p. 4. ISSN: 2192113X. DOI: 10.1186/s13677-016-0054-z.

[144] OpenFaaS. *Serverless Functions Made Simple*. URL: https://www.openfaas.com/.

[145] A OpenFog Consortium Architecture Working Group et al. "OpenFog reference architecture for fog computing". In: *OPFRA001* 20817 (2017), p. 162.

[146] OpenStack. *Open Source Cloud Computing Infrastructure*. URL: https://www.openstack.org.

[147] Oracle. *Fn Project*. URL: https://fnproject.io/.

[148] Vincenzo Eduardo Padulano et al. "Leveraging an open source serverless framework for high energy physics computing". In: *The Journal of Supercomputing* 79.8 (May 2023), pp. 8940–8965. ISSN: 1573-0484. DOI: 10.1007/s11227-022-05016-y.

[149] A. Palade, A. Kazmi, and S. Clarke. "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge". In: *2019 IEEE World Congress on Services (SERVICES)*. Vol. 2642-939X. ISSN: 2378-3818. 2019 IEEE World Congress on Services (SERVICES), July 2019, pp. 206–211. DOI: 10.1109/SERVICES.2019.00057.

[150] Milan Pavlovic, Yoav Etsion, and Alex Ramirez. "On the memory system requirements of future scientific applications: Four case-studies". In: *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*. 2011, pp. 159–170. ISBN: 978-1-4577-2064-2. DOI: 10.1109/IISWC.2011.6114176.

[151] Alfonso Pérez et al. "A programming model and middleware for high throughput serverless computing applications". In: *Proceedings of the ACM Symposium on Applied Computing*. Vol. Part F1477. Association for Computing Machinery, 2019, pp. 106–113. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297292.

[152] Alfonso Pérez et al. “On-Premises Serverless Computing for Event-Driven Data Processing Applications”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). ISSN: 2159-6182. July 2019, pp. 414–421. DOI: 10.1109/CLOUD.2019.00073.

[153] Alfonso Pérez et al. “Serverless computing for container-based architectures”. In: *Future Generation Computer Systems* 83 (June 2018), pp. 50–59. ISSN: 0167739X. DOI: 10.1016/j.future.2018.01.022.

[154] Danilo Piparo et al. “RDataFrame: Easy Parallel ROOT Analysis at 100 Threads”. In: *EPJ Web of Conferences* 214 (2019). Publisher: EDP Sciences, p. 06029. ISSN: 2100-014X. DOI: 10.1051/epjconf/201921406029.

[155] Reid Priedhorsky and Tim Randles. “Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’17. Backup Publisher: International Conference for High Performance Computing, Networking, Storage and Analysis event-place: Denver, Colorado. New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126925.

[156] Aditya Purohit. *face-mask-detector: Real-Time Face mask detection using deep learning with Alert system*. URL: https://github.com/adityap27/face-mask-detector/.

[157] Mohan Rao Divate Kodandarama, Mohammed Danish Shaikh, and Shreeshrita Patnaik. *SerFer: Serverless Inference of Machine Learning Models*. Madison: University of Wisconsin, 2019. URL: https://divatekodand.github.io/files/serfer.pdf.

[158] Thomas Rausch et al. “Towards a serverless platform for edge AI”. In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[159] Carlos Reano and Federico Silla. “A Performance Comparison of CUDA Remote GPU Virtualization Frameworks”. In: *2015 IEEE International Conference on Cluster Computing*. IEEE, Sept. 2015, pp. 488–489. ISBN: 978-1-4673-6598-7. DOI: 10.1109/CLUSTER.2015.76.

[160] Carlos Reaño et al. “Local and Remote GPUs Perform Similar with EDR 100G InfiniBand”. In: *Proceedings of the Industrial Track of the 16th International Middleware Conference on ZZZ - Middleware Industry ’15*. New York, New York, USA: ACM Press, 2015, pp. 1–7. ISBN: 978-1-4503-3727-4. DOI: 10.1145/2830013.2830015.

