Advanced Elastic Platforms for High Throughput Computing on Container-based and Serverless Infrastructures

March 2020

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science.

Author: Alfonso Pérez González
Advisor: Dr. Germán Moltó Martínez

Acknowledgments

En primer lugar me gustaría agradecer a mis padres y a mi hermana todo el apoyo que me han dado a lo largo de todos estos años. Siempre habéis estado ahí cuando lo he necesitado y sin vosotros no habría llegado hasta aquí.

También me gustaría agradecer a mi mujer por los ánimos que me ha dado a lo largo de estos años de tesis y por escuchar mis interminables charlas sobre cuál es la tecnología del momento.

Por último quiero agradecer a mis compañeros de trabajo por no perder la paciencia con mis innumerables preguntas y por ayudarme siempre que tenía algún problema. En particular quiero dar las gracias a mi director Germán Moltó por sus acertados comentarios, sugerencias e ideas durante la realización de esta tesis y por aguantar estoicamente mis entregas de última hora.

Muchas gracias a todos.

Abstract

The main objective of this thesis is to allow scientific users to deploy and execute highly-parallel event-driven file-processing serverless applications both in public (e.g. AWS), and in private (e.g. OpenNebula, OpenStack) cloud infrastructures. To achieve this objective, different tools and platforms are developed and integrated to provide scientific users with a way for deploying High Throughput Computing applications based on containers that can benefit from the high elasticity capabilities of the serverless environments. First, an open-source tool to deploy generic serverless workloads in the AWS public Cloud provider has been created. This tool allows the scientific users to benefit from the features of AWS Lambda (e.g. high scalability, event-driven computing) for the deployment and integration of compute-intensive applications that use the Functions as a Service (FaaS) model. Second, an event-driven file-processing high-throughput programming model has been developed to allow the users deploy generic applications as workflows of functions in serverless architectures, offering transparent data management. Third, in order to overcome the drawbacks of public serverless services such as limited execution time or computing capabilities, an open-source platform to support FaaS for compute-intensive applications in on-premises Clouds was created. The platform can be automatically deployed on multi-Clouds in order to create highly-parallel event-driven file-processing serverless applications. Finally, in order to assess and validate all the developed tools and platforms, several use cases with business and scientific backgrounds have been tested.

Resums

El principal objectiu d'aquesta tesi és oferir als usuaris científics una manera de crear i executar aplicacions sense servidor (i.e. *serverless*) altament paral·leles, dirigides per esdeveniments i orientades al processament de dades, tant en proveïdors en núvol públics (e.g. AWS) com en privats (e.g. OpenNebula, OpenStack). Per a dur a terme aquest objectiu, s'ha desenvolupat e integrat diferents eines que ofereixen una via per desplegar aplicacions de computació d'altes prestacions basades en contenidors, alhora que es poden beneficiar de l'alta escalabilitat present en els entorns *serverless*. Primerament, s'ha creat una eina que possibilita el desplegament de càrregues de treball genèriques al proveïdor públic en núvol AWS. Aquesta eina permet aprofitar les funcionalitats de AWS Lambda (e.g. alta escalabilitat, computació basada en esdeveniments) per al desplegament i la integració d'aplicacions computacionalment intensives que fan ús del model de funcions com a servei (*FaaS*). En segon lloc, s'ha desenvolupat un model de programació d'alt rendiment per al processament de dades i orientat a esdeveniments, que permet als usuaris desplegar fluxos de treball com un conjunt de funcions *serverless*, alhora que ofereix una gestió transparent de les dades. En tercer lloc, per a superar els problemes presents als proveïdors públics (e.g. temps d'execució limitat) s'ha creat una plataforma que permet utilitzar el model *FaaS* en infraestructures privades. A més, aquesta plataforma pot ser desplegada automàticament en múltiples proveïdors públics en núvol. Finalment, per a comprobar i validar les diferents eines i plataformes dutes a terme, s'han provat diferents casos d'ús amb interès tant per a la recerca com per a l'empresa.

Resumen

El principal objetivo de esta tesis es ofrecer a los usuarios científicos un modo de crear y ejecutar aplicaciones sin servidor (i.e. *serverless*) altamente paralelas, dirigidas por eventos y orientadas al procesado de datos, tanto en proveedores en la nube públicos (e.g. AWS) como privados (e.g. OpenNebula, OpenStack). Para llevar a cabo dicho objetivo, se han desarrollado e integrado diferentes herramientas que ofrecen una vía para desplegar aplicaciones de computación de altas prestaciones basadas en contenedores, que además pueden beneficiarse de la alta escalabilidad presente en los entornos *serverless*. Primero se ha creado una herramienta que permite el despliegue de cargas de trabajo genéricas en el proveedor público AWS. Esta herramienta posibilita que se puedan aprovechar las funcionalidades de AWS Lambda (e.g. alta escalabilidad, computación basada en eventos) para el despliegue y la integración de aplicaciones computacionalmente intensivas que usan el modelo de funciones como servicio (*FaaS*). En segundo lugar se ha desarrollado un modelo de programación de alto rendimiento para el procesado de datos y orientado a eventos que permite a los usuarios desplegar flujos de trabajo como un conjunto de funciones *serverless*, a la vez que ofrece una gestión transparente de los datos. En tercer lugar, para poder superar los problemas presentes en los proveedores públicos (e.g. tiempo de ejecución limitado), se ha creado una plataforma que facilita el uso del modelo *FaaS* en infraestructuras privadas. Esta plataforma también puede ser desplegada automáticamente en distintos proveedores públicos de la nube. Finalmente, para comprobar y validar las diferentes herramientas y plataformas desarrolladas, se han probado diferentes casos de uso con interés tanto para investigación como para la empresa.

Contents

Contents xi

1 Introduction 1

1.1 Motivation 1

1.2 Objectives. 5

1.3 Thesis Structure 6

2 Background 7

2.1 Cloud Computing 7

2.2 Containers and Container Orchestrators. 9

2.3 Serverless Computing 11

2.4 Serverless offerings 14

2.5 State of the Art 28

3 Serverless Container-aware Architectures 33

3.1 Generic Architecture. 34

3.2 Framework implementation 36

3.3 Architecture of SCAR. 38

3.4 SCAR usage . . . 41
3.5 On the Lambda function's ephemeral cache . . . 44
3.6 Study of the AWS Lambda Freeze/Thaw behavior . . . 47
3.7 Conclusions . . . 53

4 Event-Driven File-Processing Serverless Programming Model 55
4.1 Highly-scalable HTTP endpoints with API Gateway . . . 58
4.2 S3 file upload/read triggers Lambda Function . . . 59
4.3 Data management inside the Lambda Function . . . 61
4.4 Output files trigger new Lambda functions . . . 64
4.5 Job Processing with AWS Batch . . . 65
4.6 Cost analysis . . . 67
4.7 Conclusions . . . 72

5 Open-source Serverless Computing for Data-Processing Applications 75
5.1 Platform Components . . . 76
5.2 OSCAR architecture . . . 80
5.3 Case study: Video Processing Service in On-premises Infrastructure . . . 86
5.4 Conclusions . . . 93

6 Use cases 95
6.1 Adding support to programming languages and software in AWS Lambda . . . 96
6.2 Massive image processing service . . . 99
6.3 Video Processing Service in AWS . . . 104
6.4 Plant classification . . . 108
6.5 Multi-cloud workflow for video processing . . . 109
6.6 Air pollution information service . . . 112
6.7 Monetizing Private Algorithm Workflow Executions . . . 114
6.8 GROMACS in AWS Batch . . . 117
6.9 Scientific diffusion . . . 118

7 Conclusions 121

7.1 Summary and Contributions 121

7.2 Future work 123

Bibliography 125

Index 145

A SCAR client commands 145

B OSCAR Template 153

Acronyms 159

Chapter 1

Introduction

Section 1.1 gives the reader the motivation that started the presented thesis, then section 1.2 presents the main objectives of the thesis, and finally section 1.3 summarises the thesis structure.

1.1 Motivation

Over the last years the offering made by large enterprises of renting computing, storage and network capacity on a pay-per-use basis has resulted in a tremendous revolution that democratized the access to large-scale enterprise-ready computing infrastructures without large upfront investments. Indeed, Figure 1.1 shows the results of a survey carried out in 2019 among 786 technical professionals about their adoption of cloud computing. These professionals ranged from technical executives to managers and practitioners across a broad cross-section of organizations, where 58% of the participants where from companies with more than 1,000 employees. The results indicate that 94% of the companies were using the Cloud [64].

Furthermore, cloud computing has introduced the ability to provide a wide variety of well-known service models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Elasticity has been the cornerstone functionality of these services, allowing to provision more resources in order to cope with increased workloads or freeing resources in order to decrease the cost of such services. To be able to provide these elasticity capabilities, VMs have played a fundamental role allowing to

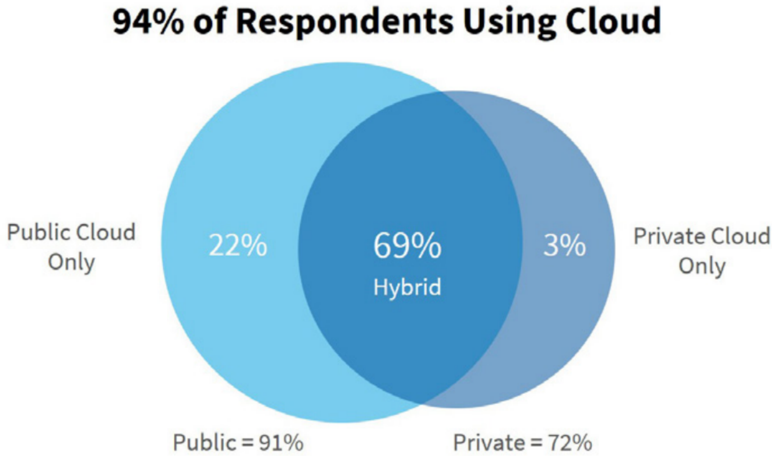

Figure 1.1: Cloud usage in 2019 among 786 different professionals.

create customized and replicable execution environments, in order to guarantee successful executions.

The main public cloud computing providers to date, Amazon Web Services [10], Microsoft Azure [134], and Google Cloud Platform [79], have fostered the migration of complex application architectures to the public Cloud in order to take advantage of the pay-per-use cost model. On the other hand, Cloud Management Platforms such as OpenStack [148] and OpenNebula [145] have enabled system administrators to create cloud infrastructures on their own hardware (on-premises).

In parallel, the mainstream adoption of Linux containers, propelled by the popularity of Docker [52], enabled users to maintain customized execution environments, in the shape of lightweight Docker images instead of bulky Virtual Machine Images. This paved the way for the microservices architectural pattern to rise, in order to decouple complex applications into several small, independently deployed services that interact via REST interfaces [55].

Creating distributed applications based on microservices required the ability to manage a fleet of containers at scale, thus fostering the appearance of container orchestrators such as Swarm mode in Docker Engine [54], Nomad

[92] or Kubernetes [114], and managed services provided by the leading public cloud providers. Examples of the latter are Amazon ECS [7], Amazon EKS [8], Azure Container Service [126], and Google Kubernetes Engine [83]. The main drawback with these services is that they are typically oriented to advanced users, in order to deal with the capacity planning required to deploy the clusters in advance and optimize the allocation of resources to containers.

Thus, to cope with such issues (e.g. scheduling of resources, management of containers, etc), serverless computing emerged. Serverless computing is based on creating application architectures that entirely rely on cloud services that provide automated resource provisioning on behalf of (and transparent to) the user. Thus, by not explicitly managing servers, operating systems, runtimes or applications, developers can focus on the definition of the code logic and the streams of data instead of devoting time to infrastructure provision, configuration and scalability.

The SPEC Cloud Group [62] defines three key features of serverless cloud architectures:

- Granular billing: the user is only charged when the application is running.
- Minimal operational logic: the cloud provider is responsible for resource management and autoscaling.
- Event-driven: short-lived execution of functions in response to events.

Complying with these features, pioneer providers in the serverless area, such as AWS Lambda [17], introduced large-scale parallelism by allowing functions (minimal operational logic) to be invoked in response to events (event-driven) such as uploading a file to a storage service while offering a pay-per-use model that only charges you by the time that the function is running (granular billing).

However, the Functions as a Service (FaaS) model that these services impose, hinders their adoption for the general execution of scientific applications. To adopt the FaaS model, applications need to be redesigned as a set of event-triggered functions coded in a supported programming language by the cloud provider, and after all, many applications cannot be easily redesigned as a set of functions. In fact, the interface between the user and the serverless platform should not only be based on functions. Instead, serverless providers should provide the means to allow the users execute generic application environments in their infrastructures. To this end, containers could provide users with the ability to run virtually any kind of application without having

to introduce changes. Thus, supporting applications defined via container images in a serverless platform would allow the users to: i) easily deploy applications which may already be packaged as Docker images; ii) use applications that depend on libraries not available in the runtime environment of the functions; iii) use programming languages not currently supported by the serverless provider.

Nevertheless, running containers on public serverless infrastructures is not the solution for all the restrictions imposed by the public serverless providers. Limitations such as predefined execution time, reduced disk space, or the restriction to process sensible data in public infrastructures revealed the need for on-premises FaaS offerings. However, these on-premises solutions are mainly focused on the execution of short-lived stateless HTTP-based requests (similar to the public offerings), and do not take into account other more generic applications and long-running jobs. Thus, we identified the need to support scalable event-driven computing for generic data-processing applications in on-premises serverless infrastructures. One direct benefit of using on-premises infrastructures is that we could tune them to suit our needs and thus, overcome the strict limitations of the public offerings. However this would also imply having to deal again with the configuration of the infrastructure resources, taking some steps back in the automatization of the offered services in comparison with the public providers. Thus, a platform that transparently manages the required resources, that is able to automatize the application deployment, and in general that facilitates the usage of the FaaS model in on-premises infrastructures could greatly benefit the scientific user community.

As summary, the presented challenges that this thesis seeks to address are: 1) Analyze the viability of executing containers in public serverless infrastructures. Usually, these closed environments have restricted functionality that limits the software that can be executed; 2) Define a methodology to allow scientific users to create serverless applications in public infrastructures, to ease the application definition, data should be automatically managed by the platform with no user interaction needed; 3) Extend the knowledge gathered from tackling the previous challenges to provide on-premises infrastructures with features similar to the ones present in public providers. In particular, the resource management and the configuration should be automated. Aligned with these research challenges, the next section proposes the main objectives of this thesis and presents a summary of the methodology envisaged to achieve such objectives.

1.2 Objectives

The main objective of this thesis is to allow scientific users to deploy and execute highly-parallel event-driven file-processing applications both in public and in private Cloud infrastructures. Moreover, to ease the application migration, the data management inside the serverless infrastructures must be as transparent to the users as possible. To better grasp this objective we divided it in different milestones that will be tackled sequentially during the development of this thesis, described as follows:

- Ease the application migration to the public serverless cloud providers and overcome some of the limitations imposed by such providers. Limitations such as closed execution environments, fixed execution time, restricted disk space or the inability to support accelerated computing hardware, such as GPUs, currently hinder the adoption of these platforms for scientific computing. Allowing the users to easily migrate their applications to these infrastructures would increase the adoption of the FaaS paradigm among new users. To this aim, we propose the development of a tool to allow the users to execute generic applications packages as container images in public serverless providers.
- Offer the users a model to define highly-parallel event-driven file-processing applications. To this aim, we propose a High Throughput Computing (HTC) [97] programming model that supports the creation of such applications. This programming model should simplify and automate the application deployment process and also automatically manage the data movement throughout the application. Furthermore, the proposed programming model should be platform-agnostic so it can be reused in different public and private cloud providers.
- Develop a platform to support on-premises FaaS for general-purpose file-processing computing applications. The goal is to facilitate the adoption of event-driven computation for scientific applications that require processing data files. To this aim, we propose the development of tools to self-deploy an scalable integrated platform that can be accessed by a Graphical User Interface (GUI). Such GUI would allow to define and manage the complete life cycle of functions that will be efficiently triggered upon users uploading files to specified folders. We aim to abstract away the details concerning the management of infrastructure resources, the function scheduling, the function and storage creation, and specially, the deployment of the platform in

different cloud providers. Like in the public offering of serverless services, users should be able to start the function execution by uploading files to a predefined storage space and after the function execution, download the generated output files.

In addition to the previous objectives, another important goal of this thesis is to share the results with the community of users. Therefore, all the tools developed during this thesis will be open source and publicly available in software repositories (e.g. GitHub) to the community.

1.3 Thesis Structure

After the introduction, the remainder of this thesis is structured as follows. First, chapter 2 describes the related work in the scope of the thesis. Then, chapter 3 describes the Serverless Container-aware ARchitectures (SCAR) tool, its architecture, and how the container cache is implemented in the AWS Lambda environment in order to efficiently support running generic application containers in this serverless service. Chapter 4 describes the programming model proposed to allow the users to define event-driven file-processing applications. Next, chapter 5 describes the Open Source Serverless Computing for Data-Processing Applications (OSCAR) platform created to support the event-driven programming model of serverless applications for on-premises cloud infrastructures. Chapter 6 describes different use cases for the developed tools and platforms in order to evaluate their benefits and limitations. Finally, chapter 7 summarizes the main contributions of this thesis and presents future works.

Chapter 2

Background

This chapter analyzes the state of the art related with serverless computing. To this aim, sections 2.1, and 2.2, show the evolution of technologies, work methodologies and programming models until the arrival of serverless technologies. Then, section 2.3 introduces serverless computing, and analyzes the public and open-source alternatives available nowadays to execute Functions as a Service (FaaS).

In the past couple of years, the term 'serverless' has been used to describe multiple diverse services and platforms that not always presented the intrinsic characteristics expected of the serverless paradigm. Therefore, to be able to differentiate the characteristics that define a serverless platform and to get a clear understanding of all the concepts behind the serverless paradigm, first we are going to briefly go through the services and models that laid the foundations and paved the way to the development of serverless technologies.

2.1 Cloud Computing

Almost fifteen years ago the term 'Cloud Computing' started to be used by companies like Amazon and Google to describe the paradigm in which people stored files, accessed computer power and executed software in the web rather than in their computers. However, it did not exist a formal definition of 'Cloud Computing' until 2011, when the National Institute of Standards and Technology (NIST) published a document [119] describing it:

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

In addition to the definition of cloud computing, the NIST document also states that "*the cloud model is composed of five essential characteristics, three service models, and four deployment models*". The essential characteristics intrinsic to all cloud services are: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. The four deployment models that allow us to differentiate who manages the infrastructures are: private cloud, community cloud, public cloud, and hybrid cloud. Finally, to be able to separate between different levels of responsibility in infrastructure management there exist the service models. The NIST originally identifies three service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS), although in the latest years, new service paradigms have appeared, such as Container as a Service (CaaS), Backend as a Service (BaaS), and Functions as a Service (FaaS). To better understand the differences between these service models, a brief description of each one is presented in the following paragraphs.

The most basic business model offered by cloud computing providers consists in renting servers, storage and network so that users can make use of these resources as they require. This service model is called Infrastructure as a Service (IaaS) and some well-known public services in this field are AWS Elastic Compute Cloud (EC2) [14], Azure IaaS [129], and Google Compute Engine (GCE) [82]. On-premises Cloud Management Platforms (CMPs) include OpenStack [148] and OpenNebula [145]. Managing these infrastructures is far from being a trivial task and requires dealing with operating systems, storage, and network configurations. Different open-source tools to ease the configuration of virtual infrastructures in multiple cloud providers can be found in the literature for example the Infrastructure Manager (IM) [38], Cloudify [45] or Occopus [141].

However, even though these tools exist, not all the users have the necessary knowledge or time to create and maintain their virtual infrastructures, and this is why the Platform as a Service (PaaS) model appeared. The PaaS model offers the users a managed environment where their applications can

be deployed without dealing with infrastructure maintenance or resource provisioning. Some of the most known PaaS services are AWS Elastic Beanstalk [15], Google App Engine [77], Heroku [96], or CloudFoundry [44].

The last of the three service models presented in the NIST document is also the most used nowadays by the end user: the Software as a Service (SaaS) model. This model offers an already deployed application and the user only has to configure some simple things (e.g. choose the email address or set some filter rules). Among the most known SaaS providers in this field are Google with G Suite [76], Microsoft with Office 365 [136] or Adobe with Creative Cloud [1]. Despite the fact that the SaaS model is the easiest to use, it is also the most restrictive one regarding the applications used because the user is confined to the application ecosystem offered by the provider.

2.2 Containers and Container Orchestrators

In the last few years and thanks to the improvements made in lightweight virtualization, the container technology has become a strong player in the fields of application building, distribution, maintenance, and deployment. Thanks to the popularization of tools like Docker [52], LXD [41], and OpenVZ [184], container technology has been adopted like an standard for application development, and delivery [122].

In addition, the microservices architecture [162], where an application is not a monolithic piece of code but instead is divided into smaller services, has also experimented a significant growth in users thanks to the container technology. Among the main characteristics of the services provided by microservice architectures are: highly maintainable and testable, loosely coupled, and independently deployable [163]. Thus, microservices architectures allows companies to increase their agility, i.e. the application delivery velocity, that such companies have to meet in order to cope with the market changes and to deliver new products, services, and features. Figure 2.1 shows a typical evolution of a monolithic system to a complete containerized cloud of microservices. The first step usually implies containerizing the complete application. From there, if new features are added or refactorized, those features can be externalized as microservices that can be deployed in the same infrastructure or in the Cloud. If the refactorization continues or new features are added as indicated, the application can eventually become a containerized microservices architecture.

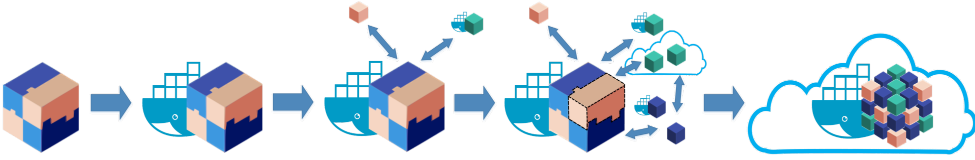

Figure 2.1: Evolution of application architecture. From monolithic on-premises to public containerized microservices[1]

However dealing with a high number of microservices, and thus containers, comes with a cost. The drawbacks of such architectures are, among others, service discovery, auto scaling, load balancing, and in general service orchestration [162]. Thus, container orchestrators appeared to ease the management of such architectures. The main open source container orchestrators available nowadays are:

- Kubernetes [114], an open-source system for orchestrating Docker containers originally developed by Google, that thanks to its reliability, auto-scaling, and load-balancing capabilities, it has become the *de facto* standard for container orchestration on both public and private clouds. Kubernetes has been successfully used to handle numerous use cases and workloads for many different organizations [112] and has gained so much momentum that the most relevant public providers currently offer specific Kubernetes-based services. This is the case of Amazon Elastic Kubernetes Service (EKS) [8], Microsoft Azure Kubernetes Service (AKS) [133] or Google Kubernetes Engine (GKE) [83].

- Docker Swarm [54], developed by Docker and included with the Docker engine distributions, it offers the expected orchestrator services like load balancing, scaling, and service discovery. Docker Swarm is managed by the same client than the docker engine which provides a direct access point to container orchestration for users of the docker client.

- Marathon [175], an open-source framework for container orchestration based on Apache Mesos [176] aimed at executing long-running applications or services. Marathon has a GUI and also offers additional features such as service metrics, event subscriptions and health checks to keep up with the status of the deployed applications.

- Nomad [92], the open-source solution for container orchestration offered by HashiCorp. In addition to Docker containers, Nomad offers support

[1]Based on the image from: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices/microservices-migration.png`

to virtual machines and binary executions, and can also be connected to other HashiCorp services such as Consul [91], for service discovery, and Vault [93], for storing and encrypting sensitive data.

Most of the public service providers offer container orchestration through managed Kubernetes services. However, some providers still offer other container orchestration solutions such as:

- Amazon Elastic Container Service (ECS) [7], a service offered by AWS to manage containers on the Cloud. This service runs the containers in the Amazon EC2 infrastructure, in order to benefit from all the features offered like elastic load balancing, service logging, and monitoring.
- Azure Service Fabric [131], the container orchestration platform developed by Microsoft for managing container-based microservices. This service not only provides container support, it also offers a Native Programming Model [132] focused on simplifying the design and development of microservices in their orchestration platform. Common orchestrator features like service discovery, health checks, and load balancing are automatically provided by the Service Fabric Platform.

2.3 Serverless Computing

With the popularization of mobile phone and web applications, developers realized that there was a real need to separate the user layer from the backend layer of the applications. With this in mind, the Backend as a Service (BaaS) model appeared. The concept behind the BaaS model is the same as the SaaS model, but with the application's developer as end user. Some services offered by public providers of this model are Google Firebase [75], AWS Amplify [12] or Azure Mobile [130]. The providers usually offer a set of Application Programming Interfaces (APIs) and Software Development Kits (SDKs) to the developers that implement features like social networking, location, and notifications and that can be used to connect the applications to cloud services like storage or user authentication.

Moreover, and continuing with the eagerness for executing lighter tasks in the cloud (in order to save time and money), the Functions as a Service (FaaS) model was created. In this service model developers focus on the bare minimum to develop cloud services, that is defining the behaviour of a piece of code (i.e. the function), while the service provider takes care of provisioning, scaling,

logging, etc. The most straightforward benefits from using this model are: delegation of the infrastructure management, high scalability, and pay-per-use (i.e. you only pay when your code is executed). However, this model also imposes some restrictions: the programming languages available to code your function are limited by the provider; it is a stateless service, meaning that the application state needs to be outsourced to external services, such as object storage services (e.g. Amazon S3) for files and managed databases (e.g. Amazon DynamoDB) for other kind of data; and finally, it is a system designed for short jobs, since long running jobs are not supported due to limited execution times.