[161] Red Hat. *Ansible is Simple IT Automation*. URL: https://www.ansible.com.

[162] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. URL: https://pjreddie.com/darknet/.

[163] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv:1804.02767 [cs]* (Apr. 8, 2018). URL: http://arxiv.org/abs/1804.02767.

[164] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *arXiv:1506.02640 [cs]* (May 9, 2016). URL: http://arxiv.org/abs/1506.02640.

[165] Martin Reisslein, Bernhard Rinner, and Amit Roy-Chowdhury. “Smart Camera Networks [Guest editors’ introduction]”. In: *Computer* 47.5 (May 2014). Publisher: IEEE Computer Society, pp. 23–25. ISSN: 0018-9162. DOI: 10.1109/MC.2014.134.

[166] Sebastián Risco. *Plataforma Serverless Híbrida de Procesado de Datos*. URL: https://riunet.upv.es/handle/10251/125892.

[167] Sebastián Risco and Germán Moltó. “GPU-Enabled Serverless Workflows for Efficient Multimedia Processing”. In: *Applied Sciences* 11.4 (Feb. 2021), p. 1438. ISSN: 2076-3417. DOI: 10.3390/app11041438.

[168] Sebastián Risco et al. “Serverless Workflows for Containerised Applications in the Cloud Continuum”. In: *Journal of Grid Computing* (2021). ISSN: 1572-9184. DOI: 10.1007/s10723-021-09570-2.

[169] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. “AFCL: An Abstract Function Choreography Language for serverless workflow specification”. In: *Future Generation Computer Systems* 114 (Jan. 2021). Publisher:

Elsevier B.V., pp. 368–382. ISSN: 0167739X. DOI: 10.1016/j.future.2020.08.012.

[170] RunasSudo. *audio2srt*. URL: https://gitlab.com/RunasSudo/audio2srt.

[171] L. Sarzyniec et al. "Design and Evaluation of a Virtual Experimental Environment for Distributed Systems". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013, pp. 172–179. DOI: 10.1109/PDP.2013.32.

[172] Swarvanu Sengupta. *faas-flow: Function Composition for OpenFaaS*. URL: https://github.com/s8sg/faas-flow.

[173] Amazon Web Services. *API Gateway*. URL: https://aws.amazon.com/api-gateway.

[174] Ricky J. Sethi et al. "Large-Scale Multimedia Content Analysis Using Scientific Workflows". In: *Proceedings of the 21st ACM International Conference on Multimedia*. MM '13. event-place: Barcelona, Spain. New York, NY, USA: Association for Computing Machinery, 2013, pp. 813–822. ISBN: 978-1-4503-2404-5. DOI: 10.1145/2502081.2502082.

[175] Mohit Sewak and Sachchidan Singh. "Winning in the Era of Serverless Computing and Function as a Service". In: *2018 3rd International Conference for Convergence in Technology, I2CT 2018*. Institute of Electrical and Electronics Engineers Inc., Nov. 2018. ISBN: 978-1-5386-4273-3. DOI: 10.1109/I2CT.2018.8529465.

[176] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. "Serverless Computing: A Survey of Opportunities, Challenges and Applications". In: *arXiv* (Nov. 2019). URL: http://arXiv.org/abs/1911.01296.

[177] Vaishaal Shankar et al. "Numpywren: Serverless Linear Algebra". In: *arXiv* (Oct. 2018). URL: https://arXiv.org/abs/1810.09679.

[178] Matthew Shields. "Control-versus data-driven workflows". In: *Workflows for e-Science*. Springer London, 2007, pp. 167–173. URL: https://link.springer.com/chapter/10.1007/978-1-84628-757-2_11.

[179] Nickolay Shmyrev. *CMUSphinx Open Source Speech Recognition*. URL: http://cmusphinx.github.io/.