Over the last five years, several FaaS providers that comply with the presented features have arisen: AWS Lambda [17], offered by Amazon Web Services, Google Cloud Functions [78], Microsoft Azure Functions [135], and IBM Cloud Functions [100] that uses the open-source Apache OpenWhisk [177]. These services allow the users to take advantage of the improvements offered by this new computing model.

After presenting both BaaS and FaaS models, we can introduce a definition for serverless computing. The Cloud Native Computing Foundation (CNCF) [46] defines in its whitepaper [47] the concept of serverless computing as:

> *Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.*

Although the terms FaaS and serverless are usually used interchangeably, the CNCF document also states that a serverless platform may provide one or both of the BaaS or FaaS services. Thus, serverless is used to generally speak about applications relying on managed services while FaaS refers to event-driven execution of functions on the computing infrastructure entirely managed by the cloud provider.

The serverless computing model aims to revolutionize the design and development of modern scalable applications, allowing developers to run ephemeral, event-driven code without provisioning or managing servers. Its evolution is reinforced by the continuous advances in container-based

technology together with the consolidation of cloud computing platforms. This new computational paradigm is experimenting an industry momentum around the cloud event abstraction [67], and as can be seen in Figure 2.2, serverless computing in combination with stream processing are the cloud services that more growth have experimented (a 50%) from 2018 to 2019. A further analysis and review of the serverless services is presented in the works of McGrath et al. [118] and Gannon [69], where they discuss the recent state of the art in this field and outline the potential of cloud event-based services.

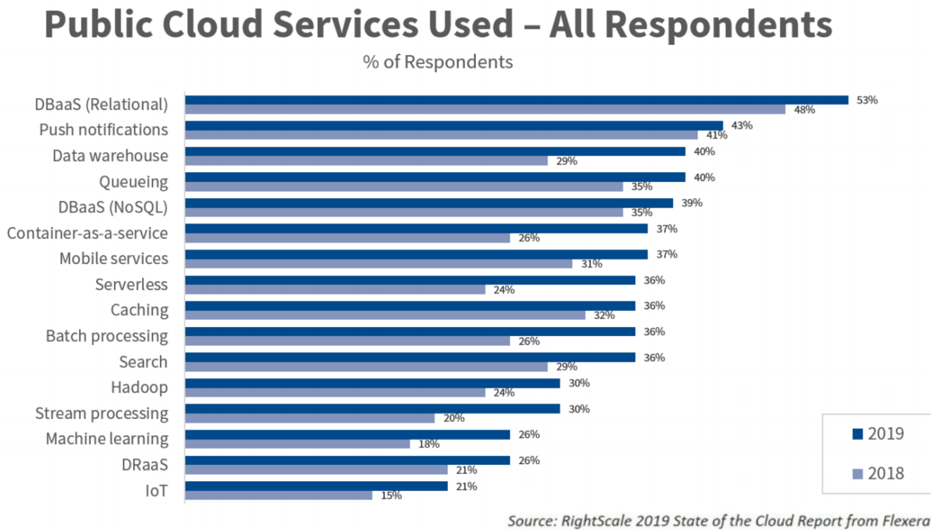

Figure 2.2: Cloud services usage in 2019 among 786 different IT professionals.

Bringing the benefits of event-driven serverless computing, especially concerning FaaS, to on-premises environments has paved the way for multiple open-source frameworks to appear. Some examples are OpenFaaS [143], Knative [84], Kubeless [37], Fission [156], Nuclio [138], FnProject [65], Riff [155], Funktion [68], OpenWhisk [177], and Qinling [149]. These platforms support the definition and execution of functions in response to events and they typically vary in the degree of support to multiple source of events, their support to programming languages and the usage of a certain container orchestration platform, such as Kubernetes.

As a summary of all the service models presented, Figure 2.3 shows the user and vendor responsibility levels depending on the service model. Remember that the BaaS service has the same responsibility levels that the SaaS model

but it is focused on the application developer and not the end user. Also remember that the serverless computing is composed by the last two columns (FaaS and BaaS).

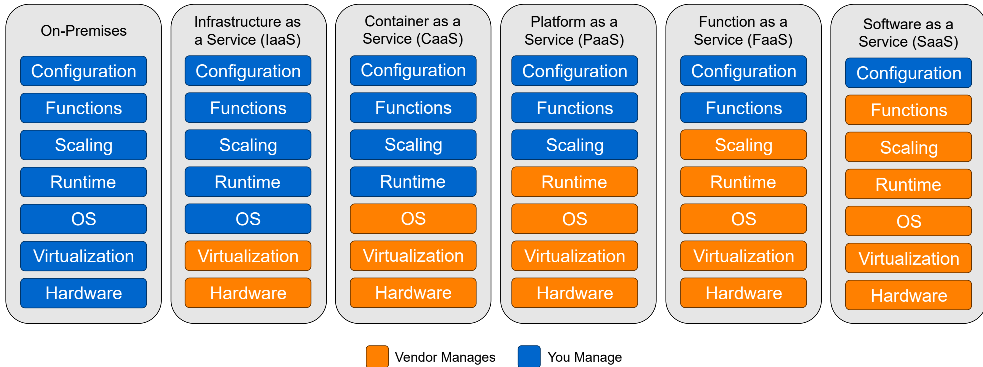

Figure 2.3: Comparison between different service models[2]

2.4 Serverless offerings

The following section summarises the serverless computing offerings by the most prominent cloud providers together with the existing open-source frameworks to achieve serverless computing in on-premises infrastructures. This section aims at helping the reader to better understand the design choices made when developing the frameworks presented in the subsequent chapters.

2.4.1 Serverless compute engines for containers

Before diving in the more specific FaaS tools and platforms, it is worth noting that the last couple of years have witnessed the appearance of serverless services oriented to run generic container applications in the Cloud. Although the idea of launching managed container services in the Cloud is not new (as we saw in section 2.2 with the CaaS platforms), these serverless container infrastructures are a step further in the abstraction of the management of container infrastructures. The idea behind this service is to allow the users to run containers on the Cloud while delegating server configuration and management to the providers. Some of the serverless

[2]Based on visualization from:
`https://medium.com/@pkerrison/pizza-as-a-service-2-0-5085cd4c365e`

container services available nowadays are: AWS Fargate [16], Google Cloud Run [80], and Azure Container Instances [125].

AWS Fargate is a service offered by Amazon that runs on top of the Amazon ECS platform (analyzed in section 2.2) and allows users to run containers without having to manage servers and their configurations. When the user selects the Fargate launch type in Amazon ECS, the only configuration required to launch the container is to specify the container image to use, the Central processing unit (CPU) and memory required, the networking policies and the access roles. There is no need to provision configure or scale the resources where the containers are running.

Google Cloud Run is the serverless container service offered by Google and it allows the users to run Docker based applications that are automatically launched when an HTTP request is received. Google Cloud Run uses the Knative platform [84] (which is analyzed in section 2.4.3) and like other serverless platforms it automatically manages the provisioning and configuring of the underlying servers. However Google Cloud Run imposes several restrictions for running a container [81]. Among others, the application inside the container should be stateless, the container should be listening for requests in the port 8080, the executables inside the container image must be compiled for Linux 64-bit, and the container must be up and running in less than four minutes.

Azure Container Instances is the solution presented by Microsoft to the serverless container platform ecosystem. The main difference with its competitors is that Azure Container Instances is focused on providing a fast and easy way to deploy isolated containers on the Azure Cloud, meaning that typical container orchestration features like auto scaling or service discovery are not covered by this platform. On the other hand, the main benefits of this platform are that it accepts Windows and Linux containers, it provides persistent storage through Azure Files [123], and it allows to deploy container instances that use Graphics processing unit (GPU) resources [124].

2.4.2 Public FaaS Providers

This section compares the most relevant public FaaS services to date: AWS Lambda [17], Azure Functions [135], Google Cloud Functions [78], IBM Cloud Functions [100], and Alibaba Cloud Function Compute [3].

First, Table 2.1 summarizes the different features offered by each service. Apart from the set of programming languages supported by each service,

notice that AWS Lambda allows to define custom runtimes, thus providing support to virtually any runtime; the drawback is that the user has to define the runtime and verify that the deployed libraries are compatible with the execution environment. IBM Cloud functions allows to define Docker containers as functions, which in turn would allow the users to execute any language they would need inside the container.

Regarding RAM memory size, AWS Lambda, IBM Cloud Functions and Alibaba Cloud Function Compute allow the users to set the memory available for each function in small increments, while Azure Functions assigns the memory automatically based on resource consumption and Google Cloud Functions offers a fixed set of five different configurations for the memory/CPU configuration. All the providers (except Google) transparently assign CPU resources based on memory allocation. It is important to point out that in AWS Lambda you get one full vCPU at 1,792MB [18], so when using memory configurations over 1,792MB, it is cost wise to deploy functions that can take advantage of the AWS Lambda multi-threading capabilities [166].

	AWS	Azure	Google	IBM	Alibaba Cloud
Service name	Lambda	Functions (Consumption Plan, runtime 2.x)	Cloud Functions	Cloud Functions	Function Compute
Languages	JavaScript (Node 8, 10) Java (8) C# (.NET Core 1.0 y 2.1) Python (2.7, 3.6, 3.7) Go (1.x) Ruby (2.5) Custom Runtimes	JavaScript (Node 8, 10) Java (8) C# & F# (.NET Core 2.2) Python (3.6)	JavaScript (Node 8, 10) Python (3.7) Go (1.11)	JavaScript (Node 8, 10) Java (8) Python (2.7, 3.6, 3.7) Go (1.11) Ruby (2.5) Swift (4.2) .NET Core 2.2 PHP (7.3) Docker	JavaScript (Node 6, 8) Java (8) C# (.NET Core 2.1) Python (2.7, 3.6) PHP (7.2)
Memory	From 128MB to 3,008MB	Automatic. From 128MB to 1,536MB	128MB/256MB/ 512MB/1,024MB/ 2,048MB	From 128MB to 2,048MB	From 128MB to 1,536MB
CPU	Automatic. Based on RAM. At 1,792MB of RAM 1 vCPU	Automatic. Up to 1 vCPU	Based on RAM: 200MHz/ 400MHz/800MHz/ 1.4GHz/2.4GHz	Automatic	Automatic
Disk space for function	512MB non-persistent	Uses Azure Files. Up to 5TB	tmpfs volume, consumes function memory	No information	512MB non-persistent
Max. code size	50MB zipped 250MB unzipped	None, you pay storage cost	100MB zipped 500MB unzipped	48MB	50MB zipped 500MB unzipped
Max. execution time	15min	10 min	9min	10min	10 min
Concurrent functions	1,000 per region (can be increased)	No limits. Depends on triggers.	No limit for HTTP execution. Other 1,000 per project.	1,000 per namespace (can be increased)	100 per region

Built-in triggers	Kinesis and Kinesis Data Firehose DynamoDB SQS ELB Cognito Lex Alexa API Gateway CloudFront (Lambda@Edge) S3 SNS SES CloudFormation CloudWatch (Logs & Events) CodeCommit AWS Config	Cosmos DB Event Hubs Event Grid Notification Hubs Service Bus (queues and topics) Storage (blob, queues, and tables) On-premises (using Service Bus) Twilio (SMS messages)	HTTP(S) Cloud Pub/sub Cloud Storage Stackdriver Logging Firebase	Alarms Cloudant database Messagge Hub Mobile push Github Custom (hooks, polling, connections)	Object Storage HTTP Time CDN Message Log API Gateway Datahub IOT

Table 2.1: Configuration options offered by public FaaS providers, as of October 2019.

Concerning disk storage space, FaaS systems are stateless by definition, i.e., the functions should be defined with no affinity to the underlying infrastructure. Therefore, no files should be stored between function invocations of the same function instance. Nonetheless, most of the services offer temporary storage space to allow the user process files during the function execution. An exception to the temporary storage policies is Azure Functions because it automatically mounts a drive in your function that connects to the Azure Files service [128], allowing the users to take advantage of their storage quota (and charging them for it).

Another limit to take into account when creating functions is the maximum code size. This limit restricts the size of the code, including libraries and dependencies that comprises the function deployment package. Whatever library, or file not present in this package has to be dynamically loaded during the execution of the function, thus prolonging the function's execution time. On the other hand, the larger this limit is, the longer it usually takes to launch a function for the first time, because the providers have to gather all the uploaded resources to create the environment where the function is going to be executed. Depending on the provider, the maximum code size can vary from 48MB to 500MB with the exception of Azure Functions that uses Azure Files and can storage up to 5TB (the Azure Files service is the only service that charges you for the storage of the function's code).

Regarding the maximum execution time, only AWS Lambda allows to execute a function more than 10 minutes (up to 15 min). The remaining providers allow to set the maximum execution time around 10 minutes, which confirms the statement that FaaS services are designed to process short-lived requests. Concerning concurrency (that is the number of parallel executions that happens at a specific time) we see that AWS Lambda limits the total concurrent executions within a given region to 1,000 instances. Likewise, Google Cloud Functions, IBM Cloud Functions and Alibaba Cloud Function Compute have a limit of 1,000, 1,000, and 100 function instances respectively, with the exception that Google Cloud functions presents no limits when invoking a function using a synchronous HTTP request. Regarding concurrent executions for Azure Functions, although there is no direct limit established, the number of concurrent function executions is restricted based on the number of cores of the underlying Virtual Machine (VM).

To finish, a list of built-in triggers is presented. AWS offers the longest predefined list of services integrated with its Lambda service (e.g. S3, DynamoDB, and Kinesis for data storage; Cognito for user sign-up, sign-in, and access control; Lex for speech recognition, etc), while others like Google

or IBM offer the possibility to define custom triggers based on HTTP(S) requests or webhooks. In addition, AWS offers Amazon EventBridge [9], a serverless event bus that allow users to connect any custom or SaaS application that generates events to other AWS tools like AWS Lambda. Similarly, Microsoft offers the Event Hubs service [127], which can also accept streams of data from several multiple sources and redirect it to other Azure services like Azure Functions.

Table 2.2, shows a comparison of the different pricing models available between the five selected public providers. The most important difference can be seen in the pricing model offered by Google Cloud Functions: they charge for memory and CPU separately, while all the other providers only charge per allocated memory and assign CPU shares correlated with the reserved memory. Another difference to remark is that IBM Cloud Functions does not charge per function invocation while all the others providers do.

	AWS	**Azure**	**Google**	**IBM**	**Alibaba Cloud**
Service name	Lambda	Functions (Consumption Plan, runtime 2.x)	Cloud Functions	Cloud Functions	Function Compute
Availability SLA	99.95%	99.95%	99.5%	99.9%	N/A
Billing increments for memory	64MB	128MB	5 sizes	1MB	64MB
Min. billed execution time	100 ms	100 ms	100 ms	100 ms	100 ms
Billing increments for execution time	100 ms	1 ms	100 ms	100 ms	100 ms
Function invocations (per 1M)	$0.20	$0.20	$0.40	Free	$0.20
Duration/Memory (per 1M GB-s)	$16.67	$16.67	$2.50	$17.00	$16.68
Duration/CPU (per 1M GHz-s)	N/A	N/A	$10.00	N/A	N/A
Network egress (per GB)	$0.09	$0.087	$0.12	$0.09	Depends on region (0.07−0.13)
Free invocations	1M	1M	2M	Free	1M
Free Memory (per month)	400,000GB-s	400,000GB-s	400,000GB-s	400,000GB-s	400,000GB-s
Free CPU (per month)	N/A	N/A	200,000GHz-s	N/A	N/A
Free network egress (per month)	Free in the same region. Part of overall EC2 free tier of 1GB	Part of overall free tier of 5GB	Free in the same region. 5GB to others	None noted for cloud functions	N/A

Table 2.2: Pricing summary for public FaaS provider, as of September 2019.

	AWS	**Azure**	**Google**	**IBM**	**Alibaba Cloud**
Service name	Lambda	Functions (Consumption Plan, runtime 2.x)	Cloud Functions	Cloud Functions	Function Compute
512MB 3M requests 1s	$25.60	$24.60	$28.95	$25.50	$25.61
128MB 30M requests 200ms	$18.50	$18	$25.86	$12.75	$18.50
128MB 25M requests 200ms **512MB 5M requests 500ms** **1,024MB 2.5M requests 1s**	$79.42	$76.5	$88.92	$74.38	$79.44

Table 2.3: Prices for different function configurations. Network storage fees, and free tier not included. Data valid as of September 2019.

To show a quick comparison of the pricing fees with some examples, Table 2.3 presents the calculated costs for different function configurations. The last price row corresponds to a combination of three functions, which represents an application composed by different functions that communicate asynchronously among them. The calculations have been done without taking into account the free tier offered for each provider. In addition, no network or storage fees have been calculated. As a remark of this table, we can see that since IBM Cloud functions does not charge for the function invocations, the IBM service becomes cheaper than the other options as the number of invocation grows. On the other hand Google Cloud Functions, which has an invocation price twice as expensive as the other providers ($0.4 instead of $0.2), increases considerably its service cost when dealing with a high number of invocations (e.g. 30 million invocations cost $12 instead of $6).

2.4.3 Open-Source FaaS Platforms

Open-source FaaS platforms appeared as an alternative to the public providers for users that needed more control over their FaaS solutions because these platforms can be adapted or extended to suit the users' needs. Furthermore, these alternatives do not restrict the users to private cloud deployments, since these open-source systems are based on container orchestrators that are also available in public providers (as is the case of Kubernetes). Therefore, these platforms can be deployed in either public or private clouds. However, the main drawback that arises when dealing with these open-source solutions is the added complexity to their management, since the deployment and operation of the platform needs to be performed by the user or system administrator. To provide additional insights in this area, the following paragraphs analyze the most relevant open-source FaaS platforms to date:

OpenFaaS

OpenFaas [143] is a FaaS platform based on Docker containers led by Alex Ellis, with contributions by the community, which supports the following container orchestrators: Kubernetes, Docker Swarm and OpenShift. It allows the users to run any code or binary application inside Docker images and offers autoscaling capabilities and metrics thanks to their API Gateway service and Prometheus respectively.

It allows to do HTTP requests to any Docker defined functions via a tiny HTTP server deployed inside each container (i.e. the function watchdog). Through the watchdog, the HTTP request received by the function is forwarded to the standard input of the container. When the function finishes, the standard output generated by the container execution is transformed into an HTTP response and returned back.

The default supported languages are NodeJS, CoffeScript, Go, Python, Java, .NET Core, R, and shell script, but as stated previously the users can extend the supported language by using a Docker template provided when defining a new function. Regarding the supported event sources, OpenFaaS provides the following connectors: HTTP requests, webhooks, the Kubernetes' cron jobs, and the OpenFaas Command-Line Interface (CLI). Also, via a provided event connector for pub/sub topics and message queues, OpenFaaS supports: Apache Kafka, AWS SQS, AWS SNS, AWS S3, Minio, CloudEvents, IFTTT, VMWare vCenter, Redis pub/sub, and RabbitMQ. Moreover, you can extend the event connector to other services by using the connector SDK [144].

Knative

Knative [84] is a Kubernetes-based platform, developed and maintained by Google, that allows to deploy and manage serverless workloads. Through a set of building blocks on top of Kubernetes, Knative allows to create container based functions taking advantage of all the functionalities provided by the underlying Kubernetes system.

Knative is comprised by two main components, eventing and serving. Eventing allows to manage and deliver events, while serving provides a request-driven compute that can scale to zero and through Istio [104] create traffic rules to balance loads across the functions. To build and deploy functions in Kubernetes, Knative uses Tekton Pipelines [173]. Tekton Pipelines allows the users to define tasks, to create Kubernetes jobs, and pipelines, to connect tasks, to create Continuous Integration (CI)/Continuous Delivery (CD) style pipelines, thus, for example, supporting the build of a Docker image from the source code available in GitHub.

Thanks to collection of loosely coupled components that can be used independently or together, Knative is used as a backend system for other open-source FaaS platforms (e.g. Riff [155]) or others are interoperable with it (e.g. OpenFaaS).

Kubeless

Kubeless [37] is a Kubernetes-native serverless framework built using the Kubernetes' Custom Resource Definitions (CRDs). All its internal functionality is enabled though Kubernetes, and among other features it offers function auto-scaling, through Kubernetes' Horizontal Pod Autoscaling feature [109], and monitoring backed by Prometheus [158].

The default event sources supported by Kubeless are: HTTP requests, Kubernetes' CronJob, Kafka, and NATS, but they can be extended using CRDs. The supported languages to define functions are: Python, NodeJS, Ruby, PHP, Go, .NET, Ballerina, Java, and any custom runtime that the user wants to add. Among other features, Kubeless offers a CLI compliant with AWS Lambda and it has been integrated in the Serverless Framework [168].

Fission

Fission [156] is a serverless platform for Kubernetes maintained by Platform9 [157]. Fission abstracts the Docker and Kubernetes management and offers the developers different environments to deploy their code. Those environments currently support Python, Go, .NET, NodeJS, Perl, PHP, Ruby, and any Linux executable (both binaries and scripts). In addition, users can extend the existing environments or create new ones by defining additional container images.

To provide a fast initialization of the functions, Fission keeps a pool of preloaded containers. Then, when a function is invoked for the first time, one of the 'warm' containers is launched. This feature allows Fission to offer cold-start latencies around 100ms. The autoscaling capability is based on CPU usage, where the user can set a target CPU at which autoscaling will be triggered. In the future its planned to allow the users to set custom metrics for scaling the functions.

Currently Fission supports the following triggers to launch functions: HTTP requests, alarms, and message queues. Other important features offered by Fission are: support to function workflows, live-reload for functions in test environments, replay of function invocations, and fully automated canary deployments.

Nuclio

Nuclio [138] is a FaaS platform oriented at real-time and data-driven applications. Nuclio claims to be able to process up to 400,000 invocations/second, which would turn this platform into the fastest open-source solution available in this respect. In addition, the platform is designed to allow its deployment in on-premises and multi-clouds environments, and even on low power devices.

The supported event sources by Nuclio are: HTTP, NATS, Kafka, Kinesis, RabbitMQ, Iguazio v3io, Azure Event Hub, and cron (locally invoked). The supported languages to define the functions are: Go, Python, .NET core, PyPy, Shell (invoke binary or script via exec), V8 (JavaScript and Node.JS) and Java.

Nuclio has recently started specializing in scientific applications, thus offering support to GPU processing and providing integrations with tools such as Spark, TensorFlow, and Jupyter notebooks.

FnProject

Fn [65] is an event-driven, open source FaaS platform that can run on any platform that supports Docker containers. Fn uses Docker containers as base for the deployed functions thus providing support to any programming language (although the officially supported languages are Go, Java, Python, Ruby, NodeJS, and C#/.Net Core).

The platform also provides Kubernetes support thanks to Helm templates. When the platform is deployed in Kubernetes, the Fn project also provides services such as Domain Name System (DNS), Transport Layer Security (TLS), metrics provided by Prometheus, and data visualization provided by Grafana [113]. Users can launch Fn functions from HTTP requests or from the Fn client provided.

Although it is a functionality still in development and currently only works for Java functions, Fn offers Fn Flow. Fn Flow allows users to build high-level workflows of functions. Some of the features of this workflows are: flexible model of function composition (allows sequencing, retrying, and error handling), code-driven (workflows are defined in code, not trough yaml files or visual graph-designers), and inspectable (one can drill on each stage of the workflow and check logs, and stack traces).

Riff

Riff [155] is an open source tool for deploying FaaS in Kubernetes. Riff offers three different function runtimes (core, Knative, and streaming) and three different languages to define your function (Java, JavaScript and Shell script).

Based on the runtime selected, the deployed functions use basic Kubernetes functionalities without load balancing or autoscaling (i.e. the core runtime), Knative to manage the autoscaling of HTTP-triggered workloads (i.e. the knative runtime) or the functions can be executed on streams of messages (i.e. the streaming runtime). In the streaming runtime messages flow between functions using streams, backed in this case by Apache Kafka [174].

Funktion

Funktion [68] is the Red Hat tool that offers an open source event driven Lambda-style programming model that runs on top of Kubernetes. Thanks to its integration with Apache Camel [24] the Funktion platform allows to define more than 200 triggers to launch its functions [25]. Unfortunately, Red Hat has stopped the funding of this project thus ending with the development of new features and the maintenance of the code.

OpenWhisk

OpenWhisk [177] is a FaaS platform managed by the Apache Foundation and created by IBM. As stated in section 2.4.2, OpenWhisk is used internally by the IBM Cloud Functions service, which turns it into one of the most stable and tested open-source solutions in the field (although the paying service offers more features).

Thanks to its open event provider interface, any service that implements it can be used as an event source. In addition, the default event sources provided are: HTTP, Github, Cloudant, IBM Message Hub, Mobile Push, Slack, Watson, Weather, and Websockets. The default supported languages for defining functions are: NodeJS, Swift, Java, Go, PHP, and Python. OpenWhisk also allows defining functions as Docker containers, so any other language can be integrated on the platforms. In addition, it allows to build workflows through the Composer programming model [101].

Qinling

Qinling [149] is the FaaS solution developed and maintained by OpenStack. Qinling uses Kubernetes as a default container orchestrator, but it allows to use as alternatives Docker Swarm, Mesos and Zun [152] (the container orchestrator solution used by OpenStack). In addition Qinling supports OpenStack Swift [150], and Amazon S3 as function storage backends.

By default, Qinling offers support to the Python programming language and provides the tools to allow the users to define their own language environments through the use of Docker containers.

Qinling is tightly integrated with the OpenStack infrastructure so the users can benefit from other OpenStack services like authorization and authentication (OpenStack Keystone [147]), an alarming service (OpenStack

Aodh [146]), or with a multi-tenant cloud messaging service for web and mobile developers (OpenStack Zaqar [151]). To connect the defined functions with external OpenStack services, Qinling allows to create webhooks that provide a unique Uniform Resource Locator (URL), and that can be used to call the functions without an authentication scheme.

2.5 State of the Art

Although it is a relatively new area, there exists in the literature several works contributing to the evolution of serverless computing. For example, the initial developments of OpenLambda are presented in [95] as an open source platform for building web services applications with the model of serverless computing. The work also includes a case study where performance of executions in AWS Lambda are compared with executions in AWS Elastic Beanstalk [15], which concludes with better performance results for AWS Lambda. The authors further evolved this platform to accommodate lean microservices that depend on large libraries that start slowly and have an impact on elasticity. For this, they introduce Pipsqueak [139], a package-aware computing platform based on OpenLambda. The work by Villamizar et al. [183] presents a cost comparison of a web application developed and deployed using three different approaches: a monolithic architecture, a microservices architecture operated by the cloud customer, and a microservices architecture operated by AWS Lambda. Results of this study show that AWS Lambda reduces infrastructure costs more than 70% and guarantees the same performance and response times as the number of users increases.

In addition, several tools related with serverless computing can be found in the literature. Some of them are: Apex [27], that facilitates the deployment of vanilla HTTP servers on serverless platforms; Podilizer [171] that implements the pipeline specifically for Java source code as input and AWS Lambda as output; and Snafu [170] that is a modular system to host, execute and manage language-level functions offered as stateless microservices to diverse external triggers. Other examples of recent works using serverless computing are open-source tools like Ooso [50], a Java library designed to execute MapReduce tasks based on Apache Hadoop and Spark on AWS Lambda, or enterprise solutions like Databricks Serverless [153], a serverless computing platform for complex data science and Apache Spark workloads. Moreover, projects like AWS Serverless Application Model (AWS SAM) [20] attempt to provide the means to define serverless functions for AWS Lambda.

In fact, AWS Lambda can be effectively exploited for scientific applications and there are few examples in the literature using it for distributed computing such as Bulk Synchronous Processing (e.g. PyWren [106]). This simplifies the access to distributed computing by avoiding to provision and configure complex clusters and, instead, define stateless functions to be run on the Cloud. The authors extended the previous work in the contribution by Shankar et al. [169] in order to introduce *numpywren* a system for linear algebra that runs on AWS Lambda. They also introduced LAmbdaPACK a domain-specific language to implement linear algebra algorithms that are highly parallel, assessing the increased compute efficiency achieved and highlighting the limitations of the Cloud provider. Indeed, the work by Spillner et al. [172] assesses the benefits of adopting serverless computing for multiple scientific domains: mathematics, computer graphics, cryptology and meteorology, using both public Cloud providers and self-hosted FaaS infrastructures. Other scientific applications in the literature are ExCamera [66], that allows fine-grained video processing, a distributed matrix multiplication system [186], or the work by Giménez-Alventosa et al. [70] that uses AWS Lambda to execute highly-parallel MapReduce jobs.

Use cases of the emerging event-based programming model can be found in the literature, like the work by Yan et al. [188] where the authors present a prototype architecture of a chatbot using the OpenWhisk platform, or the experiments described in [72] about face recognition with LEON, a research prototype built with OpenWhisk, Node-RED [107] and Docker. Furthermore, case studies of data analytics over serverless platforms, like [71], where the authors perform data processing with Spark over Apache OpenWhisk, are getting attention of researchers and developers. Regarding machine learning, in the work of Ishakian et al. [103] the AWS Lambda platform in combination with the MxNet deep learning framework [26] is used to serve deep learning models using serverless computing. This study concluded that serverless infrastructures performed within the acceptable latency ranges if the underlying function containers are initialized and waiting to be called (i.e. the serverless infrastructure is warm), a constraint that is not always met and it is not easy to control when dealing with public providers. A further analysis in serverless infrastructures states and their impacts on serverless workloads is presented in the work of Lloyd et al. [115], where they demonstrate how the performance of the functions can vary up to 15 times depending on which state the function is in.

Regarding serverless constraints, in one of the earliest works that evaluated serverless infrastructures [117], the first challenges are found: the serverless

applications need to be prepared in a portable way to be executed in the infrastructure, and the execution time limits hinder the adoption of long processing workloads. In addition authors such as Baldini et al. [34] address the problem of function composition entirely performed by serverless functions. Indeed, they demonstrate that function composition in serverless applications is achievable but exhibit several constraints to be considered such as avoiding double billing, adopting a substitution principle and treating the functions as black boxes. A subsequent work by Baldini et al. [33] identifies several challenges related to serverless computing. First, the ability to use *declarative approaches* to control what is deployed and the required tools to support it, and second, the *support for long running jobs*. The work by Erwin van Eyk et al. [62] also identifies some perspectives regarding the direction of the serverless field. They highlight the need to support hybrid Clouds, where an application could be composed by functions deployed on on-premises clusters, executing proprietary code, while other parts of the application could be run on the FaaS service offered by a public provider.

More challenges and improvent opportunites for serveless computing are presented in the work of Hellerstein et al. [94]. By analyzing three different case studies based on machine learning, data-centric, and distributed systems, the authors present several limitations that stifle the development of pure serverless applications. A posterior work from Jonas et al. [105] with a focus in increasing the types of applications that could work well with serverless computing also highlights more limitations currently present in serverless infrastructures such as inadequate storage for fine-grained operations, lack of coordination between functions, poor performance for standard communication patterns (i.e. broadcast, aggregation, and shuffle), and the lack of a predictable performance in part due to the variability in the provisioned hardware resources. However the same study concludes that the serverless usage will skyrocket in the following years and that serverless computing will overcome the presented challenges and it will become a fundamental part of the cloud technologies. The same conclusion is achieved in the study of Castro et al. [42], where the paper concludes that while it is true that the developers will need to know how to work around serverless limitations and how to map the Service Level Agreements (SLAs) of their applications to the new infrastructures, serverless computing is emerging as the new paradigm for the deployment of cloud applications.

Finally, the work by Adam Eivy [59] warns about the economic benefits of serverless computing, which strongly depends on the usage patterns and application workloads. Even though the pricing of services such as AWS

Lambda are billed in the fraction of 100ms of execution time, these can rapidly add up to surpass the cost of traditional computing approaches involving virtual machines or even dedicated hardware.

In conclusion, this chapter has covered an explanation of the different service models existing nowadays in the Cloud. Furthermore, a review of different providers and platforms both for public and on-premises FaaS infrastructures was done. Finally a review of the current state of the art in the serverless field has been carried out, highlighting scientific serverless tools, and the benefits and drawbacks present in the serverless architectures nowadays. Several scientific challenges have been identified such as restricted execution environments, limited execution time, lack of inbound connectivity in serverless providers or the lack of a predictable performance.

Therefore, this thesis aims to address some of the identified challenges and presents several solutions that can be added up on top of each other to finally allow scientific users to deploy and execute highly-parallel event-driven file-processing applications both in public and in private serverless-based Cloud infrastructures. To this aim, the following challenges are addressed in the corresponding chapters. First, chapter 3 tackles the challenge of providing generic execution environments on public serverless services. Then, chapter 4 presents a programming model that allows scientific users to create workflows of functions using a high-level declarative language without having to pre-provision infrastructure. Additionally, this chapter addresses the challenge of limited execution time and restrictive resource access by allowing the serverless application developers to connect the FaaS services with other managed computational services. Finally chapter 5 presents a platform that offers the event-based functionality of public serverless providers but for on-premises infrastructures.

Chapter 3

Serverless Container-aware Architectures

This chapter describes the creation of the Serverless Container-aware ARchitectures (SCAR) tool. Sections 3.1, 3.2, and 3.3 present the software architecture and its implementation. Then, a simple use case is shown in section 3.4. To finish, sections 3.5 and 3.6 carry out a study that analyses the cold start issue in AWS Lambda.

New architectural patterns (e.g. microservices), the massive adoption of Linux containers (e.g. Docker containers), and improvements in key features of cloud computing such as auto-scaling, have helped developers to decouple complex and monolithic systems into smaller stateless services. In turn, cloud providers have introduced *serverless* computing, where applications can be defined as workflows of event-triggered functions. Figure 3.1 shows a high-level overview of a basic event-triggered file-processing workflow. The generic workflow behaves as follows: the user uploads files to an object storage system. When the upload is finished the storage system automatically triggers the file processing service. Once the processing service finishes, the output files generated are stored in the object storage system.

Based on this simple use case and taking advantage of the highly-parallel capabilities of FaaS services, users can easily define highly-scalable event-based workflows to process their files. However, serverless services, such as AWS Lambda, impose serious restrictions for the applications executed

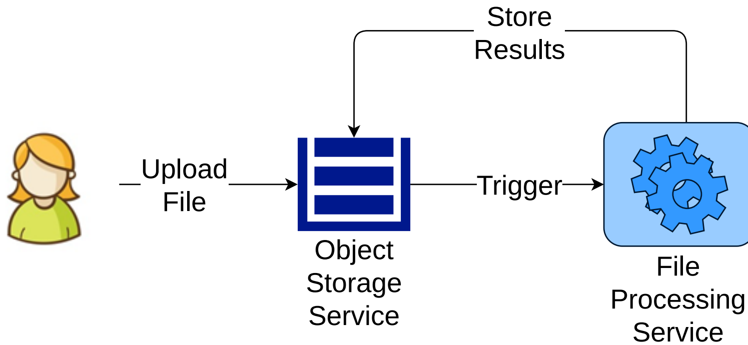

Figure 3.1: Generic high-level approach for event-based file processing applications.

inside their services (e.g. using a predefined set of programming languages or difficulting the installation and deployment of external libraries). Thus, the goal of this chapter is to provider end users / scientists with a highly scalable event-driven system to perform file processing on customized execution environments. To achieve this goal, this chapter introduces a framework called Serverless Container-aware ARchitectures (SCAR)[1] [154]. The SCAR framework, in combination with a simple programming model, can be used to create highly-parallel event-driven serverless applications that run on customized runtime environments defined as Docker images on top of AWS Lambda.

3.1 Generic Architecture

Figure 3.2 expands the black box schema presented in Figure 3.1 and describes the generic architectural approach designed to support container-based applications on FaaS platforms. This approach has been designed to maximize the integration of existing services typically offered by cloud providers. The main components used to define this generic architecture are:

- Function Service: in charge of executing cloud functions in response to events.

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

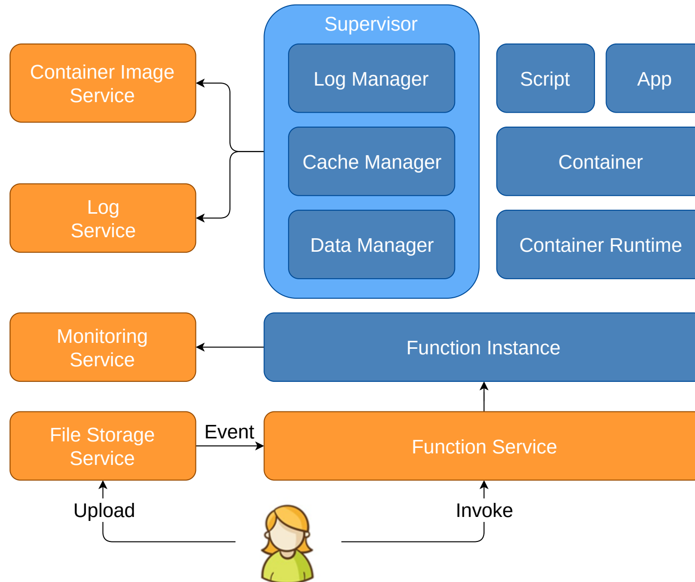

Figure 3.2: Architectural approach for supporting container-based file-processing applications on top of FaaS platforms.

- File Storage Service: hosts the files uploaded by the user and sends events to the function service so the files can be processed by the functions. N file uploads will trigger N function invocations where each one processes exactly one file.
- Log Service: where the information concerning the execution of the function is logged.
- Monitoring service: provides metrics of the resources consumed by the function. It also helps tracking the function response times, and failure rates.
- Container Image Service: service in charge of storing the container images that include the application to be executed, its required environment, and its dependencies.

In addition, an invocation of a function (i.e. *Function instance*) involves the execution of the *Supervisor*, responsible for: i) *Data Management* from the *File storage service* into the temporary data space allocated to that particular function invocation; ii) *Cache management* in order to minimize

the data movement from the *Container image service* to the data space available to the function; iii) *Log management*, to store the output of the execution of the container. The supervisor delegates on a *Container runtime* in order to instantiate a *Container* out of a container image, on which either a script or an application is executed inside the customized runtime environment provided by the container.

3.2 Framework implementation

The architecture designed allows to adapt SCAR to any cloud provider. However, since there are different API definitions implemented by such providers, it was initially chosen AWS Lambda as the serverless back-end. Thus, the following sections identify and justify the different technology choices taken while developing SCAR.

3.2.1 Serverless services: AWS Lambda

As described in section 2, there are several public serverless services that could be used to develop SCAR (e.g. AWS Lambda, Google Cloud Functions, Azure Functions, IBM Cloud functions). AWS Lambda was chosen due to its reliability, ease of use and popularity. AWS Lambda was the first cloud provider that offered the FaaS model and it has used this advantage in combination with additional services (e.g. AWS S3, Amazon DynamoDB, Amazon RDS, Amazon Kinesis, etc) and unique functionalities like edge processing and function chaining, to become the *de facto* platform for enterprise serverless computing [116].

However, the FaaS service offered by AWS Lambda imposes some restrictions that have to be taken into account when developing a framework on top of such service. The most important limitations imposed by AWS Lambda are:

- Constrained computing capacity currently limited by a maximum of 3,008 MB of RAM, where CPU performance is correlated with the amount of allocated RAM.
- Maximum execution time of 900 seconds (15 minutes).
- Read-only file system based on Amazon Linux.

- 512 MB of read/write disk space in the */tmp* folder, which *may* be shared across different invocations of the same Lambda function (the sharing capabilities are further analyzed in section 3.5).
- Default concurrent execution limited to 1,000 invocations of the same function (which can be increased up to 3,000 if requested).
- Supported execution environments: Node.js (8.10, 10 and 12), Java (8 and 11), C# (.Net Core 2.1), Python (2.7, 3.6, 3.7, 3.8), Go (1.x), Ruby (2.5), and custom runtimes.
- No inbound connections allowed for the Lambda invocations.

For the file storage service, Amazon Simple Storage Service (S3) was chosen. S3 is an object storage designed to provide durable and highly available access to files, stored in *buckets*, which are created in a specific AWS Region. An additional and useful feature of S3 is that it can automatically send event notifications when certain actions occur. For example, the *s3:ObjectCreated* event type is published whenever the S3 API calls PUT, POST or COPY are used to create an object in a bucket, that is, when a file is uploaded to the storage service. These events can be linked with different services such as Amazon SNS (a push messaging service), Amazon SQS (a message queuing service), and AWS Lambda, allowing the automatic invocation of such services. However, using the S3 link functionality as is and connecting several functions with one bucket implies that all of them will be launched each time a new file is uploaded. To avoid this behavior and to be able to allow a one-to-many ((S3 bucket) to Lambda functions) relationship without unwanted invocations, SCAR creates a unique structure of folders for each linked function based on the functions' data inside the S3 bucket, thus allowing to link several functions with one bucket, but only invoking one when a file is uploaded to the specific folder of such bucket.

Amazon CloudWatch [6] is the monitoring service provided by Amazon Web Services (AWS). In particular, CloudWatch Logs is a service to monitor, store and access log files produced from different sources and services in AWS. Therefore, the standard output generated by the AWS Lambda function invocations are sent to CloudWatch Logs in the form of log streams transparently to the user. By parsing such log streams the logs regarding the execution of a given invocation can be retrieved and shown to the user.

To provide all these functionalities, the client heavily uses the Boto 3 library [21] to interact with the AWS services.

3.2.2 Containers: Docker, Docker Hub and udocker

Among the different choices for Linux containers that we saw in section 2.2, we chose Docker due to its mainstream adoption for software delivery. This is exemplified by Docker Hub [53] a cloud-based registry service that hosts Docker images and can automatically create them by linking source code repositories such as GitHub, thus providing a centralized place to distribute Docker images.

Since external packages cannot be installed inside the execution environment of a invoked Lambda function (i.e. no root privileges are available to install Docker), a mechanism is needed to run a container out of a Docker image in user space without requiring prior installation or superuser rights. These features are available in udocker [180, 74], a tool to execute containers in user space out of Docker images without requiring root privileges. The udocker tool allows pulling images from Docker Hub and creating containers by non-privileged users on systems where Docker cannot be installed. This tool has demonstrated to be useful to run jobs on customized execution environments in both Grid environments, such as the European Grid Infrastructure [121], and High Performance Computing (HPC) clusters of PCs in the context of the INDIGO-DataCloud European project [102].

Udocker provides several execution modes, described in the documentation [73]. However, due to the restrictions of the execution environment provided in AWS Lambda, only the *F1* execution mode properly works, which involves using Fakechroot [165] with direct loader invocation. Using this approach, it is possible to run a process in the execution environment defined by a Docker image without actually creating a Docker container.

Notice that process isolation is automatically provided by the execution model of AWS Lambda, where different invocations of the same function are executed on isolated runtime spaces.

3.3 Architecture of SCAR

As exposed previously, SCAR allows users to define Lambda functions where each invocation will be responsible for executing a container from a Docker image stored in Docker Hub and, optionally, execute a shell-script inside the container for further versatility. Figure 3.3 describes the architecture of SCAR at a high level and presents the two main components that have been developed, the SCAR client, and the SCAR supervisor.

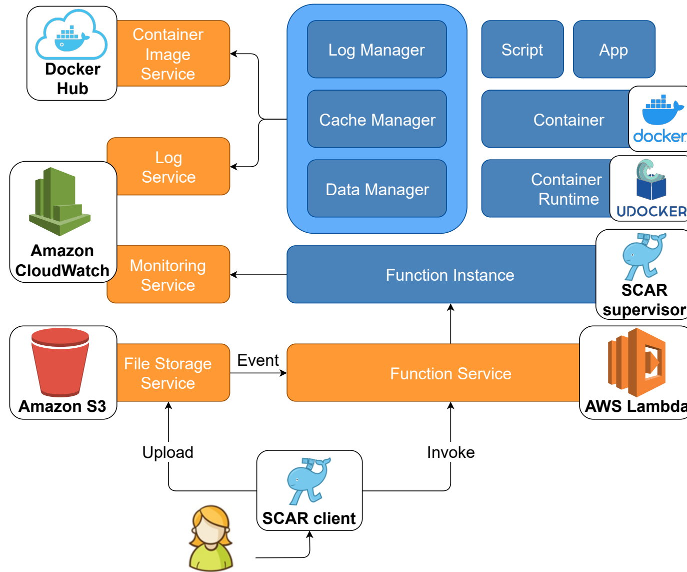

Figure 3.3: High level architecture of the SCAR framework.

3.3.1 The SCAR client

The SCAR client offers the users a Command-Line Interface (CLI) to manage the lifecycle of the serverless resources. Through the CLI, the user can initialize, launch, delete, and list the AWS Lambda functions as well as upload and download files from the S3 storage and check the generated logs in CloudWatch.

The input accepted by the client are command line parameters or a configuration file with a YAML Ain't Markup Language (YAML) format. The benefits of using the configuration file over the CLI parameters are, among others, a simplification on the infrastructure definition, and ease of the redistribution, repeatability, and reproducibility of the infrastructure configurations.

Once the input is passed to the SCAR client, the process of creating the serverless resources is as follows: i) the input configuration file or CLI parameters are validated; ii) the client gathers the required files, scripts, and libraries to create the deployment package; iii) the Lambda function, the CloudWatch Logs and the required S3 buckets and folders are created and linked automatically.

When the infrastructure is deployed, the user can invoke the Lambda functions also through the SCAR client. If the function invocation is synchronous, the client waits for the function to finish and automatically prints the generated output (i.e. the output of the executed script inside the container) in the command line. In addition, the SCAR client also offers the possibility to manage files stored in a S3 bucket (i.e. with the *put* and *get* commands), and also allows the user to retrieve a complete trace of the execution logs generated by the function execution. To see the full list of commands available in the SCAR client, please check Appendix A.

Additionally, the SCAR client is able to manage multitenancy in a transparent way to the user. The configuration files required for the client to work are generated in the user space, thus allowing different system users have different configuration files. Moreover, the client user can also define different credentials with different access rights and permissions to manage the AWS platform resources, allowing SCAR to handle different resources with different credentials if required. Finally, to avoid access to other user's functions from the SCAR client, each time a function is created, a set of tags are automatically defined. Among other properties, these tags include the user's name, so when the functions are listed, only the functions identified by the user name are shown.

3.3.2 The SCAR supervisor

The SCAR supervisor is a library that is executed in each function invocation to automatically manage the data stage in and stage out, and the container image. To ease the supervisor version management, the code reusability, and to provide a unified supervisor for all the created Lambda functions, the supervisor library is deployed as a Lambda layer [32]. Lambda layers allow to package libraries and other dependencies and share them across Lambda functions, thus avoiding to package and deploy the same library each time a new Lambda function is created. So, by creating a Lambda layer with the supervisor library in it, SCAR provides file management and generic environment capabilities to all the Lambda functions that use such layer.

Figure 3.4 summarizes the supervisor workflow that, once the function is triggered by an event, behaves as follows: 1) the event received is analyzed: if it has been generated by an S3 bucket, i.e. a file was uploaded to S3, the file that created the event is downloaded into the function's ephemeral storage; 2) the Docker image specified by the user is downloaded and stored also in the function's storage (if the image is already on the storage folder, this step

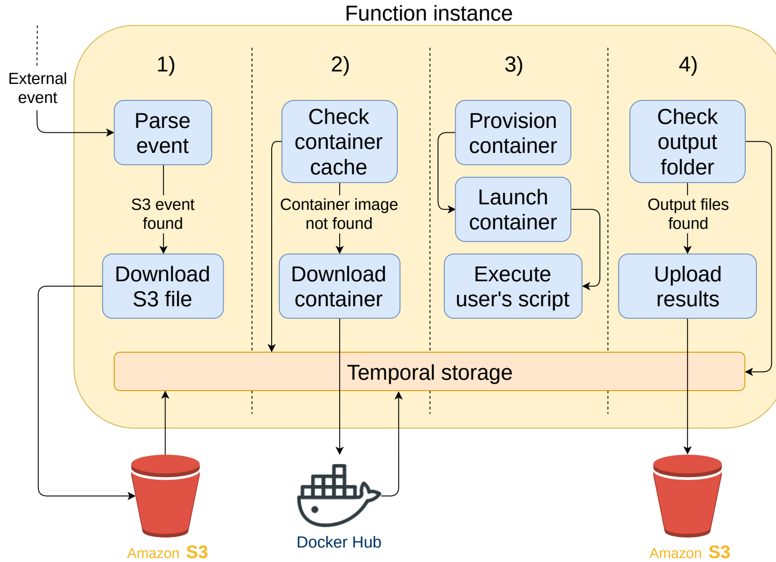

Figure 3.4: Workflow of the supervisor library inside a Lambda function instance.

is skipped); 3) the downloaded S3 file, the script that the user defined (if any) and the required environment variables are passed down to the Docker container when the udocker execution is launched; 4) after the execution, if the container has generated output files and an output S3 folder has been especified in the Lambda function definition, the generated files are automatically copied to the corresponding folder of S3 bucket and the function execution is finished.

In addition, the supervisor also traces the complete container execution and merges these traces with the function's log system.

3.4 SCAR usage

This section demonstrates how to execute an existing docker container using SCAR. This simple example is used to show the container execution capabilities inside the Lambda execution environment. The container executed combines two programs not available by default in the Lambda environment, the *fortune* package (that displays a pseudorandom message from a database of quotations), and the *cowsay* package (that generates

ASCII pictures of a cow with a message). In addition, the cowsay package needs the Perl libraries to be executed (a language not supported by the Lambda environment). The Docker image used is available in Docker Hub[2] and all the source code of the example is available in the SCAR's Github repository[3]. Additional more complex use cases are available in chapter 6.

The usage of SCAR in order to run this example on AWS Lambda is as follows:

- First, create a configuration file with (at least) the function name and the docker image to use:

```
cat >> cowsay.yaml << EOF
functions:
  scar-cowsay:
    image: grycap/cowsay
EOF
```

Listing 3.1: YAML creation script.

In this configuration file (i.e. *cowsay.yaml*) the user is specifying the function name as *scar-cowsay* and the Docker image to use as *grycap/cowsay*, an existing repository in Docker Hub.

- Second, initialize the Lambda function:

```
scar init -f cowsay.yaml
```

Listing 3.2: SCAR init command.

This command creates all the resources required to comply with the configuration file specified by the user. In this case, SCAR checks that the supervisor layer exists (if not, the SCAR client creates a new one), packages the required supervisor code, deploys the function in AWS Lambda and, finally, creates the log group in CloudWatch.

- Third, invoke the function:

```
scar run -f cowsay.yaml
```

Listing 3.3: SCAR run command.