[180] Christian Sicari et al. "OpenWolf: A Serverless Workflow Engine for Native Cloud-Edge Continuum". In: *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2022, pp. 1–8.

[181] Tyler J. Skluzacek et al. "Serverless Workflows for Indexing Large Scientific Data". In: *WOSC '19: Proceedings of the 5th International Workshop on Serverless Computing*. Association for Computing Machinery (ACM), 2019, pp. 43–48. DOI: 10.1145/3366623.3368140.

[182] Fedor Smirnov et al. "Apollo: towards an efficient distributed orchestration of serverless function compositions in the cloud-edge continuum". In: *dl.acm.org* (Dec. 2021). DOI: 10.1145/3468737.3494103. URL: https://dl.acm.org/doi/abs/10.1145/3468737.3494103.

[183] Tito Spadini, Dimitri Leandro de Oliveira Silva, and Ricardo Suyama. "Sound Event Recognition in a Smart City Surveillance Context". In: (Oct. 2019). URL: http://arxiv.org/abs/1910.12369.

[184] Josef Spillner, Cristian Mateos, and David A. Monge. "Faaster, better, cheaper: the prospect of serverless scientific computing and HPC". In: *Communications in Computer and Information Science*. Vol. 796. ISSN: 18650929. Springer, Cham, 2018, pp. 154–168. ISBN: 978-3-319-73352-4. DOI: 10.1007/978-3-319-73353-1_11.

[185] Spotify. *Listening is everything*. URL: https://www.spotify.com.

[186] AI-SPRINT. *AI in Secure Privacy-Preserving Computing Continuum*. URL: https://www.ai-sprint-project.eu/.

[187] Statista. *Public cloud computing market size 2023*. en. URL: https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009/.

[188] Forrest Stroud. *What is A Micro-VM?* Aug. 2015. URL: https://www.webopedia.com/definitions/micro-vm/.

[189] The Linux Foundation. *Cloud Native Computing Foundation.* URL: https://www.cncf.io/.

[190] W. Thompson, N. Bhowmik, and T.P. Breckon. "Efficient and Compact Convolutional Neural Network Architectures for Non-temporal Real-time Fire Detection". In: *Proc. Int. Conf. Machine Learning Applications.* IEEE, Dec. 2020, pp. 136–141. DOI: 10.1109/ICMLA51294.2020.00030. URL: http://breckon.org/toby/publications/papers/thompson20fire.pdf.

[191] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. "High-performance cloud computing: A view of scientific applications". In: *I-SPAN 2009 - The 10th International Symposium on Pervasive Systems, Algorithms, and Networks.* 2009, pp. 4–16. ISBN: 978-0-7695-3908-9. DOI: 10.1109/I-SPAN.2009.150.

[192] Matthew Viljoen et al. "Towards European Open Science Commons: The EGI Open Data Platform and the EGI DataHub". In: *Procedia Computer Science.* Vol. 97. ISSN: 18770509. Elsevier B.V., 2016, pp. 148–152. DOI: 10.1016/j.procs.2016.08.294.

[193] Darrell M West. *The future of work: Robots, AI, and automation.* Brookings Institution Press, 2018.

[194] Xiaolong Xu et al. "Blockchain-based cloudlet management for multimedia workflow in mobile cloud computing". In: *Multimedia Tools and Applications* 79.15 (Apr. 2020). Publisher: Springer, pp. 9819–9844. ISSN: 15737721. DOI: 10.1007/s11042-019-07900-x.

[195] *YAML.* URL: https://yaml.org/.

[196] Hong Zhang et al. "Caerus: NIMBLE Task Scheduling for Serverless Analytics". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21).* 2021, pp. 653–669.

[197] Miao Zhang et al. "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. New York, New York, USA: ACM Press, 2019, pp. 61–66. ISBN: 978-1-4503-6298-6. DOI: 10.1145/3304112.3325608.