[2] https://hub.docker.com/r/grycap/cowsay
[3] https://github.com/grycap/scar/tree/master/examples/cowsay

The client performs a synchronous invocation of the Lambda function specified in the configuration file, and then waits until the invocation ends. Once the invocation has ended, the output is printed in the user console:

```
Request Id: abd78643-cc74-4caa-b4b9-be730fbfa95b
Log Group Name: /aws/lambda/scar-cowsay
Log Stream Name: 2019/07/25/[$LATEST]507
   a50093815495da3c02ffdc5094510
 ________________________
/ It is your destiny. \
|                      |
\ -- Darth Vader       /
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Listing 3.4: Output for the SCAR run command.

The response returned by the function execution includes the output generated by the container execution (the talking cow in this case), and some extra information that can be of use to track the function execution. The extra parameters added to the container output are:

- *Request Id*: it is a unique identifier of the Lambda function invocation that is automatically generated by AWS Lambda service.
- *Log Group Name*: identifies the group of logs comprising all the invocations of functions with the same name.
- *Log Stream Name*: identifies the log stream generated by the function invocation. If the function is updated, fails, or too much time passes between invocations, another log stream is automatically created by the CloudWatch service.

3.5 On the Lambda function's ephemeral cache

The official Frequently Asked Questions (FAQ) of the AWS Lambda service [19] states that code executed inside a Lambda invocation "*should assume there is no affinity to the underlying compute infrastructure. Local file system access, child processes, and similar artifacts may not extend beyond the lifetime of the request, and any persistent state should be stored in Amazon S3, Amazon DynamoDB, or another Internet-available storage service.*" However, in the same FAQ, just in the question below it states that "*to improve performance, AWS Lambda may choose to retain an instance of your function and reuse it to serve a subsequent request, rather than creating a new copy.*"

Therefore, while it is true that the code of the function should be designed in a stateless manner to avoid errors, due to the freeze/thaw mechanism, the Lambda function environment can sometimes be reused and retain files from previous executions, as analyzed in the paper by Wang et al. [185]. Then, by understanding how the freeze/thaw cycle present in the AWS Lambda environment works, it might be possible to speed up the function invocations (by storing files shared among all the function invocations) and thus, reduce the infrastructure cost for the users. *Caching* is a fundamental technique used by SCAR that takes advantage of the aforementioned cycle. The following section presents a study of the freeze/thaw cycle in AWS Lambda, and how SCAR implements such caching behavior.

3.5.1 AWS Lambda Freeze/Thaw Cycle

As previously exposed, AWS Lambda functions offer closed stateless environments. The closed environment implies that the Lambda service executes the user code in a sandbox that isolates their processes, files, and environment variables from other function invocations. This stateless functionality also implies that this sandbox environment does not save any file or configuration among function invocations. Designing a caching functionality in this environment could seem rather challenging, but after analyzing how the initialization of AWS Lambda works a solution to the caching problem is presented.

As Figure 3.5 depicts, the first time a Lambda function is invoked, the underlying system has to gather the resources specified by the user, such as the memory, and provision a new sandbox with the user's code and the required dependencies. After the sandbox initialization, the function handler is launched and then the execution starts. Once the function finishes, if some

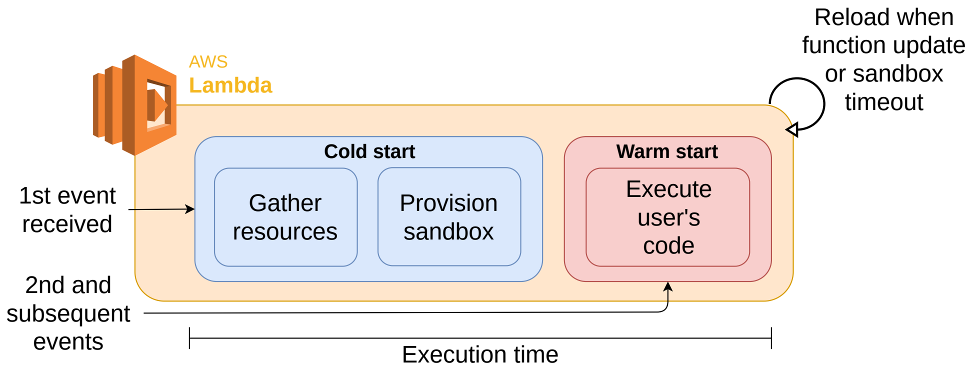

Figure 3.5: Lambda service behavior regarding function initialization.

time passes before calling the function again, the initialization process has to be repeated (i.e. the sandbox has to initialize, the user's code has to be loaded, and the handler has to be launched). This process is always repeated if the user changes the function's code or modifies the defined Lambda environment. However, if the user decides to invoke the function again without modifications and not too much time has gone by, the Lambda service may reuse the previously defined sandbox. When this situation occurs, the files that were written in the */tmp* folder in the last Lambda execution will continue to exist. To take advantage of this situation, we define a cache system to store and reuse the Docker containers inside the Lambda function storage. In the next section, the implementation of the supervisor's cache is presented.

3.5.2 The SCAR supervisor's cache

The first invocation of a Lambda function will pull the Docker image from Docker Hub into */tmp*, which can take a considerable amount of time (in the order of seconds), depending on the size of the Docker image. Subsequent invocations of the Lambda function *may* already find that Docker image available in */tmp* and, therefore, there is no need to retrieve the Docker image again from Docker Hub.

However, caching is not restricted to the Docker image. In addition, the container file system created with udocker, is also shared among all the Lambda invocations that may find it already available in */tmp*. The rationale behind this approach is that since Lambda functions are provided with a

read-only file system, so are provided the scripts executed in the containers executed in the Lambda functions. Notice that due to the stateless environment inherent to the Lambda functions, caching does not introduce side effects. It just reduces the invocation time whenever the cache is hit and the container is already available. Therefore, the duration of the Lambda function invocation using SCAR is greatly dependent on the ability for the Lambda functions to find a cached container file system.

Concerning the overhead introduced by SCAR, users should be aware that it requires a reduced amount of memory and disk space to run (~36MB of RAM and ~16MB of disk space). Indeed, it is the size of the container what really determines how much free disk space is available to be used by the deployed applications. Empirical experiments show that an image in Docker Hub larger than 220MB will hardly fit inside the ephemeral storage allocated to the Lambda function, due to the storage requirements of both the Docker image and the container file system unpacked by udocker.

However, thanks to modern minimal Linux distributions such as *Alpine Linux* [4] which offers a fully functional Linux distribution in only 2.6MB[4], and tools like *minicon* [88] that is able to minimize already existent containerized Linux distributions (e.g. a Node.js application along with the server reduced from 686MB to 45.6MB, or the size of an Apache server reduced from 216MB to 50.4MB) it is possible to create Docker images that comply with the AWS Lambda size constraints. On the other hand, the user has to bear in mind that tiny images like Alpine, usually change basic system libraries to be able to offer the reduced size (i.e. Alpine uses *musl_libc* instead of *glibc*), which sometimes can be associated with slower performances depending on the script or program executed by the user. As stated in the comparison table of musl with other different C/POSIX libraries [48], an script that requires thousands or millions of memory allocations, stated in the comparison table as *Tiny allocation & free* and *Big allocation & free* can present execution times up to 2x when compared with the standard glibc library available in Debian images.

To finish, regarding the remaining time available for the execution of the application, once the first invocation is finished and the container is cached, SCAR takes a negligible time to check if the container exists, thus allowing the application to run during almost the maximum time allocated to the function.

[4] `https://hub.docker.com/_/alpine/?tab=tags`

3.6 Study of the AWS Lambda Freeze/Thaw behavior

Due to the importance of the AWS Lambda's freeze/thaw cycle for the SCAR framework, an study of the behavior of this feature is conducted in order to extract optimized invocation patterns to be used by SCAR. For all the tests in this study, only the time used to download the Docker image from Docker Hub and run actions (i.e., creation of the container via udocker) are measured. The time spent by the script executed inside the container is considered negligible for this study. Also, to provide all the different tests with the same environment, the amount of RAM memory was set to 512 MB for all the functions, which is the minimum RAM needed to run the largest test in this study.

3.6.1 Request-response vs Asynchronous calls

The first comparison is done between the two different invocation types that AWS Lambda offers (i.e. *request-response* (or synchronous) and *asynchronous*) [22]. We analyze the time spent for each invocation. Each one involves creating a container out of a Docker image (in this case centos:7 [43]) stored in Docker Hub and executing a trivial shell-script inside. Ten different Lambda functions were created for each invocation type (i.e. request-response and asynchronous). Each function was invoked through the SCAR client as fast as possible a hundred times (i.e. a total of 2,000 invocations, 10*100 req-resp + 10*100 async). Figure 3.6 shows the average execution time for each invocation type.

The *req-resp* line refers to the *request-response* invocation type and it shows that all the invocations performed after the first one take a negligible time to execute. The first invocation takes 25 seconds on average to download the container image from Docker Hub, create the udocker container, and run the script inside it. Then, the subsequent invocations execute in almost negligible time (in comparison). This is a coherent behavior when using the request-response type in combination with the SCAR client, because all the function executions wait for the previous execution to finish. Thus, subsequent invocations after the first one will find the container file system already available in the ephemeral space of the Lambda function invocation (i.e. cached by SCAR).

The *async* line refers to the *asynchronous* invocation type and it shows an slightly erratic behavior in the execution time of the functions along the first 40 invocations. Indeed, the asynchronous model carries out all the

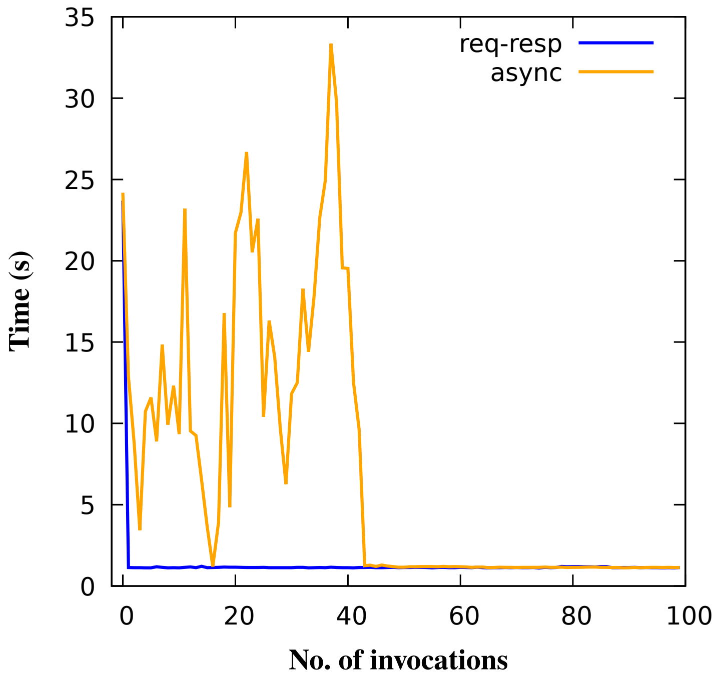

Figure 3.6: Average execution time for each invocation type.

invocations almost simultaneously, though there is a slight delay due to the SCAR invocation manager. Therefore, to serve all the invocations at the same time, the AWS Lambda service provisions different underlying sandboxes and consequently, these resources do not share the */tmp* space, so all of them need to retrieve the Docker image from Docker Hub and create the container. This means that the execution is performed successfully, but requires additional time. It can be seen that after approximately 40 invocations, the AWS Lambda service finally starts reusing the underlying sandboxes, so the container is finally found in the ephemeral cache and, therefore, the execution time of the subsequent invocations decreases considerably, as expected.

The main conclusion of this first experiment is that a newly created Lambda function with SCAR should be executed at least once, in order to cache the container in the ephemeral disk space, before attempting to perform multiple asynchronous invocations.

3.6.2 Container size

To further investigate how the container size affects the caching behavior when using the asynchronous invocation type, different functions were invoked that use different container image sizes in order to analyze the duration of the invocations. To carry out this experiment three different function types were created: the *minideb* type, which uses the Docker image *bitnami/minideb* [36] and has an image size of 22MB; the *c7* type, which uses the Docker image *centos:7* [43] and has an image size of 70MB; and finally the *ub14* type, which uses the Docker image *grycap/jenkins:ubuntu14.04-python* [86] and has an image size of 153MB. Each one of these types are used to create ten functions (i.e. a total of 30 different Lambda functions) and each different function is invoked a hundred times (i.e. 3,000 invocations in total, (10*100)*3). For the sake of clarity, the execution times of the functions belonging to the same type are presented as average values. Figure 3.7 shows the results of these invocations.

As expected, the *minideb* function, which represents the smallest container image, is the first one to present a cached behavior. The *c7* function, which has the medium size container image, is cached after approximately 30 invocations and the function with the largest container image, *ub14*, requires more invocations, approximately 80, before it is cached. This figure clearly shows the relation between increasing the container image size and the time that takes the container image to be cached in the ephemeral disk space. As explained above, this is directly related to the asynchronous way of working of the Lambda functions, in which the system does not wait for the previous invocation to finish. Therefore, the container image can be cached or not depending on the time passed since the first execution and the size of such container image.

As seen in the experiments, on the one hand there is the request-response invocation, where SCAR achieves a cached behavior starting from the second invocation, at the expense of waiting for that first invocation to end. On the other hand, there is the asynchronous invocation type, in which all the invocations can be carried out in parallel, but the container size affects the time until the containers are cached by SCAR in the ephemeral disk space.

In the programming model proposed by SCAR, the executions benefit from both invocation types by performing the first invocation as request-response and the reminder invocations as asynchronous. To assess the advantages of this approach, an experiment was carried out using the aforementioned approach. The same function types described in the previous test were used to carry out

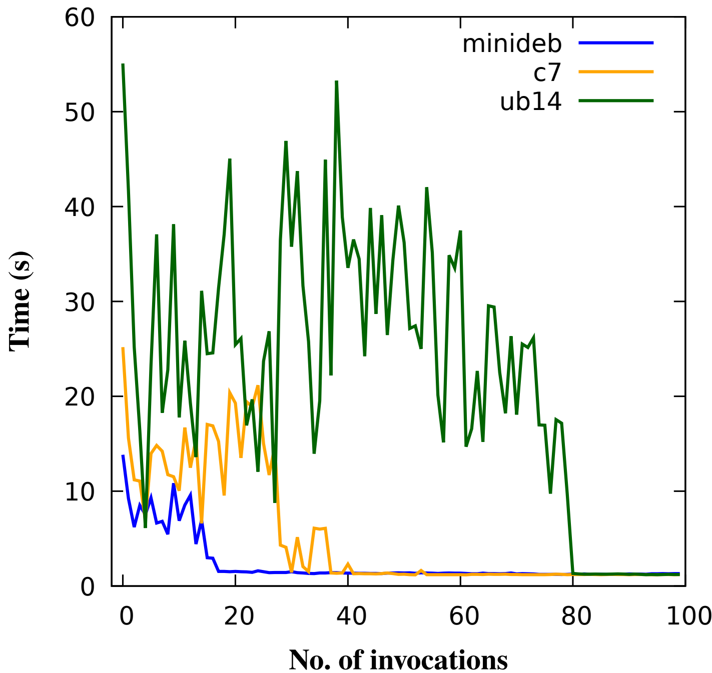

Figure 3.7: Average execution time for different container sizes. All the invocations are asynchronous.

this experiment: *minideb*, *c7* and *ub14*. These types were used to generate 30 different functions and each function was invoked a hundred times. Again, for the sake of clarity, the execution times of the functions belonging to the same type are presented as average values.

Figure 3.8 presents the results of this experiment. The erratic behavior of the caching system has disappeared and all the invocations present a cached performance starting from the second invocation and thereafter. This approach allows SCAR to reduce the overall execution time and, therefore, we adopt it for the event-driven file processing programming model. As such, a Lambda function created with SCAR is previously *preheated* (i.e. invoked with a request-response type), so that subsequent parallel asynchronous invocations already find the container cached in the ephemeral disk space.

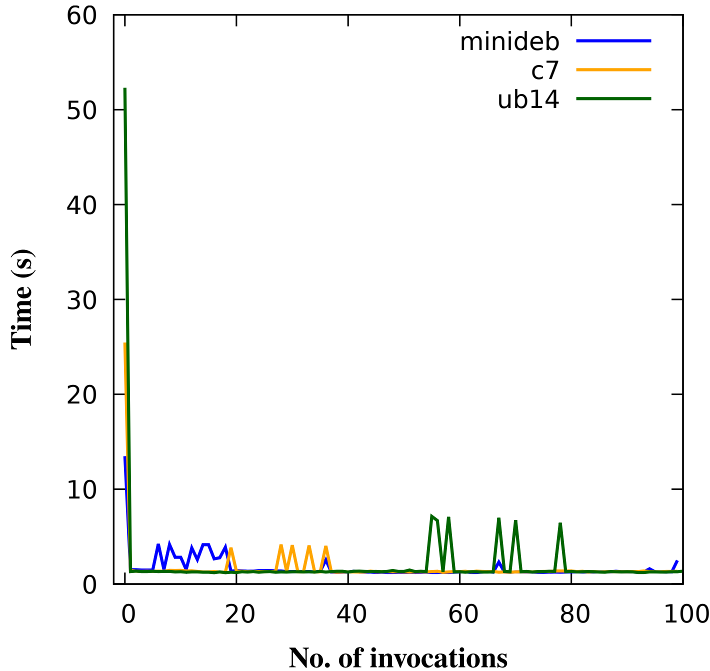

Figure 3.8: Average execution time for different container sizes. The first invocation type is request-response and subsequent are asynchronous.

3.6.3 Duration of the cache between invocations

The last experiment designed investigates the influence of the time between invocations on the ability for AWS Lambda to reuse the container, which ends up on SCAR being able to find a cached container, thus speeding up the Lambda function invocation. Since the benefits of *preheating* a Lambda function were identified in previous sections, it is important to know the time before it *cools down* again, i.e., when the invocation of the function will not reuse the same sandbox and, therefore, SCAR will have to retrieve the Docker image again from Docker Hub and create a new user-space container.

To this end, this experiment invokes the Lambda function in predefined ranges of time (each one of them defined by increasing 15 minutes the previous waiting time). Eleven time ranges were defined, from 0 to 150 minutes of waiting time.

A thousand different functions were created and invoked just once in each period, resulting in a total of 11,000 invocations launched.

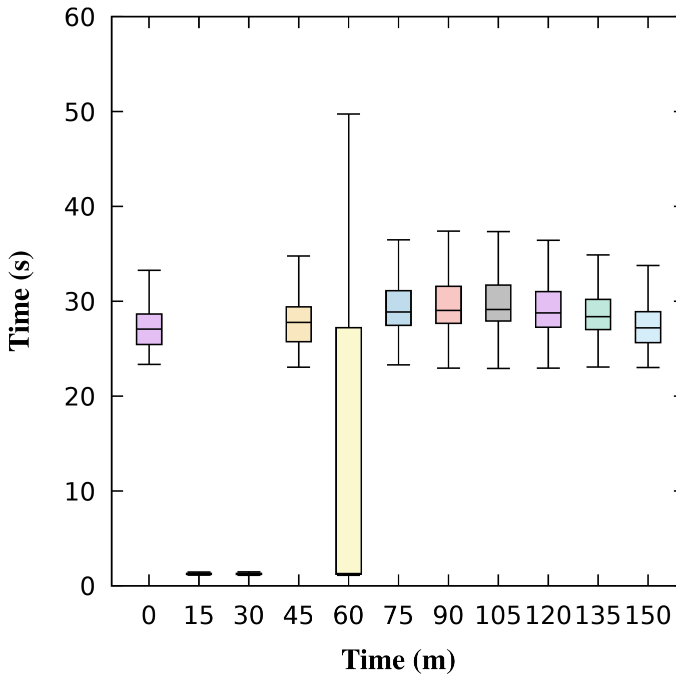

Figure 3.9: Execution time of the Lambda function related to the time waited (in minutes) from a previous invocation of the same Lambda function.

Figure 3.9 shows the results of this experiment using a box plot representation of the time used by each invocation. The whiskers of the box plot extend for a range equal to 1.5 times the interquartile range. By analyzing the figure we can extract several conclusions:

- All the functions show that the invocations are being cached between 15 and 30 minutes
- If we wait more than 60 minutes the cache is lost and all of our functions need to download again the container image
- Waiting 45 or 60 minutes do not ensure the correct working of the cache due to the randomness of the underlying system.

Additionally, note that the container size and therefore the cache size does not affect the time that the cache is maintained by the Lambda service. The duration of the cache is directly related to the time passed between invocations of the same Lambda function. These conclusions are in line with other studies that have also analyzed the duration of AWS Lambda functions as it can be seen in the works of Gimenez Alventosa et al. [70], and Wang et al. [185].

To finish this section we could remark that understanding the behavior of the freeze/thaw cycle and, therefore, when an invocation of a Lambda function created with SCAR will be cached, enables to adopt best practices when adopting serverless computing to execute generic applications. Furthermore, the knowledge gathered with this experiments is applied in the development of the SCAR framework to offer the users an efficient serverless programming model.

3.7 Conclusions

This chapter has introduced SCAR, a tool to execute container-based applications using serverless computing, exemplified using Docker as the technology for containers and AWS Lambda as the underlying serverless platform. This open-source tool is a step forward contribution to the state of the art, and opens new avenues for adopting serverless computing for a myriad of scientific applications distributed as Docker images.

Using the proposed framework, customized execution environments can now be employed instead of being locked-in to programming functions in the programming languages supported by the serverless platform (in our case AWS Lambda). SCAR has easily introduced the ability to run generic applications on specific runtime environments defined by Docker Images stored in Docker Hub.

In addition, SCAR not only provides means to deploy containers in AWS Lambda, it also manages the Lambda functions' lifecycle and eases the execution of the serverless workflow by applying optimizations without the need of user intervention, such as caching the container's underlying file system to minimize the execution time or adjusting the API call type to maximize the probabilities of finding the container cached in memory. Bursty workloads of short stateless jobs are specially appropriate to benefit from the ultra-elastic capabilities of AWS Lambda, both in terms of the amount of concurrent Lambda function invocations (in the order of thousands) and the rapid elasticity (in the order of few seconds).

However, the current limitations of AWS Lambda in terms of maximum execution time (15 minutes), maximum allocated memory (3,008 MB) and, most important, ephemeral disk capacity (512 MB), impose serious restrictions for the applications that can benefit from SCAR. To overcome these limitations, we developed an extended version of the high throughput computing programming model introduced in this chapter. This extended programming model and its new features and capabilities are presented in the next chapter.

Chapter 4

Event-Driven File-Processing Serverless Programming Model

This chapter introduces a programming model to create highly-parallel event-driven file-processing serverless architectures in combination with generic execution environments (i.e. containers) and the SCAR tool presented in chapter 3.

In the previous chapter we introduced a framework and a simple programming model that allowed users to define event-driven container-aware architectures. Although it worked for short lived jobs and small sized containers, the proposed framework couldn't overcome all the limitations imposed by the cloud provider. In this chapter we introduce an extension of the programming model in combination with a library (i.e. the supervisor) that allows the users to create serverless workflows that can bypass these imposed limits. Combining different services offered by AWS we are able to create serverless workflows that support the execution of long-running jobs with intensive computing requirements (i.e. high memory usage, big disk sizes or even the use of accelerated hardware such as GPUs).

The programming model assumes these two reasonable premises: i) the user wants to process a set of files that could be in a storage service or in a local machine; ii) after the function execution, the output files will be transferred to a storage service outside the space allocated to the Lambda function. Since Lambda functions are stateless so has to be the architecture using this service. Therefore, we only use the functions to process the data, leaving all

the data input/output management to the library implementing the programming model.

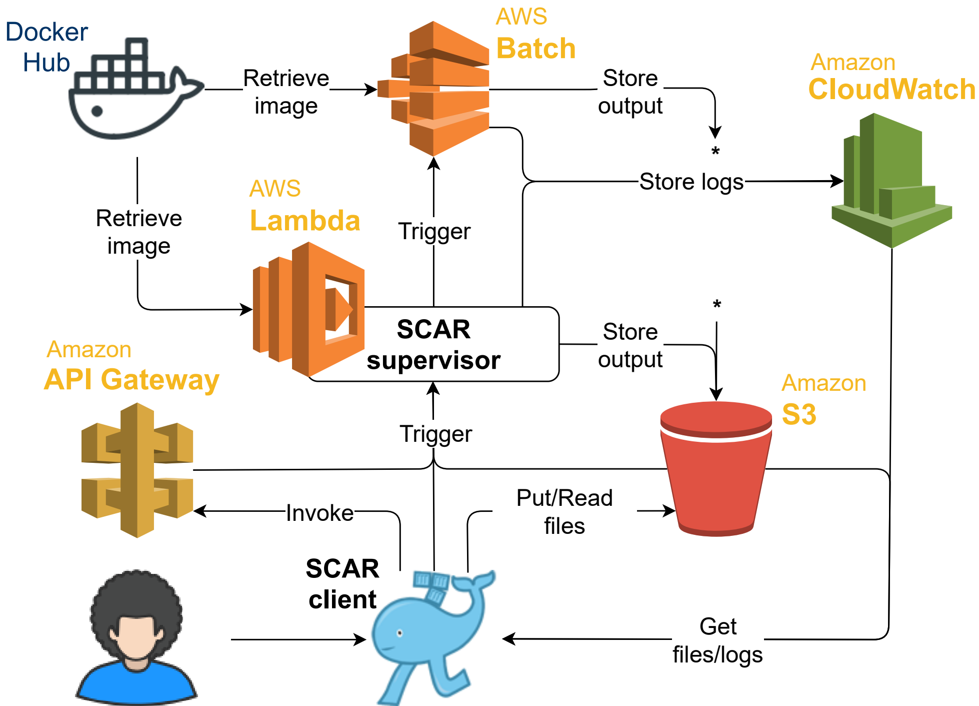

Figure 4.1: High level architecture diagram of the programming model proposed for creating event-driven file-processing serveless applications.

Figure 4.1 shows the proposed programming model. In addition to the AWS services used previously, that were AWS Lambda (for the functions), Amazon S3 (for the object storage), and Amazon CloudWatch (for logs and metrics), this extended programming model also makes use of:

- Amazon API Gateway [5]: is a service that supports creating, maintaining, and monitoring REST APIs at any scale. This service can be linked with AWS Lambda, allowing to provide REST endpoints connected with Lambda functions. This feature provides an easy to access entrypoint to the function workflow and allows to connect the

Lambda functions with any external service that can generate HTTP calls.

- AWS Batch [29]: this service offers a platform to run thousands of batch computing jobs. AWS Batch dynamically provisions the required compute resources based on resource requirements of the batch jobs submitted. Moreover, the jobs executed in AWS Batch have no execution time restriction and can use higher values of RAM. Therefore, we provided a gateway in order to delegate jobs from AWS Lambda to AWS Batch, to overcome the time limitation imposed by the Lambda service. In addition, the environment of a batch job is defined as a Docker container, thus ensuring that the same execution environment is going to be present in the Lambda functions and the Batch jobs. The auto-scaling capabilities of AWS Batch allow to execute a large number of jobs featuring the dynamic provision of computational resources.

Figure 4.1 shows how the users have different ways to process their files: 1) the user can submit a file to be processed using an HTTP request through a previously defined and linked Amazon API Gateway endpoint; 2) the user can upload a file to an S3 bucket that is linked with the Lambda function or read the files from a non-linked S3 bucket; 3) an S3 bucket, that could be connected to the same or another AWS Lambda function, can trigger the execution of more Lambda functions automatically, thus effectively implementing serverless workflows (as stated in section 3.2.1, SCAR automatically creates the required folder structure to provide this functionality).

Once a Lambda function has been triggered to process a job, the programming model offers three ways to process it:

- Using AWS Lambda default behavior).
- Using AWS Batch: when it is known that the job will take more than 15 minutes to be processed or needs access to more computing resources than those provided by AWS Lambda.
- Using a mixed approach between AWS Lambda and AWS Batch: when the requirements of the job are unknown. This approach executes the job in AWS Lambda, but if the job does not finish before the allocated function's timeout, then it is delegated to AWS Batch.

The supervisor behaves equally independently of the service that is processing the job, so the jobs executed in AWS Batch can also take

advantage of storing their outputs in a S3 bucket, and their logs in Amazon CloudWatch. In the following sections, the available workflows presented in this programming model are described in more detail.

4.1 Highly-scalable HTTP endpoints with API Gateway

The first initialization approach, shown in Figure 4.1, offers the users the capability to define HTTP-based entry points to AWS Lambda workflows. By providing this functionality, the programming model offers the users the capabilities of a highly-scalable REST API to invoke scientific services with automatic scaling to zero, and the ability to rapidly scale beyond the limits achieved by VMs in EC2. To provide this functionality, an Amazon API Gateway endpoint is required to be linked with the Lambda function to be invoked (this can be done automatically with the SCAR CLI). When the API endpoint and the Lambda function are linked, the HTTP request sent to the endpoint is routed to the Lambda service and preprocessed by the supervisor, thus leaving the content of the HTTP request ready to be used as a file by the container executed inside the function.

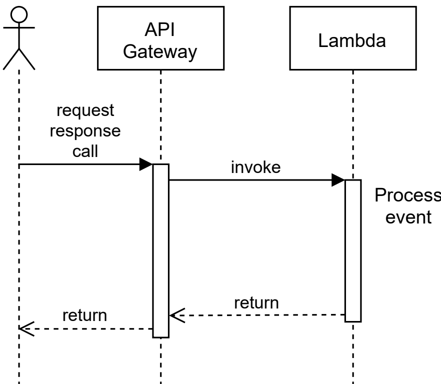

Figure 4.2: Synchronous sequence.

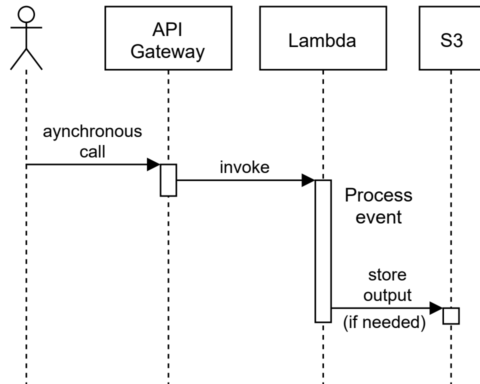

Figure 4.3: Asynchronous sequence.

As shown in figures 4.2 and 4.3, when invoking the function the user can perform a synchronous or an asynchronous invocation. A synchronous invocation instructs the API Gateway to wait for the Lambda function's response and to send back that response to the user[1]. On the other hand, an asynchronous invocation returns immediately the control to the user, but it

[1] A synchronous request to API Gateway has a maximum waiting time of 29 seconds. If the Lambda function takes more time to respond an error is raised

cannot ensure the correct invocation and execution of the function. The asynchronous invocation method can be useful to invoke almost simultaneously hundreds of function instances, allowing the user to take advantage of the high scalability provided by AWS Lambda.

It is important to point out that only synchronous calls to the API Gateway allow to return a response (e.g. a processed file) without using an external storage service. As shown in Figure 4.3, an asynchronous call has no connection with the SCAR client, thus making impossible to send the function response back. If an external file storage is used, then both types of calls allow to store the function responses.

4.2 S3 file upload/read triggers Lambda Function

The second initialization approach shown in Figure 4.1, and more in detail in Figures 4.4, 4.5, and 4.6, shows how Lambda functions can be invoked by uploading or reading files from an S3 bucket. This functionality allow users to upload files to an object storage to automatically trigger the file processing in parallel with no infrastructure pre-provisioning, featuring automated elasticity and without any economic cost if the application is not being used. In the following sections, the three ways of triggering functions using S3 buckets are described.

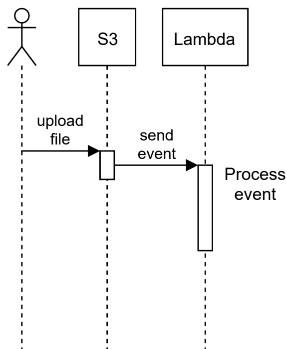

Figure 4.4: Upload a file to a linked S3 bucket.

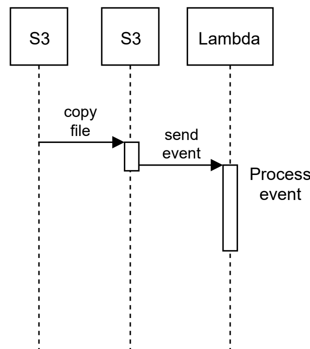

Figure 4.5: Copy a file to a linked S3 bucket from a non linked S3 bucket.

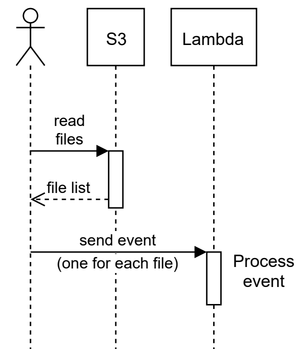

Figure 4.6: Read files from a non linked S3 bucket.

Upload a file to a linked S3 bucket

Figure 4.4 shows the sequence diagram followed when the user uploads a file to a linked S3 bucket. When the upload finishes, the S3 service sends an event to the AWS Lambda service with information about the created file. Then, the service invokes the Lambda function passing the event information. If a massive number of files is uploaded to S3 an internal queue is managed automatically by the Lambda service in order to retry the Lambda invocations for up to 6 hours, thus being able to cope with the increased workloads without any user intervention [23].

Copy a file to a linked S3 bucket from a non linked S3 bucket

As shown in Figure 4.5 the user can copy a file from a second available bucket to the bucket linked with the Lambda function. This will cause, as in the first case, to trigger an event from S3 to Lambda and the invocation of a Lambda function. By using this path the user takes advantage of the high transfer rates between S3 buckets and can have all the files pre-uploaded in S3, thus avoiding the need to upload a file each time a Lambda function needs to be invoked.

Read files from a non linked S3 bucket

Figure 4.6 shows the sequence that occurs when the Lambda function is invoked in parallel using an S3 bucket that does not need to be the source of events. To follow this path, through the SCAR CLI the user specifies a bucket where the files to be processed are stored. Then, for each file found, the client creates an event and invokes the Lambda function specified. This approach allows the users to take advantage of already existing data sets, as in the second path, and also reduces the invocation time between Lambda instances by invoking all of them following the pattern defined in section 3.6.2 (i.e. the first invocation is performed synchronous while the following invocations are made asynchronous).

Finally, the SCAR CLI allows the users to upload files to S3 buckets, and to read files from S3 buckets, without resorting to using the AWS tools (AWS CLI or AWS Management Console).

4.3 Data management inside the Lambda Function

After the S3 bucket sends an event with information about the uploaded file, or the API Gateway redirects the HTTP request to the linked Lambda function, AWS Lambda invokes the function passing in the event information as a JSON-structured document.

Figure 4.7 resumes the internal steps taken by the Lambda function since the event is received until the output files are uploaded again to an S3 bucket. The following paragraphs explain the internal steps carried out by the function supervisor developed:

- Create temporal folders: the supervisor's execution first creates two random folders (for input and output files) are created and their paths are stored as environment variables $\$TMP_INPUT_DIR$ and $\$TMP_OUTPUT_DIR$ for the Lambda function. By doing so, the user, through the defined shell script, can always know where the downloaded input files are, and where to store the container output folders so they are automatically managed by the supervisor.
- Process event: once the function instance is created, the received event is processed to check its source and an input file is created in the ephemeral storage allocated to the function invocation (i.e. /tmp). If the source is the API Gateway, the event body is read and its content is stored as the input file. If the event source is Amazon S3, the event information is further parsed to extract the information that allows the supervisor to identify the file, and then the file is downloaded from the S3 bucket using the Boto library. This step finishes creating a global variable (i.e. $\$INPUT_FILE$) that stores the path to the input file created. This global variable is exposed to the programming model for the user to know the path to the data file to be processed by the shell-script.
- Check container cache: after the event has been processed, the supervisor checks if the required container's image and file system are available in the Lambda function's cache located in the ephemeral storage path of the function (i.e. /tmp). If the container image is available in the cache, then the download of the container from Docker Hub is skipped. Otherwise, the container image is retrieved from Docker Hub and then stored in the aforementioned ephemeral storage folder. Afterwards, the supervisor checks if the container file system is also available. If the file system is found this initialization step is also skipped, and when the container's

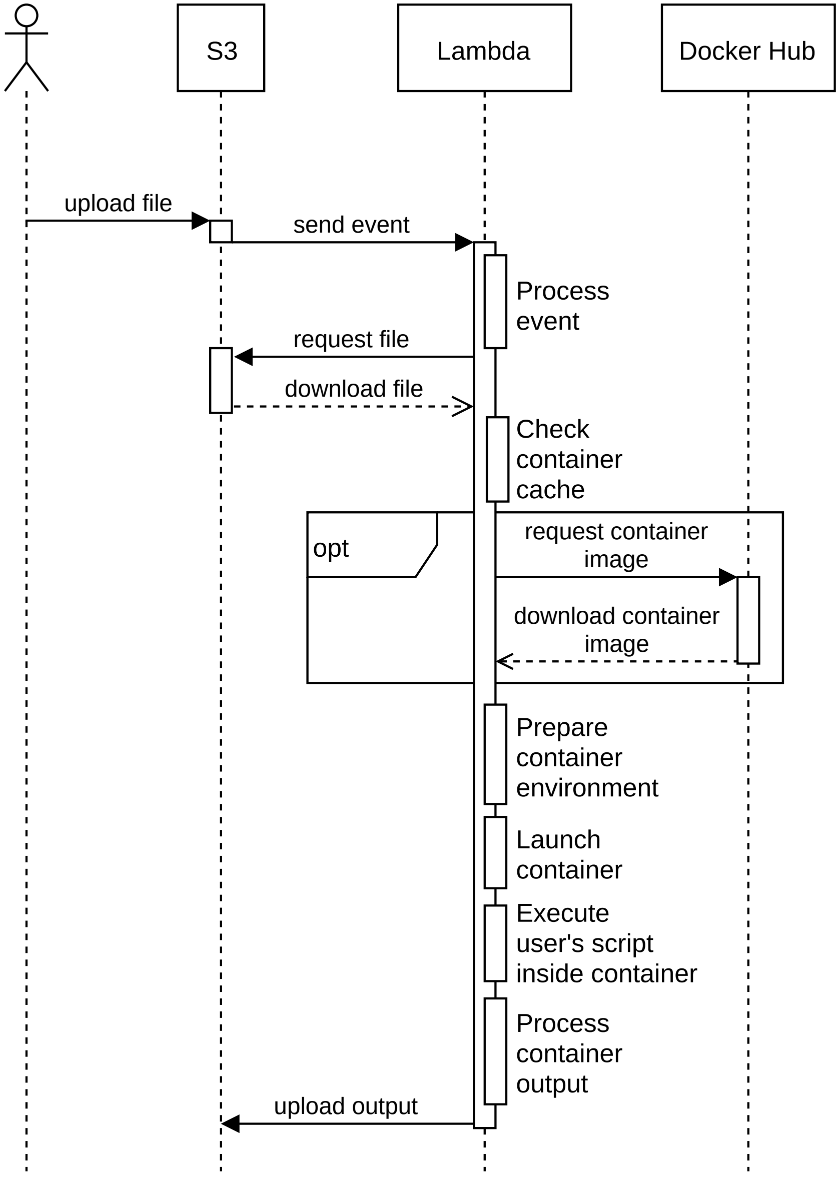

Figure 4.7: Data management inside the Lambda Function.

file system is not found, the supervisor automatically creates it through udocker and stores it also in the ephemeral storage folder.

- Prepare container environment: once the container is available, the environment variables and the temporal output data folder required is created (if required). Additionally, several global variables from the AWS Lambda environment are injected in the container: the extra payload path, if there is any, ($EXTRA_PAYLOAD$), the function's request id (($REQUEST_ID$), the temporary input/output folder variables created at the beginning, and any other environment variables that the user previously defined with the SCAR CLI). To finish, the supervisor provides the container with the script uploaded by the user.

- Launch container: when all the environment is prepared, the container is launched with a timeout that can be defined by the user (the default timeout is the function's execution time minus 10 seconds, that is enough time to upload any file that can be generated by the function's execution). If the container does not finish on time, the execution of the container is forced to finish and if the user defined it, the job is delegated to AWS Batch (the default behavior is stopping the execution and throwing an error message). If the container finishes successfully then, the only step remaining is processing the generated output.

- Process container output: as the final step, if the container generates output files and the Lambda function is linked to an output source, the files are automatically copied to the specified folder of the output source. In Figure 4.7, the output source is an S3 bucket, so just before the Lambda function finishes, all the files are uploaded to the corresponding bucket.

As the last step in Figure 4.7 shows, if a container generates output files, the supervisor can automatically upload them to a S3 bucket. From the S3 service perspective, uploading a file from the SCAR client or from the supervisor is the same, so after initializing the workflow from the SCAR client by uploading a file, such workflow could automatically be continued without user interaction, and thus new functions could be launched, if the output file generated by the first invoked function is uploaded again to S3 by the supervisor. This approach is further analyzed in the following section.

4.4 Output files trigger new Lambda functions

The third and last approach presented in Figure 4.1 and in Figure 4.8 allows the user to define a chain of functions communicated by the events generated by Amazon S3.

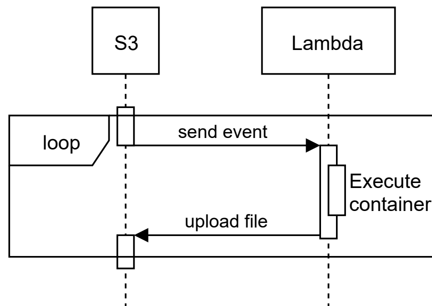

Figure 4.8: Launching Lambda functions using S3 buckets without human intervention.

By using this approach, the user can define input and output buckets and folders for each of the Lambda functions created, making the output folder defined for one function the input folder of another function. Then, the user only has to invoke the first function of the chain using one of the other invoke alternatives defined above. The remaining linked functions are invoked by the respective events created by Amazon S3 upon the creation of the files. This approach paves the way to define data-oriented workflows, even supporting recursive function invocations.

This simple approach for connecting functions allow the platform users to create scientific workflows modeled as a series of data processing functions and intermediate storage services. The scientific workflows defined using this approach are data-oriented, can be easily reused and refactored, provide intermediate storage points so the data manipulation can be tracked, and finally and most important, allow to define complex and generic workflows without pre-provisioning any infrastructure. Applications designed following this model can be created on the Cloud and will wait to be called without incurring in cost for the maintainer of the application. This is clearly a step forward in cloud application design and liberates the developers from having to calculate the best trade off between the amount of provisioned resources to provide a good service and the service cost.

4.5 Job Processing with AWS Batch.

In the early years of computer science, batch processing appeared to allow users run or schedule jobs without the need of continuous interaction. Queuing job requests for processing, in contrast to managing every job one by one, permitted users to delegate resource scheduling and allocation to the underling system. Thus, batch processing is greatly used in many different fields where computing needs are huge (in the order of petabytes) and resources need to be shared (like in on-premises infrastructures), such as particle physics, weather forecasts, or space data analysis. However, some of these jobs are so demanding that even with all the resources of a private cluster a response cannot be provided in a reasonable amount of time. Fortunately, with the rise of cloud computing, batch processing became available in the public Cloud, where most of the restrictions of on-premises clusters are overcome.

Regarding AWS, Amazon offers the AWS Batch service. This service provides users with a simple and efficient way of running hundreds of thousand of batch computing jobs. AWS Batch automatically manages all the underlying resource allocation needed to run the submitted jobs. However, before running any job, the users need to specify what resources from the pool of virtual machines available in AWS EC2 can be used and with what parameters should be spawned. This parameters, such as the EC2 instance type (which can be provided with GPU support if needed), the maximum or minimum virtual CPUs, or the network configuration, define the compute environment that is going to be used for creating the underlying virtual infrastructure that will execute the batch jobs. In addition to the compute environment, a job queue and a job definition must be created in order to use AWS Batch. The job queue is used to connect compute environments with jobs, and with a job definition, the user sets the resources needed for the job that is going to be processed (i.e. the container image to use for the job environment, the RAM memory and CPUs required, etc.). After all the required configuration is done, users can define batch jobs that run inside Docker container images, with no execution time limits in an infrastructure with the capability of scaling down to zero or scaling up to process hundreds of thousands of jobs. However, the main drawback of this service is its provisioning time, that depending on the resources selected can vary from one to more than fifteen minutes, thus making this service more suitable for long running jobs with bigger requirements than the jobs executed in AWS Lambda. In consequence, a combination of both services, AWS Lambda and AWS Batch, would allow us to provide support for a wider

set of jobs while keeping serverless features such as scaling down to zero, scaling up to thousands of requests, and pay-per-use.

As exposed in section 3.2.1, among other limitations, Lambda functions are restricted to 15 minutes of maximum execution time, 3,008MB of RAM, 512MB of hard disk, and cannot have access to accelerated hardware devices such as GPUs. By allowing the user to define AWS Batch Compute Environments and combining AWS Lambda invocations with Batch jobs, it is possible to overcome these limitations. SCAR does not only automatically redirects the Lambda jobs to the Batch infrastructure if needed, additionally, SCAR simplifies the AWS Batch compute environment, job queue and job definitions by automatically creating all the required AWS configurations based on the SCAR configuration file[2]. The users do not need to manually manage any AWS Batch configuration because the complete lifecycle of the Batch infrastructure can be done though SCAR.

Figure 4.9 shows the sequence followed when a user defines a job to be executed directly in the AWS Batch environment. As in the jobs executed in AWS Lambda, the Batch supervisor allows the access to storage services like S3 without user interaction.

As summary of this section, it can be concluded that thanks to AWS Batch the users can define elastic virtual clusters with GPU support, automated scaling to zero, and the ability to self-deploy additional working nodes to cope with increased workloads for event-driven compute-intensive file processing. Bear in mind that currently AWS Lambda only offers support to CPU based executions, thus limiting the different jobs that can be executed, with this approach it is opened the definition of serverless workflows to many different applications that would require GPU resources. Additionally, combining AWS Lambda invocations with Batch jobs allow the scientific users to define workflows that support a combination of short and long jobs while keeping serverless features such as no configuration management, and the pay-per-use model.

[2]The complete set of variables that can be defined for the batch environment can be found in the SCAR configuration file that is automatically created when SCAR is executed for the first time. Also more information about the batch mode and the batch configuration con be found in the SCAR documentation `https://scar.readthedocs.io/en/latest/batch.html`

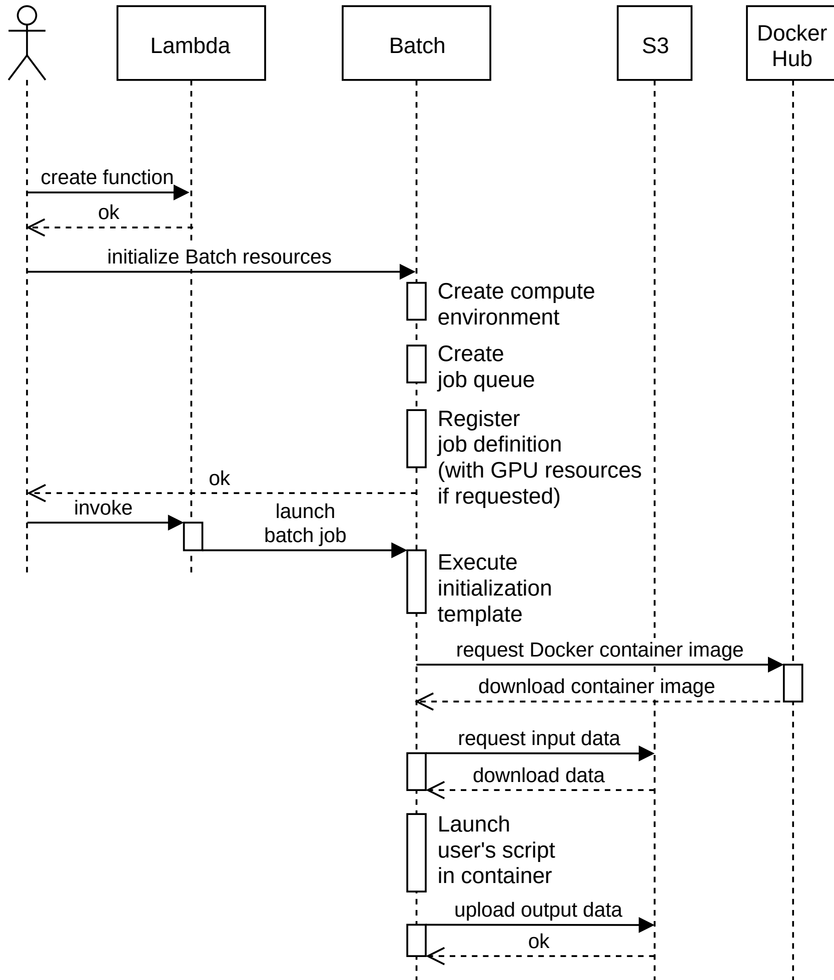

Figure 4.9: Launching AWS Batch jobs through AWS Lambda.

4.6 Cost analysis

To test the scalability and parallel file processing capabilities of the proposed programming model we are going to deploy a medical image analysis service in which several workloads are going to be tested (processing 1, 10, 100, and 1,000 images). The tool used to analyze the images was presented in the paper by Torro et al. [179]. This tool employs the Diffusion-Weighted

Imaging (DWI) method [35] to extract meaningful information about the microscopic motions of water in human tissues. The application receives as input a Medical Resonance Imaging (MRI) file to analyze, among other parameters. When analyzing the MRI file, the application uses OpenMP to perform automated parallelization thus taking advantage of the multi-threading capabilities in AWS Lambda functions with more than 1,792MB (as explained in section 2.4.2). The application binary used in this test case, the Dockerfile to create the container image, and the script executed can be found in the SCAR's repository[3]. After executing the different workloads, it will be carried out a cost analysis of the cloud infrastructures used in this experiment.

This case study was executed in four different environments:

- A local PC machine with 4 CPUs (model i5-4590) and 8GB of RAM: to provide a ground case for the execution time, a standard PC was used.
- A c5.large EC2 machine with 2 virtual CPUs and 4GB of RAM: the most basic machine available in AWS EC2 for computer-intensive workloads. This machine was selected to test the computation capabilities of the most basic compute-intensive machine available in EC2.
- A c5.18xlarge EC2 machine with 72 virtual CPUs and 144GB of RAM: the second biggest machine available in AWS EC2 for compute-intensive workloads. This machine has large parallel capabilities and it was selected to compare the monolithic approach of one big VM against the decoupled processing approach offered by Lambda functions.
- A Lambda function with 3,008MB of RAM and without concurrency limits defined.

To avoid the cold start of the Lambda functions (see section 3.6, together with the works by Wang et al. [185] and Pérez et al. [154] for additional details), the SCAR client *preheats* the Lambda function doing a synchronous first call during the initialization step. Once the initialization finishes, the function invocations are performed asynchronously and executed in parallel. Figure 4.10 shows the execution times for the different infrastructures involved. In this figure, the vertical axis (i.e. seconds) is represented using a logarithmic scale due to the big differences in the execution times of the different workflows.

[3] `https://github.com/grycap/scar/tree/master/examples/dwi-ivim`

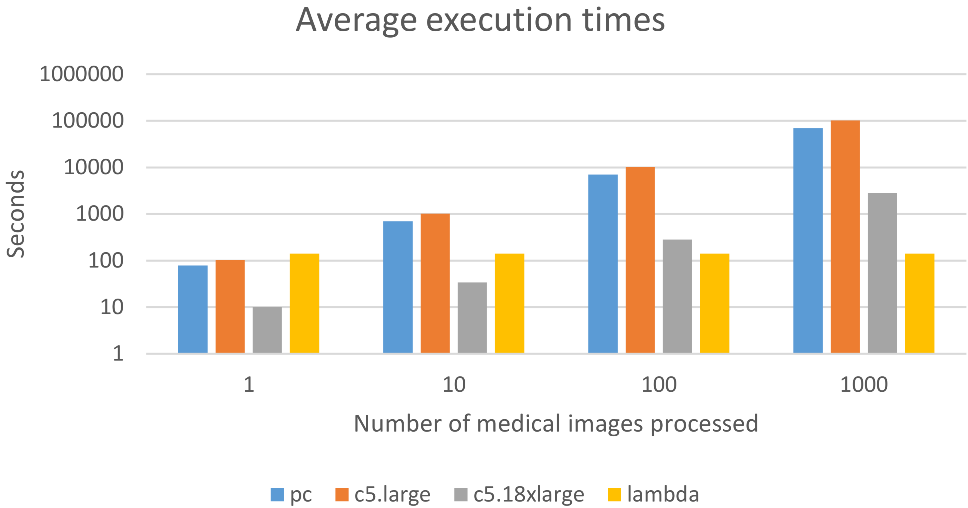

Figure 4.10: Average execution times to process different number of images in different execution environments.

Figure 4.10 summarizes the results of the proposed workloads. It can be seen how the local PC and the c5.large machines scale almost at the same rate due to its similar parallelization capabilities. Processing one image already requires all the computing power available in these machines, so when the number of images to process is ten times bigger, the time used to process such images also grows ten times. On the other hand, processing one image with the c5.18xlarge machine outperforms the rest of the platforms tested (around ten times faster) thanks to its high computing power. When using the same machine to process ten images, the large parallel capabilities of its architecture still beat the rest of tested platforms, as it can be seen how the time taken to process the images doesn't grow at the same rate that the number of images processed (the number of images is ten times more and the increase of time is five times more). However, it can also be seen that processing one hundred, and one thousand images saturates the parallel capabilities of the c5.18xlarge machine and the processing time starts growing at the same rate than the number of images.

Finally, it can be seen how processing one image with the Lambda function takes more time than any of the other alternatives, this is due to the average processing capabilities of the underlying resources of the Lambda functions (2

vCPUs and 3GB of RAM). Still, it can also be appreciated that the Lambda infrastructure takes the same time to process one, ten, one hundred, or one thousand images, thus outperforming the rest of platforms when a high level of parallelism is required (in the one hundred and one thousand tasks). This results are in line with the parallel capabilities announced by the Lambda service, which is designed to process embarrassingly parallel workloads.

4.6.1 Cost analysis of the programming model

In this section it is going to be analyzed if it is cost-wise to use the proposed FaaS programming model in comparison to a more established computational paradigm that are the virtual machines in the Cloud (i.e. AWS Lambda vs AWS EC2). To calculate the cost, the last workload of the case study presented in the previous section is going to be used, that is, analyzing the cost of processing 1,000 medical images with AWS Lambda and with two types of AWS EC2 instances (c5.large, and c5.18xlarge). Table 4.1 resumes the AWS infrastructures used and their respective properties.

AWS Service	Type	Cost ($/h)	CPU units (ECU)
Lambda	3,008MB	0.176292	5.75*
EC2	c5.large	0.085	8
	c5.18xlarge	3.06	278

Table 4.1: Cost of the AWS services used extracted from the AWS documentation. The EC2 instances used are on-demand. ECUs for Lambda are estimated based on the execution time.

In order to compare different instance types with different underlying architectures, AWS introduces the EC2 Compute Unit (ECU) [30]. As it can be seen in Table 4.1, only the ECU for the `c5.large` and the `c5.18xlarge` are defined in the official documentation [31] Thus, the ECU provided by the Lambda service are estimated by comparing the execution time of processing the same test image in an already measured machine (i.e. `c5.large`) and then applying equation 4.1 being *function_instance_time* the time used by a Lambda invocation to process one image.

$$\frac{c5.large_cpu_time\ \mathrm{s} * c5.large_ECU}{function_instance_time\ \mathrm{s}} \tag{4.1}$$

AWS Service	Type	Time to process 1,000 images (s)	Cost ($)	Machines needed to match Lambda time	Cost of the machines used to match Lambda time ($)
Lambda	3,008MB	142 (2.36 min.)	6.95	-	-
EC2	c5.large	102,500 (28.47 h.)	2.42	722	-
	c5.18xlarge	2,820 (47 min.)	2.40	20	2.41

Table 4.2: Summary of the costs of the image analysis. Services' cost extracted from Table 4.1, adopting per-minute billing.

Table 4.2 summarizes the cost calculations for the cost analysis. The equation used to calculate the cost of different EC2 instances (represented in the 4th column (Cost ($)) of Table 4.2 is the following:

$$instance_time \text{ s} * \left\lceil \frac{instance_cost}{3,600} \right\rceil \text{\$/s} \tag{4.2}$$

Also, Table 4.2 shows the number of concurrent instances needed to match the lambda execution time and its respective costs. To calculate the total cost presented in the last column, first it is calculated the number of machines needed (N), dividing the total execution time of the application in each instance by the time used by Lambda and rounding up. Then, with the number of machines (N) and using equation 4.3, it can be calculated the total cost of a multi-instance execution.

$$N * function_instance_time \text{ s} * \left\lceil \frac{ec2_instance_cost}{3,600} \right\rceil \text{\$/s} \tag{4.3}$$

In Lambda, the average execution time for the complete execution is 142 seconds. An invocation is performed for every image to analyze, so SCAR concurrently instantiates 1,000 Lambda functions. So, when using Lambda functions the time to process 1 or 1,000 images is the same and the total cost of the execution is, applying equation 4.2 and multiplying by 1,000 function instances, 6.95$.

Processing the images with the `c5.large` instance takes 102,500 seconds (i.e. almost 29 hours) with a cost of 2.42$. Thus to match the Lambda execution time, we would require 722 `c5.large` instances working concurrently. The cost of deploying such number of machines is not calculated because 722 EC2 instances surpasses by far the default maximum number of machines allowed

by the provider for each zone (i.e. 20). On the other hand, the `c5.18xlarge` takes 2,820 seconds (aprox. 47 min) with a cost of 2.40$. So, if we wanted to process the same number of images in the same time as the Lambda service, we would need 20 `c5.18xlarge` machines and it would cost us 2.41$.

To finish this cost analysis, two important aspects should be emphasized. First, the AWS EC2 service sets a default soft limit in the number of concurrent instances running at the same time at 20, that can be increased if requested to the provider. Second, and the most important, launching several EC2 instances at the same time to execute a high number of jobs inside them requires, in addition, some type of orchestration service. To be able to deploy all the instances concurrently we would need a service like AWS Auto Scaling [13], and also a job scheduler (e.g. SLURM [167], Torque [49], HTCondor [187], etc.) to ensure the job execution and tracking. These extra software layers complicate the multi-instance execution, in contrast with the easy parallelism that the Lambda services offer in combination with the the SCAR framework and the programming model presented.

Considering the cost analysis done, it can be outlined that the ease of use and the reduced execution times of the Lambda platform implies a price increment. The AWS Lambda service is more expensive than the EC2 instances, but on the other hand, it is easier to configure and launch, specially when used in combination with SCAR. Furthermore, cheaper solutions usually involve increasing the complexity of the execution, since orchestration tools would be needed to manage the execution. Thus, in cost terms, better solutions than AWS Lambda exist, but featuring important drawbacks that can make them unfeasible to the user with little or no experience in infrastructure deployment, which in fact, are the target users of the programming model.

4.7 Conclusions

This chapter has introduced a programming model to create highly-parallel event-driven serverless applications. The execution environment for this applications was provided by containers created out of customized Docker images.

The ability to run code in response to events and the large-scale elasticity provided by the underlying serverless platform opens new avenues for efficient High Throughput Computing tasks. Furthermore, the programming model allows to abstract away many implementation details typically required on computing frameworks. Moreover a cost analysis was carried out comparing

the serverless programming model presented and more common cloud computing architectures. Although the cost analysis revealed that running a serverless architecture could be costlier than deploying EC2 machines, the savings in configuration and execution time in combination with the pay-per-use model offered by AWS Lambda make the serverless architectures a good option to deploy applications that have to deal with a high amount of short lived tasks. In addition, several limitations identified in the previous chapter, such as the amount of memory allocated to each function invocation and, the most limiting one, the maximum execution time, were tackled by the presented programming model and can be overcome thanks to the combination of different AWS services.

Also, it is important to remark that the proposed programming model enables to deploy complex applications and can automatically provide an HTTP endpoint to trigger its execution where the cost is linearly dependent on the amount of requests to the endpoint and the resources consumed. Exposing a highly-available highly-scalable cost-effective endpoint for a generic application on a cloud platform paves the way for the adoption of serverless computing for the execution of complex scientific applications, even data-oriented workflows.

However, users do not always have access to public cloud infrastructures or depending on the restrictions of their projects, they are not allowed to use such infrastructures. That is why the next chapter focuses on the adaptation of the presented programming model to an on-premises functions-as-a-service platform. We will show how such model can be used in less restricted environments offered by on-premises infrastructures, thus allowing the users to create serverless workloads in their own clusters.

Chapter 5

Open-source Serverless Computing for Data-Processing Applications

This chapter presents the Open Source Serverless Computing for Data-Processing Applications (OSCAR) platform. OSCAR, in combination with the programming model presented in chapter 4, allows to define event-driven file-processing on-premises serverless architectures. Section 5.1 presents the open-source components integrated in the OSCAR platform. Section 5.2 explains the platform architecture, and section 5.3 tests the created platform with a use case for video processing.

The advent of open-source serverless computing frameworks has introduced the ability to bring the Functions as a Service (FaaS) paradigm for applications to be executed on-premises. As reviewed in section 2.4.3 there already exists a myriad of open source platforms that try to cover the FaaS on-premises platform gap. However most of those frameworks are focused on processing short-lived HTTP requests, like the public FaaS services that they mimic. Moreover, data-driven scientific applications can benefit from the ability to trigger scalable computation in response to incoming workloads of files to be processed. Thus, the challenge faced in this chapter is to provide an on-premises platform with the focus on the event-driven execution of compute-intensive applications, and to provide a seamless integration with

different storage back-ends as sources of events. To this end, this chapter introduces an open-source framework to achieve on-premises serverless computing for event-driven data processing applications that features:

- The automated provisioning of an elastic Kubernetes cluster that can grow and shrink, in terms of the number of nodes, on multi-Clouds.
- The automated deployment of a FaaS framework together with a data storage back-end that triggers events upon file uploads.
- A service that provides a REST API to orchestrate the creation of such functions.
- A graphical user interface that provides a unified entry point to interact with the aforementioned services.

Implemented together, these features offer a framework capable of deploying computing platforms for users to create highly-parallel event-driven file-processing serverless applications. The user can define such applications on customized runtime environments provided by Docker containers that will run on an elastic Kubernetes cluster.

5.1 Platform Components

Some of open-source projects reviewed in section 2.4.3 have a strong community behind them and keep adding new features each month. However there is not a unique project that offers all the features that are requested to tackle the proposed scientific challenge. So, when OSCAR was designed, it was decided to integrate different features of these open-source solutions to provide a platform that can cover the requirements needed to overcome the proposed challenge.

Figure 5.1 provides a high-level overview of the open-source components integrated in the OSCAR platform. The integrated components can be distributed in three main categories: 1) the software used to provide automated deployment and elasticity to the Kubernetes orchestrator; 2) the Kubernetes orchestrator itself; 3) the software used to provide a serverless environment on top of the Kubernetes orchestrator. The following sections explain these open-source software were integrated with Kubernetes.

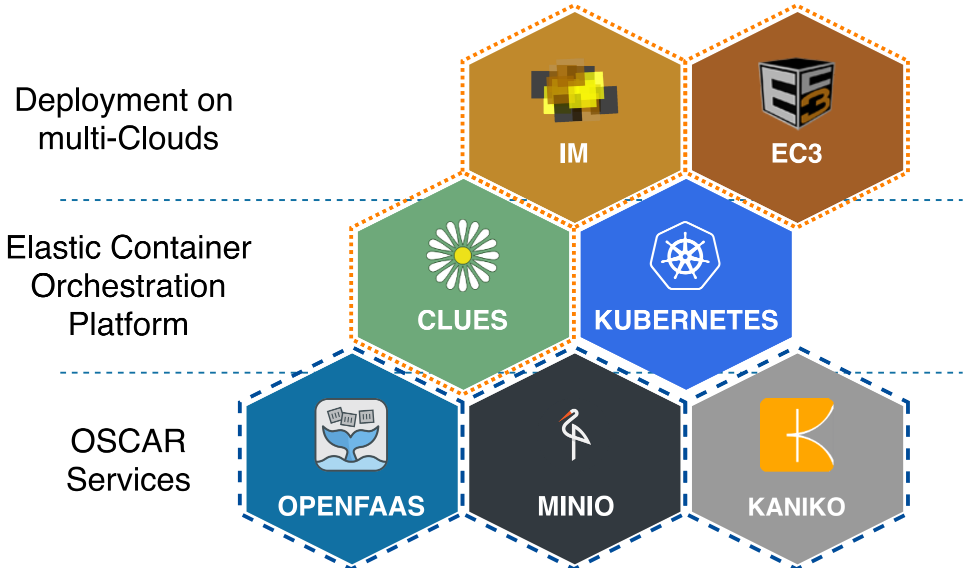

Figure 5.1: Main components used in the OSCAR architecture. The orange dotted line marks the components that allow elasticity at the level of virtual machines (i.e. powering on and off nodes). The blue dashed line delimits the components that provide the functions as a service and the event programming model.

5.1.1 Automated Kubernetes Cluster Deployment and Elasticity

As explained in section 2.2 Kubernetes has become the *de facto* solution for users that want to manage container microservices on different clouds. Among other features, Kubernetes is able to horizontally autoscale containers thanks to the Horizontal Pod Autoscaler [109]. With the Vertical Pod autoscaler [110], Kubernetes can down-scale pods that are over-requesting resources, and up-scale pods that are under-requesting resources based on their usage over time, and finally, thanks to the Cluster Autoscaler [111], Kubernetes is also capable of resizing the number of nodes in a given node pool, based on the demands of the received workloads. In conclusion, Kubernetes is a powerful platform with many different features that supports a multitude of different use cases. However these adaptability comes with a cost and the platform suffers from a steep learning curve and correctly configuring and adapting all the Kubernetes services required to provide an automatically managed FaaS platform can be a difficult task by the scientific users with no experience in cluster management. Therefore, in order to

facilitate the configuration, deployment and management of the elastic Kubernetes cluster and all its related services the OSCAR platform relies on the following tools:

- *Infrastructure Manager (IM)* [38], an open-source tool to describe complex application architectures using high-level declarative languages such as Resource Application Description Language (RADL) [89] and the standard specification Topology and Orchestration Specification for Cloud Applications (TOSCA) [140] in order to deploy them on multiple back-ends such as public Clouds (e.g. AWS, Microsoft Azure, Google Cloud Platform) and on-premises cloud management platforms (e.g. OpenNebula, OpenStack). The IM is used to support automated multi-cloud deployments.
- *CLUster Elasticity System (CLUES)* [2], an open-source modular elasticity system that supports a wide variety of plugins in order to introduce elasticity capabilities for cluster-based computing. Many plugins are supported in order to introduce horizontal elasticity for different types of clusters: i) based on an Local Resource Management System (LRMS), supporting SLURM and PBS/Torque; ii) based on container orchestration platforms, supporting Apache Mesos, Kubernetes and Nomad, and iii) based on HTC, supporting HTCondor. Although nowadays Kubernetes relies on the Cluster Autoscaler tool to offer a native solution to the horizontal node autoscaling issue, when this architecture was designed this component did not exist. So, thanks to CLUES, we can provide horizontal node elasticity, provisioning and terminating nodes when needed and additionally provide support for many different Cloud back-ends since it is integrated with the IM.
- *Elastic Cloud Computing Cluster (EC3)* [39, 40], an open-source tool to deploy through the IM virtual elastic compute clusters on multi-clouds that can scale in and out in terms of the number of nodes according to certain elasticity rules defined in the corresponding CLUES plugin. EC3 allows us to offer a tool that combines the benefits of IM and CLUES in a simple CLI. Thanks to EC3, we can develop a RADL recipe with the configuration of the complete cluster to be deployed, which allows us to have a unique point of maintenance when dealing with the cluster configuration. A complete EC3 recipe for the OSCAR platform can be found in Appendix B.

5.1.2 FaaS services in Kubernetes

After introducing the tools used to manage the low level elasticity (at the node level), this section presents the open-source software developments integrated with OSCAR to offer the users FaaS functionality for event-driven file-processing applications.

- *OpenFaaS* [143]: is a framework for building serverless functions with Docker and Kubernetes. OpenFaaS was analyzed as one of the open-source FaaS platforms available for on-premises infrastructures in section 2.4.3. Although this framework presents issues when executing long-running jobs, it is designed in a modular and extensible manner, which allowed us to modify and adjust the required components to the requirements of compute-intensive jobs. In OSCAR, OpenFaaS is going to be the system used to manage the function creation, deletion and invocation.

- *MinIO* [137]: is an object storage server that features an Amazon S3 compatible API. MinIO is used to provide data persistence for the input and output data of the functions in OSCAR. In addition, by taking advantage of its eventing features MinIO, in combination with OpenFaaS, allows to support data-driven function workflows with a similar approach to what was done with AWS S3 and AWS Lambda (see section 4.2 for more information about the execution workflows available).

- *Kaniko* [108]: a tool to build container images from a Dockerfile, inside a Kubernetes cluster. The main feature of Kaniko is that it does not depend on a Docker daemon, thus building Docker container images in userspace. Inside the OSCAR platform, Kaniko is used to manage the container modifications needed for the infrastructure (including the faas-supervisor binary) without user interaction.

In addition, other services need to be deployed to support the functionality of all the integrated systems. A distributed Docker registry is used to store the automatically created container images that represent the functions deployed by the user. Also, a Network File System (NFS) is used to provide shared folders among the cluster frontend and working nodes. NFS is used to provide persistent volumes among all the nodes that comprise the Kubernetes cluster, and are used by the MinIO, Docker registry and Kaniko services.

5.2 OSCAR architecture

In the previous sections we have presented deployments that already existed. In this section we are going to show how these components were integrated (including the modifications to adapt some of them) and also the new software designed to offer the user an easy to use platform for deploying FaaS workflows in on-premises Cloud platforms.

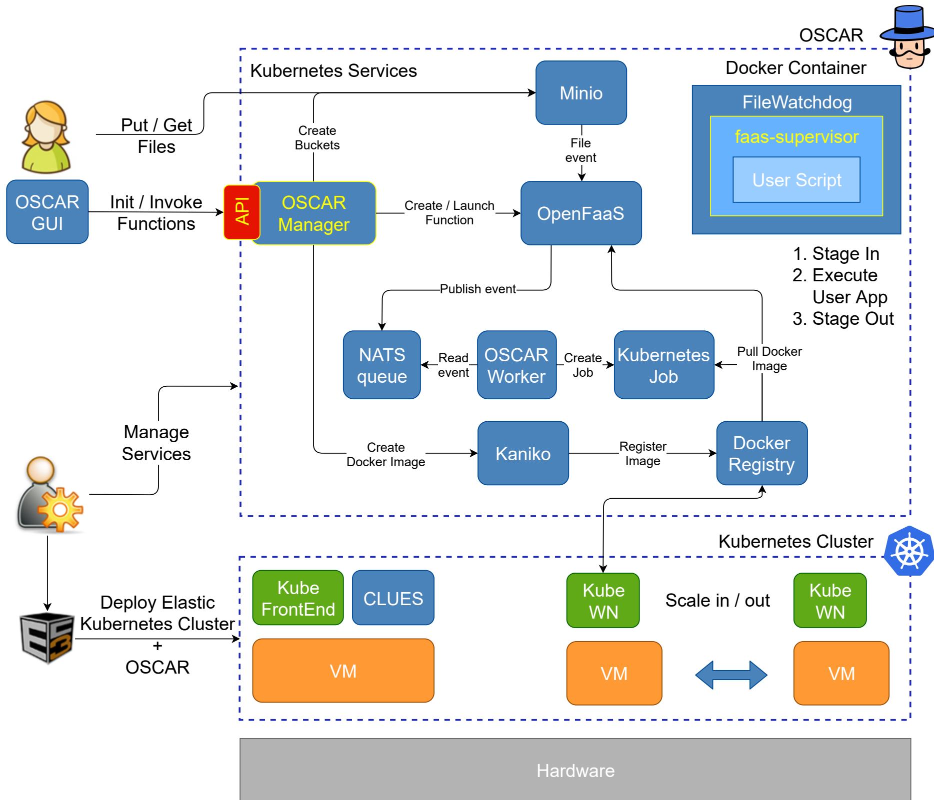

Figure 5.2: Architectural approach for supporting container-based file-processing applications on serverless platforms. The yellow font remarks the components developed by the doctoral candidate.

Figure 5.2 provides an overview of the interaction among the services deployed in the elastic Kubernetes cluster dynamically provisioned by the OSCAR framework. The figure is divided in two main components, the underlying Kubernetes cluster in which all the services are deployed and that

provides automatic node elasticity and the OSCAR platform that integrates all the high-level services that provide the FaaS functionality. Also, as it can be seen in the figure, the user does not need to worry about cluster management, once a system administrator has deployed the OSCAR platform, which can be accessed through the OSCAR GUI providing an easy usage of the FaaS infrastructure.

5.2.1 OSCAR Manager

OSCAR manager is the component in charge of orchestrating all the other high-level services present in the Kubernetes cluster. It offers a REST API that allows the user to initialize, invoke, and delete functions. The process to create a function is completely transparent to the user and is comprised of the following steps:

- Through the web interface, the user generates a request to create a function that is received by the OSCAR Manager service. The basic fields needed to create a function are, the function's name, the Docker container image to use, and the script to be executed inside the function (which is the function's code).
- Using as base image the user's image, and through Kaniko, a new Docker image is created with the faas-supervisor binary injected. The faas-supervisor is in charge of managing the input data required and the output data generated by the function execution. Once the Kaniko build is finished, the image is stored in the shared Docker registry deployed in Kubernetes.
- Based on the function's name, the OSCAR manager creates the required input and output buckets in the MinIO service. These buckets and all its internal files are accessible from the OSCAR user interface, thus easing the data uploading and downloading from the platform.
- Finally, the function is created in OpenFaaS. OpenFaaS will use as image for the function the new image created by Kaniko and retrieves it from the Docker registry. In addition, the input bucket created in MinIO is automatically linked with the corresponding function. Thus, when an file is uploaded to the bucket, an event with the file information is generated and passed to the linked function.

Besides creating the function and buckets, OSCAR manager is also in charge of processing the MinIO events. This feature is related with the MinIO configuration, because each time a new webhook is added to the MinIO configuration file (i.e. each time a new function is created), the MinIO service needs to be reconfigured, what takes around 90 seconds and completely blocks the usage of the storage service. So, to avoid changing the configuration file each time, a webhook is created, but with the OSCAR manager instead of the new created functions. Then, each time an event is generated the manager is in charge of redirecting it to the required function.

5.2.2 OpenFaaS and the OSCAR worker

OpenFaaS is designed to process short-lived requests and, therefore, attempting to execute several long-running jobs at the same time typically ends up collapsing the Kubernetes cluster due to the lack of resources for all the processes. To be able to support long running jobs a new service was developed to replace the *nats-queue-worker* deployed by default by OpenFaaS, i.e. OSCAR worker in Figure 5.2.

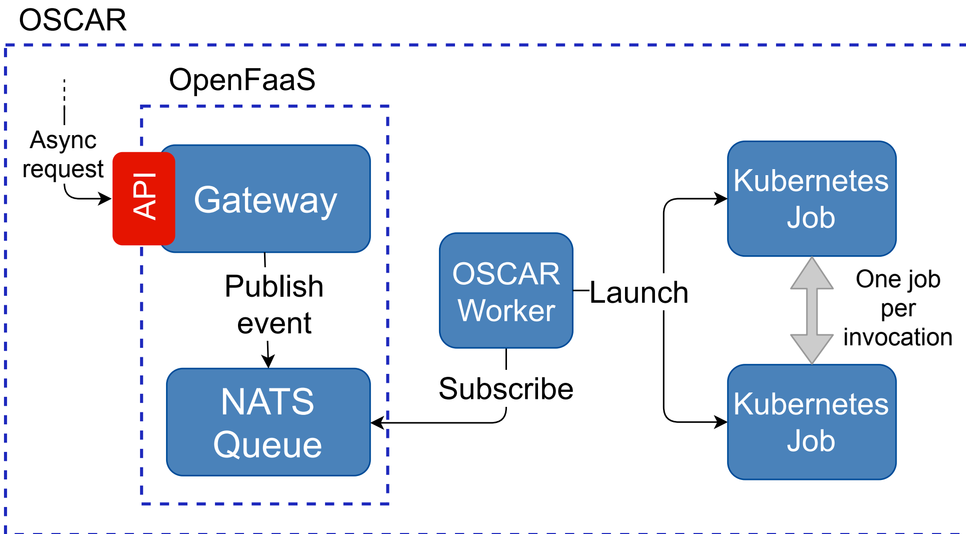

Figure 5.3: OSCAR worker used to support long-running jobs in combination with OpenFaaS[1].

[1] Schema extracted from: `https://github.com/grycap/oscar-worker`

As it can be seen in Figure 5.3, the OSCAR Worker transforms asynchronous requests sent to OpenFaaS into Kubernetes jobs. Thus, the steps taken when executing a function are as follows. First, when the user uploads a file to a MinIO bucket, an event is triggered, processed by the OSCAR manager, and sent to OpenFaaS as an asynchronous request which is stored in the NATS queue provided by OpenFaaS. Second, the OSCAR worker (that is subscribed to the NATS queue) reads the asynchronous request and then creates and submits a new job to Kubernetes. Submitting a job to Kubernetes allows OSCAR to delegate the resource management to both Kubernetes and the CLUES elasticity system. CLUES is able to detect when the Kubernetes cluster needs more resources (based on CPU and RAM memory usage) and provisions new nodes accordingly. Likewise, if CLUES detects that Kubernetes has spare nodes that are no longer needed, it terminates them, thus freeing the resources.

5.2.3 The FaaS supervisor

As stated before, one of the goals of this thesis is to allow the users to execute FaaS workflows in public and private clouds. We also saw that during the development of SCAR and the high throughput programming model, a library was developed to manage the data stage in and out when dealing with FaaS functions. Thus, when we approached the development of a library to manage the data in a FaaS on-premises platform, we decided to adapt and generalize the SCAR supervisor and transform it into the 'faas-supervisor' library. Depending on the underlying platform used, the faas-supervisor is used as a Python library (in AWS Lambda) or as a binary (in OpenFaaS), but the functionality offered is the same, it automatically provides the mechanisms to download the required data inside the function environment and to upload the generated data from the function execution.

The faas-supervisor has been designed to process one file for each event received, and its generic execution steps (common to all providers) once the function is initialized are as as follows:

- Create the required temporal folders: to provide users with a generic way of dealing with the input and output folders, at the beginning of the supervisor execution, a pair of random folders (for input and output) are created and then the environment variables $\$TMP_INPUT_DIR$ and $\$TMP_OUTPUT_DIR$ are set. Thus, the user's script will always find the downloaded input files in the specified input folder, and the files

created by the script in the output folder will be automatically uploaded to the linked storage service.

- Parse the received event: currently the supervisor is able to differentiate between events generated by Amazon API Gateway, Amazon S3, MinIO, and Onedata [142] (Onedata is a high-performance data management solution that offers unified data access across globally distributed environments and multiple types of underlying storage). In addition, the supervisor provides a fail-safe mechanism to save an event that is not recognized so the user can find it stored as a file called *event_file* in the input folder. If an API Gateway event is detected, the body of the HTTP request in analyzed and its content stored consequently.

- Read the storage variables: currently the supervisor supports the following storage providers: MinIO, Onedata, AWS S3 and local storage. The storage authentication variables are defined as environment variables inside the functions' environment. At the beginning of the function's execution, the supervisor looks for the environment variables that follow the pattern $STORAGE_AUTH_\$1_\$2_\$3$, where $1 is the storage provider (MinIO, S3, Onedata), $2 is the variable type (user, pass, token, space, host), and $3 is the automatically generated storage id. If there are variables that follow those rules, the corresponding authentication objects are created.

- Create the supervisor: finally, once the event and the storage authorization variables are parsed, the required supervisor is created. The environment is checked and depending on which platform it is deployed, the supervisor behaves accordingly (executing an script in OpenFaaS, or running udocker in AWS Lambda).

In addition, the development of the supervisor library has been detached from the developments of SCAR and OSCAR and can be checked out in its own Github repository [87]. This, in combination with its modular design allows other developers to easily create new data storage plugins, and event managers.

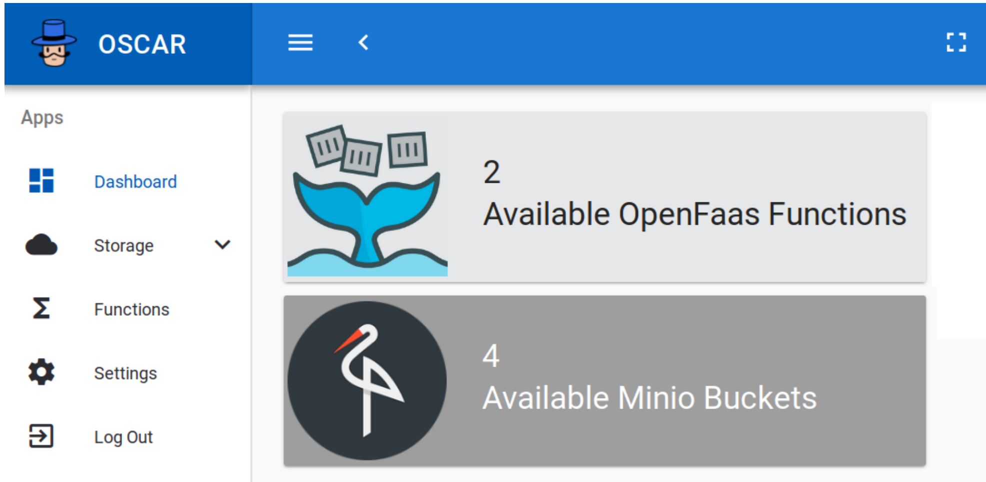

Figure 5.4: OSCAR GUI. Main screen with the summary of the deployed resources.

5.2.4 The OSCAR GUI

To provide the users with a single access and management point to the OSCAR platform, it was developed a Graphical User Interface (GUI) that connects the different services inside the Kubernetes cluster.

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

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

Figure 5.5 depicts the Functions tab where you can create, edit or delete functions and Figure 5.6 shows the Storage tab where the information of the buckets is shown, as well as the stored files. In the Storage tab users can

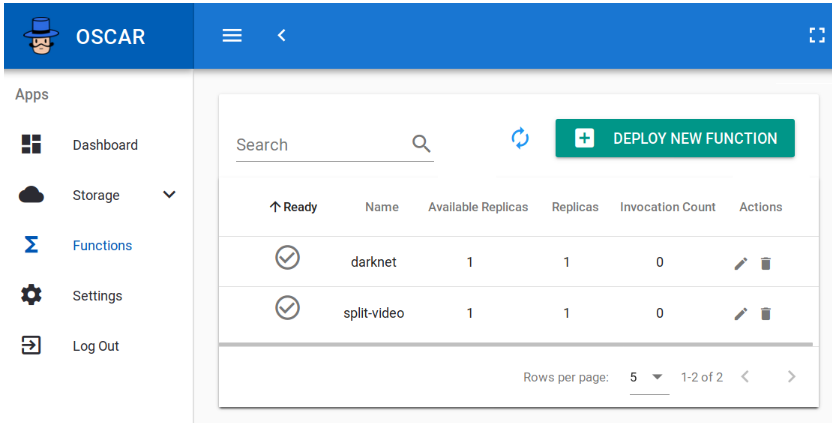

Figure 5.5: OSCAR GUI. The functions tab shows the user the functions created and the function status.

upload the files to be processed, remove them from the buckets or download the output files generated by a function.

5.3 Case study: Video Processing Service in On-premises Infrastructure

In order to assess the OSCAR framework we deployed a serverless video processing service in an on-premises OpenNebula-based cloud. The service comprises two functions linked by means of an storage bucket, so when the first function finishes, the second function is triggered automatically. The goal of this service is to apply object recognition to the frames of the video uploaded by the user as input. Figure 5.7 shows the workflow of the architecture proposed. The files needed to reproduce the case study are open-source and available in GitHub[2].

The video processing function uses the *ffmpeg* library to extract the keyframes from the input video. To be able to create different workloads, the keyframe extraction rate has been changed to generate different amounts of images. The image processing function uses the *darknet* framework in combination with the You Only Look Once (YOLO) v3 library [161] to detect the objects in the image. All the libraries and frameworks have been compiled to support CPU-based executions.

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

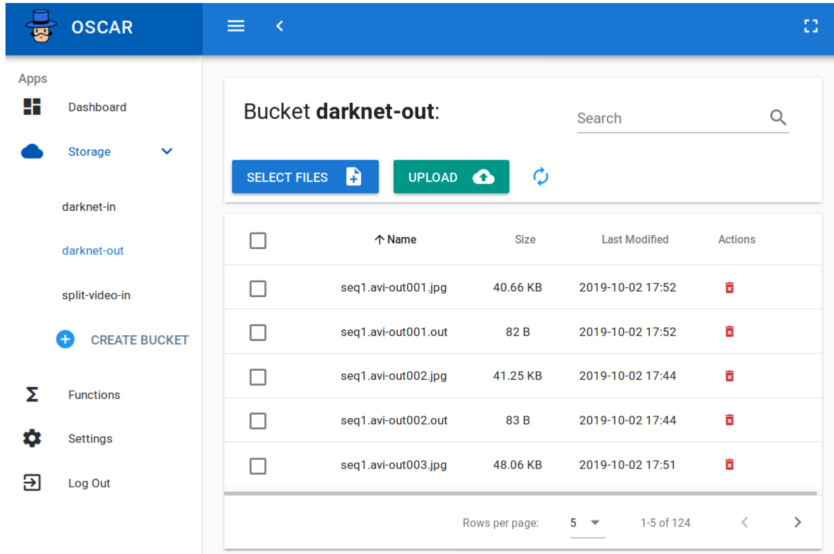

Figure 5.6: OSCAR GUI. The storage tab shows the user the MinIO buckets automatically created when the function is deployed in the infrastructure and the stored files inside those buckets.

The following steps were taken during the experiment execution:

1. Using the OSCAR GUI the user creates the video processing and the image processing functions. By defining the output bucket of the video processing function as the input bucket of the image processing function the user is creating the workflow that is going to be triggered when a file is uploaded into the input bucket. Bear in mind that the creation of the required containers, the container registration in the internal Docker registry, the creation of the needed MinIO buckets, and the creation of the OpenFaaS function is automatically performed without any user interaction.

2. Through the OSCAR GUI, the user uploads to the input bucket of the video function the video to process. After this step the user interaction is not required anymore until the retrieval of the output data.

3. After the video upload finishes, MinIO creates an event that is pushed to OpenFaaS which stores the event received in the NATS queue. Afterwards, the OSCAR worker reads the NATS queue and launches the function as a Kubernetes job. During the function execution, the

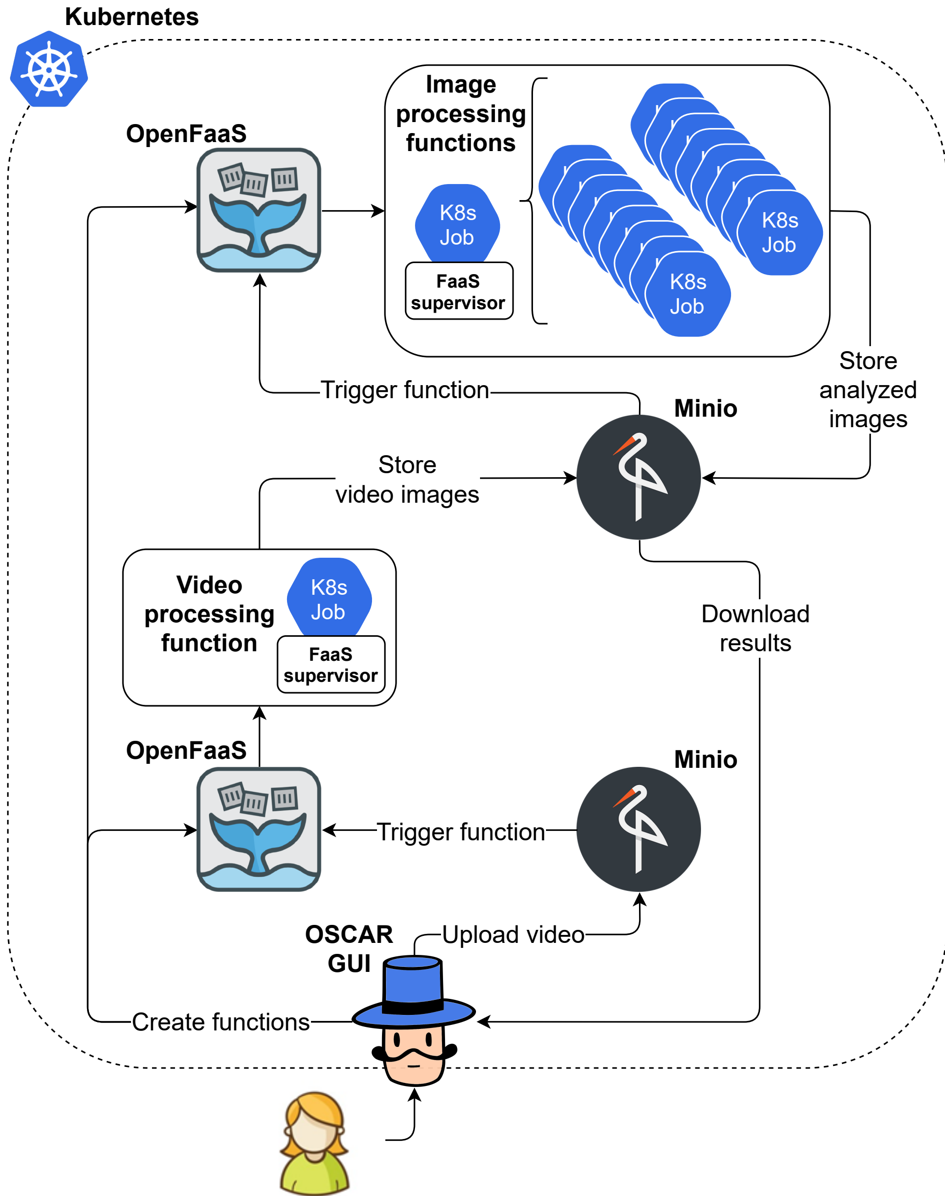

Figure 5.7: Simplified workflow of the video processing service. Two functions are used to process the video. First, a function to extract the video keyframes, and second, a function to analyze the generated keyframes. the function with the image processing functionality is triggered automatically for each keyframe created by the video processing function.

FaaS supervisor library that acts as a wrapper of the function instance retrieves the video from the MinIO bucket and stores it inside the function's ephemeral storage space, in the input folder automatically created at the beginning of the supervisor execution. Then, the user script is executed and the generated output is stored in the specified output folder. As the last step of the video function, the OSCAR supervisor uploads all the files present in the output folder to the output bucket defined, thus triggering the image processing function.

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

5. The last step involves the user downloading the files generated by all the executed functions.

As summary, this use case demonstrated the feasibility of using the OSCAR platform to manage FaaS functions in on-premises infrastructures. The platform allows the users to upload files to an object storage system and automatically triggers the processing of the files in an elastic manner, thus solving the challenge of freeing the users from the infrastructure management when dealing with on-premises environments.

5.3.1 Results

To test the scalability and reliability of the OSCAR platform, three different workloads were used, based on the number of images extracted from each video. For the first workload we extracted 10 images, 100 for the second workload and 1,000 for the third. These workloads involved the invocation of 11, 101 and 1,001 functions respectively in the OSCAR cluster, one for the video processing and the remaining for the image processing.

The physical infrastructure used in all the experiments is composed by two type of nodes. The first node type has two Intel(R) Xeon(R) CPU E5-2683 v3 2.00GHz (14 cores) processors, 64 GB of memory RAM, 240 GB of Solid

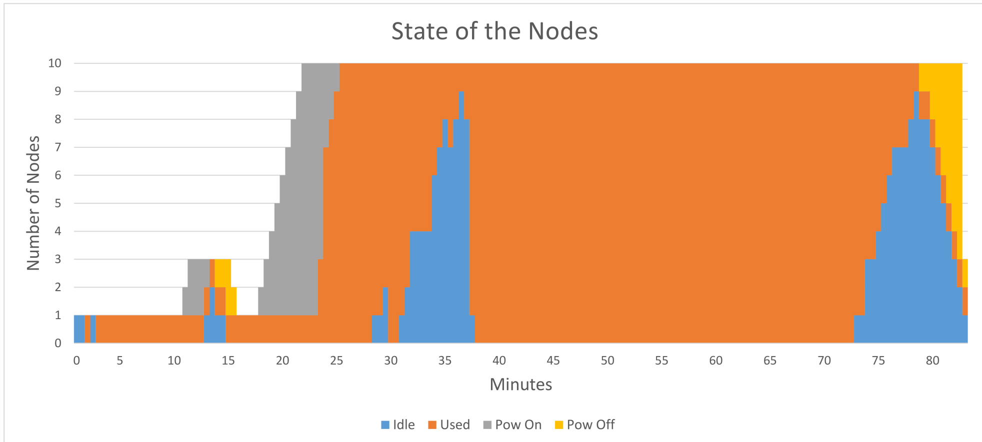

Figure 5.8: State of the nodes during the execution of the three different workloads.

State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The second node type has two Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz (14 cores) processors, 128 GB of memory RAM, 250 GB of Solid State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The Storage Area Network is a Dell Equallogic PS4210 with 16 TB availables. Finally, the hardware is managed by the OpenNebula Cloud Management Framework and the KVM hypervisor.

The specifications for the virtual cluster used in the case study are the following: the front-end has 8 virtual CPUs and 16 GiB of RAM. The working nodes have 4 virtual CPUs and 8 GiB of RAM. The specifications of the nodes were selected to simulate two of the most common instances used to process compute-intensive workloads in Amazon Web Services [56]. The front-end is equivalent to a c5.2xlarge and the nodes are equivalent to a c5.xlarge. The complete virtual cluster is composed by 1 front-end and a maximum of 10 nodes which will be powered on on-demand. The cluster size of 10 working nodes was selected because it allows to demonstrate the scaling capabilities of the deployed platform, and at the same time, it keeps the load of the infrastructure controlled. All the machines used are virtual machines deployed in the OpenNebula-based on-premises cloud described above.

Figure 5.8 shows the state of the virtual nodes during the execution of the three proposed workloads. The colors of the areas represent the following: the blue area represents the nodes that are idle (i.e. waiting for jobs); the dark orange area represents the nodes that are busy processing Kubernetes jobs;

the grey area shows the number of nodes that are powering on and the gold area shows the number of nodes that are powering off. The data in the graph is stacked, thus the areas that have no color represent nodes that are powered off and are not consuming resources in the infrastructure.

In order to immediately process small workloads, the deployed cluster always has one working node available. This is represented by the orange area along the first 10 minutes in Figure 5.8. Afterwards, the first workload starts (i.e. process a video and ten extracted images). In the minute 10, the video is processed by the available node and the images to analyze are generated. CLUES realizes that it does not have the required computing resources to process the new function invocations and provisions additional nodes to process the incoming workload. This process can be seen in the first grey area in minute 12. The nodes in *power on* state (i.e. being deployed and provisioned) take 3 minutes to be ready which is enough time for the already deployed working node to process the 10 jobs in the queue. Therefore, the new nodes that are deployed are in idle state and after a couple of minutes are powered off to save resources (this is represented by the gold area in minutes 14-16).

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

With the third workload it is tested the reliability of the infrastructure under a high load. As stated at the beginning of the section, 1,001 function invocations are launched and processed. In Figure 5.8 it can be seen that this is carried out between the minutes 37 and 78. In minute 37 all the working nodes that are idle receive new function invocations to process and the cluster continues processing them until minute 73 where the working nodes start to be idle. After being idle for 5 minutes and not receiving new function invocations, CLUES starts to terminate nodes until only one working node is left.

Figure 5.9 shows the RAM memory and CPU usage of the nodes along the execution of the three workloads tested. The graph represent the stacked resources for each node and the green line represents the total amount of provisioned resources in the cluster infrastructure with two different Y axis

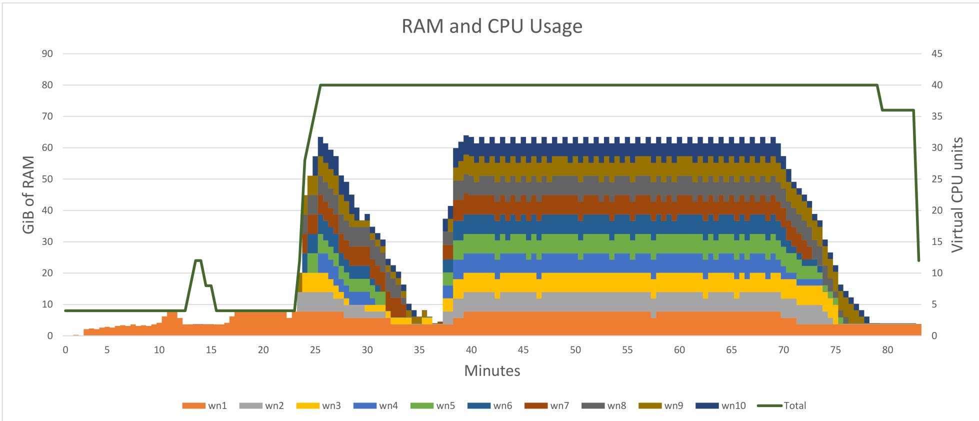

Figure 5.9: RAM and CPU usage during the execution of the three different workloads (11, 101 and 1,001 functions executed respectively).

(GiB of RAM, and number of vCPUs). This can be represented because the CPU and memory resources are linearly related and grow accordingly. Since each function invocation has the same resource requirements and all the functions use the maximum resources available in each execution ,the graphs of RAM and CPU usage can be combined. It is also important to know that no more than three functions per node could be deployed due to the image function requirements (i.e. 1 CPU and 2 GiB of RAM) and that Kubernetes also needs to deploy their own pods to control each node (those pods also use CPU and RAM resources). This behavior caused that several GiB of RAM and CPUs were unused because the unused space in each node was less than the minimum space required by the functions and that is represented by the white area under the green line which is the total amount of resources.

As in Figure 5.8, Figure 5.9 clearly depicts the execution of the three workloads. From minutes 10 to 12 there is a peak in resources consumption in the available working node. After the cluster finishes deploying new nodes this peak has disappeared (i.e. the execution of the functions has finished) and the new reserved resources (the working nodes) are freed again (terminated). The second workload starts in minute 17 and has its maximum peak of RAM and CPU consumption in minute 26 after all the nodes have been deployed. As the functions are processed, the usage of the resources of the working nodes decreases but the nodes are not terminated, thus not releasing the reserved infrastructure resources. The third workload starts in minute 37, when the maximum number of functions per node are deployed

again and this behavior continues until the 1,000 images are processed in the minute 78. Once CLUES detects that the working nodes are idle for 5 minutes, it terminates them. The green line (total amount of resources reserved) at the end of the graph going down represents this release of resources.

As a summary of the results, the framework was able to process three different workloads, executing 11, 101 and 1,001 functions. The first 11 function invocations were processed in 3 minutes and no extra nodes were needed (two new nodes were powered on but were never used). Processing the 101 function invocations made the cluster reach its top performance as seen in Figure 5.9 and it took 19 minutes to finish (including the deployment time of nodes which is 3 minutes). The third workload, processing 1,001 functions, also fills all the available processing slots of the infrastructure's nodes. The third workload finishes in 41 minutes. This time could be improved by deploying a bigger cluster (e.g. 20 nodes instead of 10). Thanks to the elasticity of the cluster, these nodes would only be used when a high amount of function invocations are needed to be processed, being powered off the remaining time. To deploy a bigger cluster, the user only has to change the maximum number of nodes available when deploying a new cluster. The OSCAR framework will manage everything else to use those new resources.

5.4 Conclusions

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

Users are provided with an open-source platform offered via a convenient web interface that simplifies the creation and execution of the functions. The users just need to upload their files in order to trigger the concurrent

execution of the application. The application will process the uploaded files and leave the output data files in the corresponding folder for the users to retrieve them. Being able to interact with a computing platform without requiring the definition of jobs, and by means of a web browser represents a step forward towards simplifying application execution for data-processing applications.

In the case study we saw that due to the resource requirements of the Kubernetes infrastructure, the RAM memory and CPU resources of the working nodes could not be completely used. Further work in the infrastructure refining the requirements and the behavior of the required pods could lead to a better usage of the cluster resources and thus to a higher throughput when processing functions.

As final remarks, the aforementioned components are currently being used in production in the European Grid Infrastructure (EGI) Federated Cloud [63], a federated IaaS Cloud, composed of academic private clouds and virtualised resources and built around open standards, whose development is driven by the requirements of the scientific communities [57]. This integration allows scientists to self-deploy their virtual infrastructures on a federated Cloud in order to tackle challenging computational problems. OSCAR can be deployed through the EC3 Portal available in the EGI Applications on Demand[3] service.

[3] EGI Applications on Demand: https://www.egi.eu/services/applications-on-demand/

Chapter 6

Use cases

This chapter introduces different use cases where the frameworks introduced in previous chapters are tested. In addition, several use cases created by the user community are presented. Finally, we will describe the most relevant scientific contributions carried out during this thesis and we will talk about user acceptance and the projects in which the presented developments are being used.

Previous chapters have covered the developments of the tools and platforms created during this thesis. To test such tools, a couple of use cases were presented, a medical image analysis service (in section 4.6), and a video processing service (in section 5.3). This chapter introduces additional use cases where the tools developed have been used. In addition, since the developments have been released under an open-source license, we will also show use cases created by the community. All the use cases presented in this chapter can be found in the public Github repositories of their respective projects (SCAR[1] and OSCAR[2]). Table 6.1, summarizes the use cases presented, what tools where used to create them, which cloud provider was used and the degree of participation on the development of each use case.

[1]https://github.com/grycap/scar
[2]https://github.com/grycap/oscar

Use Case	Tools	Cloud	Development
Adding programming languages support	SCAR (Lambda)	AWS	Created
Massive image processing service	SCAR (Lambda)	AWS	Created
Video processing service	SCAR (Lambda & Batch)	AWS	Created
Plant classification	OSCAR	On-premises	Participated
Multi-cloud workflow for video processing	OSCAR, SCAR (Batch)	On-premises, AWS	Community
Air pollution information service	OSCAR	On-premises	Community
Monetizing private algorithm workflow executions	SCAR (Lambda & Batch)	AWS	Community
GROMACS	SCAR (Batch)	AWS	Community

Table 6.1: Summary of the use cases presented.

6.1 Adding support to programming languages and software in AWS Lambda

One of the most straightforward applications of the SCAR framework is allowing the execution of programming languages that are currently not supported by the AWS Lambda. The supported languages tested and added by SCAR to AWS Lambda are *elixir*, *erlang*, *R*, and *Ruby*. In addition, any version of any language can be executed inside the Lambda service if you can containerize it (and assuming that the container created complies with the size restrictions), so you could run deprecated or newer versions of the languages already supported.

Other direct use of the SCAR framework is allowing the execution of software that have additional dependencies like external libraries or packages. Once the container is created, the user can define a simple script in bash to execute the program. The SCAR repository[3] includes several examples of these use cases: for example: a command line client to manage AWS resources (aws-cli); an open-source neural network framework (darknet); a tool to convert video and audio (ffmpeg); a tool for image manipulation

[3] https://github.com/grycap/scar/tree/master/examples

(imagemagick); a scientific application for bayesian inference of phylogeny (MrBayes); a tool to generate HTML 5 documentation from OpenAPI/Swagger 2.0 API specifications (spectacle) and, finally, a Python library that allows users to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently (Theano).

In conclusion, any software that can be containerized and complies with the size and memory restrictions applied by AWS Lambda can be executed. As an example of how to add from scratch unsupported software, the following section briefly describes how the support for the Elixir language was introduced by using SCAR.

Adding support to the Elixir language

To successfully execute an unsupported language inside the AWS Lambda environment, the user needs to define the container image that includes all the required language libraries and dependencies. Listing 6.1 shows the Dockerfile defined with the Elixir packages inside.

```
FROM grycap/erlang

# elixir expects utf8.
ENV ELIXIR_VERSION="v1.4.5" \
        LANG=C.UTF-8

RUN set -xe \
        && ELIXIR_DOWNLOAD_URL="https://github.com/elixir-lang/elixir/
            releases/download/${ELIXIR_VERSION}/Precompiled.zip" \
        && buildDeps=' \
                ca-certificates \
                curl \
                unzip \
        ' \
        && apt-get update \
        && apt-get install -y --no-install-recommends $buildDeps \
        && curl -fSL -o elixir-precompiled.zip $ELIXIR_DOWNLOAD_URL \
        && unzip -d /usr/local elixir-precompiled.zip \
        && rm elixir-precompiled.zip \
        && apt-get purge -y --auto-remove $buildDeps \
        && rm -rf /var/lib/apt/lists/*

CMD ["/bin/sh","/usr/local/bin/iex"]
```

Listing 6.1: Dockerfile used to create the elixir container.

Once the Docker image is built, the user can initialize the Lambda function using a SCAR configuration file. Listing 6.2 presents the simple SCAR

configuration file defined for this example. Using this configuration file we can create the function with the command: '*scar init -f scar-elixir.yaml*'. Listing 6.3 shows the output generated by the creation of the elixir function. As we can see, the *faas-supervisor* presented in the previous chapter is used.

```
functions:
  scar-elixir:
    image: grycap/elixir
```

Listing 6.2: YAML file used define the elixir function (scar-elixir.yaml).

```
Using existent 'faas-supervisor' layer
Creating function package
Function 'scar-elixir' successfully created.
Log group '/aws/lambda/scar-elixir' successfully created.
```

Listing 6.3: Ouput for the SCAR init command.

To finish, the user needs to define the script to be executed inside the already created Lambda function. Listing 6.4 shows a script to check if the elixir language is working properly. We can launch the Lambda function and the script execution, with the command: '*scar run -f scar-elixir.yaml -s elixir-hw.sh*'.

```
#! /bin/sh

#Elixir example from: https://github.com/philnash/elixir-examples/tree/
    master/hello-world

cd /tmp

export LANG=en_US.UTF-8
cat << EOF > hello-world.exs
IO.puts "Hello World from Elixir code!"
EOF

elixir hello-world.exs
```

Listing 6.4: Bash script executed inside the elixir container (elixir-hw.sh).

Listing 6.5 shows the output generated by the Lambda function execution. In the last line, we can see that the Elixir code that we defined in Listing 6.4 is executed successfully and the hello world message is printed.

```
Request Id: da6f291a-8bd4-439f-a9d1-4bb1cde1a588
Log Group Name: /aws/lambda/scar-elixir
Log Stream Name: 2019/09/08/[$LATEST]2a6d122827ea4a41a3bd03f128fe67da
Hello World from Elixir code!
```

Listing 6.5: Ouput for the SCAR run command.

Notice that we just demonstrated how to add a new programming language with a couple of scripts and the SCAR tool, thus, bringing the massive elasticity features of AWS Lambda to a community of users of a certain programming language (Elixir in this use case).

6.2 Massive image processing service

In this use case SCAR is used to deploy a serverless service that recognizes different patterns in images using Deep Learning techniques. The framework used for pattern recognition is Darknet [159], an open source neural network framework written in C, in combination with the YOLO [160] library. The Docker image used for this case study can be found in the *grycap/darknet* Docker Hub repository[4]. Darknet and the YOLO library are memory intensive applications, so the function created for this use case has 2,048MB of RAM.

Figure 6.1 describes the complete architecture and the data workflow designed to carry out this case study. We took advantage of the feature offered by the SCAR programming model to automatically perform Lambda invocations out of a set of files already available in a cloud storage service (Amazon S3, in this case). This feature allows the user to reuse an existing S3 bucket with data files in order to perform an analysis across all the files in that bucket, or folder inside the bucket. For each file read, one Lambda function is invoked, up to the 1,000 soft limit of concurrent Lambda invocations. Each Lambda invocation executes a predefined shell-script and processes exactly one file. Listing 6.6 shows the SCAR configuration file used to define the workflow presented in Figure 6.1.

```
functions:
  scar-darknet-s3:
    image: grycap/darknet
    memory: 2048
    init_script: yolo.sh
    s3:
      input_bucket: scar-darknet
```

Listing 6.6: SCAR configuration file usd to define the massive image processing service.

For this experiment we used: i) a synthetic dataset created out of 100 random images of animals and objects and scaled up to 1,000 images already stored in an AWS S3 bucket with a total size of almost 500MB; ii) the Docker image stored in Docker Hub which contains all the libraries and dependencies needed

[4] https://hub.docker.com/r/grycap/darknet

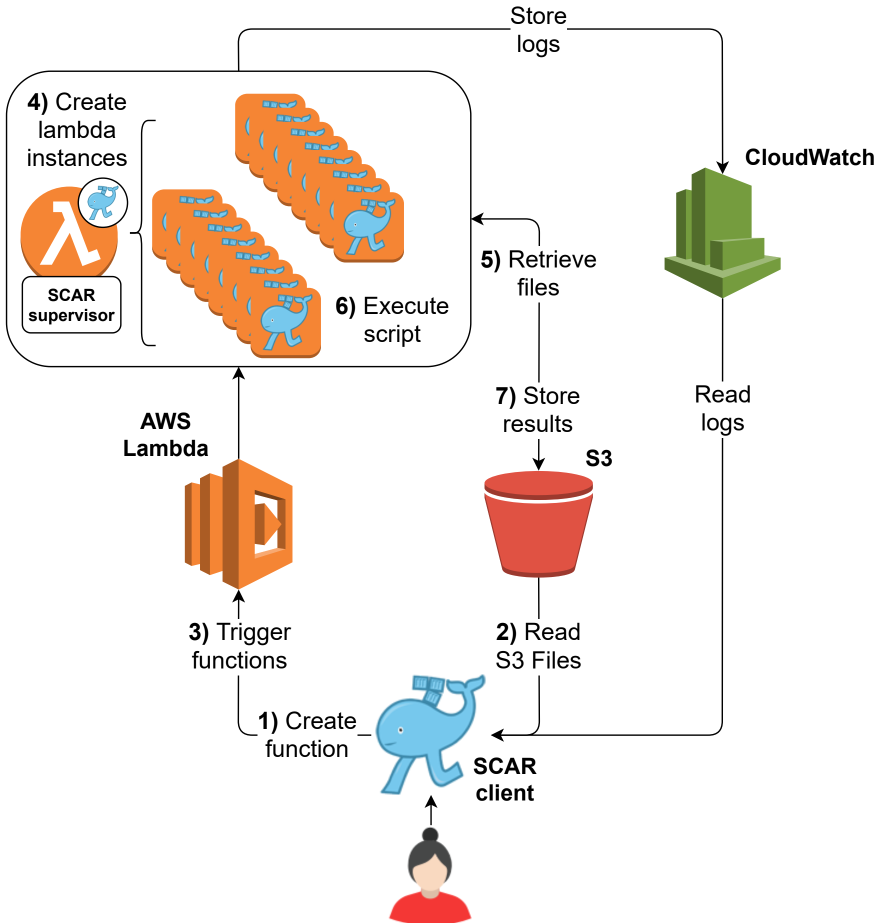

Figure 6.1: Workflow of the Massive Image Analysis Service. The user, through the SCAR client, 1) creates the Lambda function, and 2) reads the files needed to process. Then SCAR automatically 3) sends the events to AWS Lambda making 4) the service to invoke the function as many times as events received. 5) Then, the supervisor deployed inside each invocation retrieves the required files 6) and executes the user script. 7) To finish, the supervisor stores the output files generated by the lambda invocations at the defined output bucket.

to execute the Darknet software (see listing 6.7) and, iii) a shell-script executed inside the container, in charge of processing the input image using DarkNet and the YOLO library to obtain the output (see listing 6.8) .

```
FROM bitnami/minideb

COPY darknet.tar.gz /tmp/

RUN tar xvzf /tmp/darknet.tar.gz -C /opt/ \
  && rm /tmp/darknet.tar.gz

RUN apt-get update \
  && apt-get install -y --no-install-recommends wget ca-certificates \
  && wget https://pjreddie.com/media/files/yolo.weights -P /opt/darknet/
      \
  && apt-get remove -y wget ca-certificates \
  && apt-get autoremove -y \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
```

Listing 6.7: Dockerfile used for the creation of the massive image processing service environment.

```
#!/bin/bash

RESULT="$TMP_OUTPUT_DIR/result.out"
OUTPUT_IMAGE="$TMP_OUTPUT_DIR/image-result"

echo "SCRIPT: Analyzing file '$INPUT_FILE_PATH', saving the result in '
    $RESULT' and the output image in '$OUTPUT_IMAGE.png'"

cd /opt/darknet
./darknet detect cfg/yolo.cfg yolo.weights $INPUT_FILE_PATH -out
    $OUTPUT_IMAGE > $RESULT
```

Listing 6.8: Script used to launch the YOLO object detection library of the Darknet framework.

To simplify the creation of the execution scripts, the SCAR supervisor provides several environment variables that contain information about the input file received, and the temporal input and output folders created during the function invocation. In this case, it can be seen in listing 6.8 that we are using the *INPUT_FILE_PATH* and the *TMP_OUTPUT_DIR* environment variables to retrieve the input file and to store the generated output files respectively. After the creation of the script variables, the Darknet invocation command (i.e. *darknet detect ...*) receives an image as an input file and stores the results in two separate files, the *OUTPUT_IMAGE* which will be the image with the recognized objects, and the *RESULT* file which will contain the percentage of certainty for each recognized object. To finish, all the output files available in the folder *TMP_OUTPUT_DIR* are automatically uploaded by SCAR to the folder of the specified S3 bucket. The *echo* command present in listing 6.8 is

added to ease the traceability of possible errors, because the SCAR supervisor passes the container standard output to the AWS CloudWatch Logs service, so if the container execution fails we can check it after the functions finishes. A sample of the input and output images respectively provided and generated in this use case, are presented in figures 6.2 and 6.3 respectively. Also the output file generated after processing the image is shown in listing 6.9.

Figure 6.2: Test image passed to the Darknet framework.

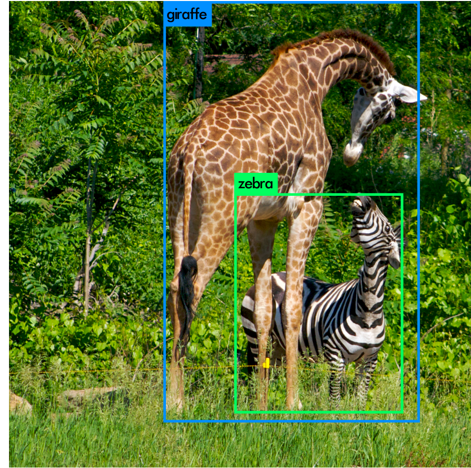

Figure 6.3: Animals recognized after the execution of the YOLO library. Output generated by the Darknet framework.

```
Predicted in 11.708465 seconds.
giraffe: 90%
zebra: 80%
```

Listing 6.9: Output file generated with the certainty of the recognized objects.

Results

After the execution of this use case, the following metrics were retrieved:

- 2 minutes of real time used to finish the experiment.
- 880 minutes as the total aggregated execution time across the multiple Lambda function invocations.
- 4,575 different objects and animals recognized.

As summary, in only two minutes we have read, downloaded, processed and uploaded a thousand images without having to worry about the deployment and the management of the architecture.

It is important to point out four main conclusions that arise from this case study:

1. Without SCAR the user has no easy way of using specific libraries such as Darknet in serverless providers like AWS Lambda.
2. The user does not have to manage the deployment of computational infrastructure, auto-,scaling, coordinating the execution of jobs, etc. Instead, the serverless computing platform introduced seamless elasticity by performing multiple concurrent executions.
3. The simplicity of the programming model introduced by SCAR just requires the user to write a shell-script to process a file assuming that will be automatically delivered. This is probably the simplest, most convenient approach to perform a file-processing application on the Cloud.
4. Once the Lambda function has been created by SCAR, this turns into a reactive service that is left on the Cloud at no cost unless it is triggered again by uploading a new file to the bucket. This will cause a new Lambda function invocation, resulting in the creation of the container and execution of the shell-script to process the file. This has an important economic factor, since real pay-per-use is enforced as opposed to the pay-per-deploy approach that happens when deploying VMs in a cloud service, which has a cost regardless of the actual use.

In addition, notice that the ability to scale in the order of thousands of Lambda function invocations reduces the requirement for a job scheduler, in cases where the incoming workload can be seamlessly absorbed by the underlying computing platform. In this use case, after reading the files, the SCAR client creates one asynchronous invocation for each file found. When the Lambda service receives an asynchronous invocation it doesn't launch the function immediately, instead it queues the received event and if the scaling policies allow it, that is, the concurrency limits are not exceeded, it takes the event from the queue and then launches the function to process such event. This feature allows the service to manage more events than their concurrency limits without failure. For example, if the user wants to process 20,000 files, then 1 synchronous invocation (i.e. the first one), and 19,999 asynchronous

invocations are created by the SCAR client and then sent to the Lambda service. The Lambda service will queue the invocations and will try to invoke the function again for up to 6 hours with increasing retry intervals. However, have in mind that if the queue is too big to be processed some events in the queue might be not read. A deeper explanation of this automatic functionality can be found in the AWS Lambda documentantion [23].

To finish, we present a cost of the case study execution. Based on the metrics extracted earlier, the average execution time of the invocations is 52.8 seconds. As we said earlier, each function instance uses 2,048MB of RAM, and one invocation per photo available in the bucket was carried out, so 1,000 functions were launched in total. Then, the AWS Lambda pricing calculator[5] indicates that the cost of this use case is:

$$\begin{aligned}
&\text{RoundUp } (52,800) = 52,800 \text{ ms(Duration rounded to nearest 100 ms)}\\
&1,000 \text{ requests} \times 52,800 \text{ ms} \times 0.001 \text{ sec/ms} = 52,800 \text{ total compute (seconds)}\\
&2 \text{ GB} \times 52,800 \text{ seconds} = 105,600 \text{ total compute (GB-s)}\\
&105,600 \text{ GB-s} \times 0.0000166667 \text{ USD} = 1.76 \text{ USD (monthly compute charges)}\\
&1,000 \text{ requests} \times 0.0000002 \text{ USD} = 0.00 \text{ USD (monthly request charges)}\\
&1.76 \text{ USD} + 0.00 \text{ USD} = \mathbf{1.76\ USD}
\end{aligned} \tag{6.1}$$

However, since AWS Lambda offers a free usage tier that includes 1,000,000 requests and 400,000 GB-seconds of compute time per month, and this use case involved 1,000 requests and 105,600 GB-seconds, the cost of classifying the images using the free tier is 0 USD.

6.3 Video Processing Service in AWS

We already saw a video processing service in section 5.3 refered to OSCAR. To demonstrate that the same workflow can be deployed in a public serverless provider, we present this use case. A video analysis service that takes a video as an input and generates an analysis of the keyframes of such video using the state-of-the-art real time object detection system [161]. The goal of the service is to apply object recognition to certain frames of the video in order to reason about the content of the video. Figure 6.4 shows the workflow of the architecture proposed.

[5] https://aws.amazon.com/lambda/pricing/

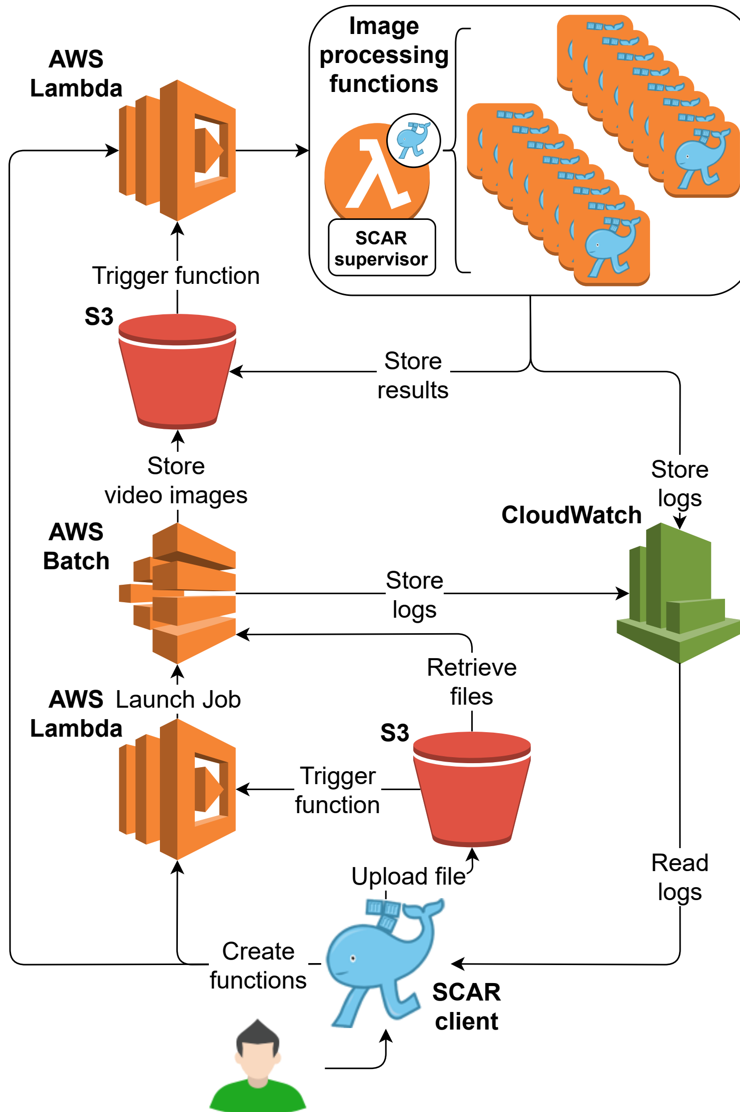

Figure 6.4: Workflow of the Video Analysis Service. Two Lambda functions and one Batch environment are defined. The execution starts after an S3 bucket invokes a Lambda function that triggers a Batch job in charge of extracting the keyframes of the input video. Then, for each keyframe extracted, a function to analyze the keyframes is launched.

The *ffmpeg* library is used to extract the keyframes from the input video and the *darknet* framework in combination with the YOLO library is used to analyze the extracted keyframes. The docker images used in this use case are the public images *grycap/ffmpeg* and *grycap/darknet* for the video extraction and the image processing respectively.

Listing 6.10 and 6.11 show the SCAR configuration files used to define the architecture of the use case.

```
functions:
  scar-batch-ffmpeg-split:
    image: grycap/ffmpeg
    init_script: split-video.sh
    execution_mode: batch
    s3:
      input_bucket: scar-ffmpeg
      output_bucket: scar-ffmpeg/scar-batch-ffmpeg-split/video-output
```

Listing 6.10: SCAR configuration file for defining a default Batch environment and a specific output folder.

Listing 6.10 defines the execution mode as *batch*, so the defined Lambda function automatically transforms the received event into a Batch job and redirects such job to the Batch service without trying to execute it first. In addition, a specific output folder inside an S3 bucket is defined. By specifying a folder instead of letting SCAR to create a default output folder we have absolute control of the data workflow.

Listing 6.11 shows the definition of the Lambda function used to analyze the video keyframes. You can see that the input bucket defined in this file is the same as the output bucket defined in listing 6.10. Linking the functions using buckets allow us to define serverless workflows comprised by different Lambda functions that can be executed automatically without user intervention.

```
functions:
  scar-lambda-darknet:
    image: grycap/darknet
    memory: 3008
    init_script: object-detection.sh
    s3:
      input_bucket: scar-ffmpeg/scar-batch-ffmpeg-split/video-output
      output_bucket: scar-ffmpeg/scar-batch-ffmpeg-split/image-output
```

Listing 6.11: SCAR configuration file defining a Lambda function to execute the Darknet framework.

In addition, listings 6.12 and 6.13 show the scripts used inside the defined containers.

```
#! /bin/sh

echo "SCRIPT: Splitting video file $INPUT_FILE_PATH in images and
    storing them in $TMP_OUTPUT_DIR. One image taken each second"
ffmpeg -loglevel info -nostats -i $INPUT_FILE_PATH -q:v 1 -vf fps=1
    $TMP_OUTPUT_DIR/out%03d.jpg < /dev/n
```

Listing 6.12: Script used to split the video in 1 second keyframes.

```
#!/bin/bash

RESULT="$TMP_OUTPUT_DIR/$FILE_NAME.out"
OUTPUT_IMAGE="$TMP_OUTPUT_DIR/`basename $INPUT_FILE_PATH .jpg`"

echo "SCRIPT: Analyzing file '$INPUT_FILE_PATH', saving the result in '
    $RESULT' and the output image in '$OUTPUT_IMAGE.png'"

cd /opt/darknet
./darknet detect cfg/yolo.cfg yolo.weights $INPUT_FILE_PATH -out
    $OUTPUT_IMAGE > $RESULT
```

Listing 6.13: Script used to launch the YOLO object detection library of the Darknet framework (object-detection.sh).

After defining the containers, the scripts, and the SCAR configuration files, the workflow is launched by uploading a video file to the input folder of the lambda Batch function. The following trace details the steps followed during the use case:

1. Using the SCAR client, the user uploads a video to the input folder of the S3 bucket linked with the video processing function. This is the last step that requires user intervention.

2. After the video upload finishes, the S3 bucket automatically sends a trigger to activate the video processing function. The SCAR supervisor deployed inside the Lambda function detects the Batch environment and automatically launches a Batch job attaching all the environment variables required to execute such job. Just before the Batch job is launched, the SCAR supervisor retrieves the video from the S3 bucket and stores it in a folder shared with the Batch container. Then, the container is launched and the job extracts the keyframes of the video and stores them in the `$TMP_OUTPUT_DIR`. The SCAR supervisor stores those files in the output folder of the specified S3 bucket, thus triggering the image processing Lambda function.

3. To finish, an image processing function that analyzes the keyframe is triggered for each keyframe stored in the S3 bucket. Then through the supervisor, the function stores the result back in their own defined output folder of the S3 bucket.

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

As summary of this use case, we demonstrated how to combine different AWS services to create a video analysis service in the Cloud. Among the features of the architecture comprising the analysis service we can remark that it has automated elasticity and scale-to-zero capabilities, no upfront investment has to be made by the user, and no infrastructure pre-provision is needed.

6.4 Plant classification

This use case was adapted from the repository[6] of the DEEP-Hybrid-DataCloud project [51]. All the adapted code to reproduce this use case can be found in the OSCAR repository[7]. The goal of this use case is to provide a classification service for different plant images at the same time that different EGI services are integrated. To achieve this, we are going to combine services from the EGI platform. The OSCAR cluster is deployed on top of an elastic Kubernetes cluster deployed on the EGI federated cloud. As a storage service we are going to use a Onedata [142] space through EGI Datahub [182][58]. The EGI Datahub service makes existing large scale open data collections discoverable and available for both EGI users and the general public in the case of open data collections[8].

To deploy the cluster, first the users have to connect to the EC3 service trough the EGI Applications on Demand portal[9]. Once the users are properly identified through the EGI Check-in service, they can deploy the OSCAR platform by selecting it in the LRMS selector as seen in Figure 6.5.

After the cluster has been deployed, the usage of the OSCAR platform is the same as in all the other presented use cases. By using the provided GUI

[6] https://github.com/deephdc/plant-classification-theano

[7] https://github.com/grycap/oscar/tree/master/examples/plant-classification-theano

[8] The Datahub web portal can be found here: `https://datahub.egi.eu`

[9] https://marketplace.egi.eu/42-applications-on-demand-beta

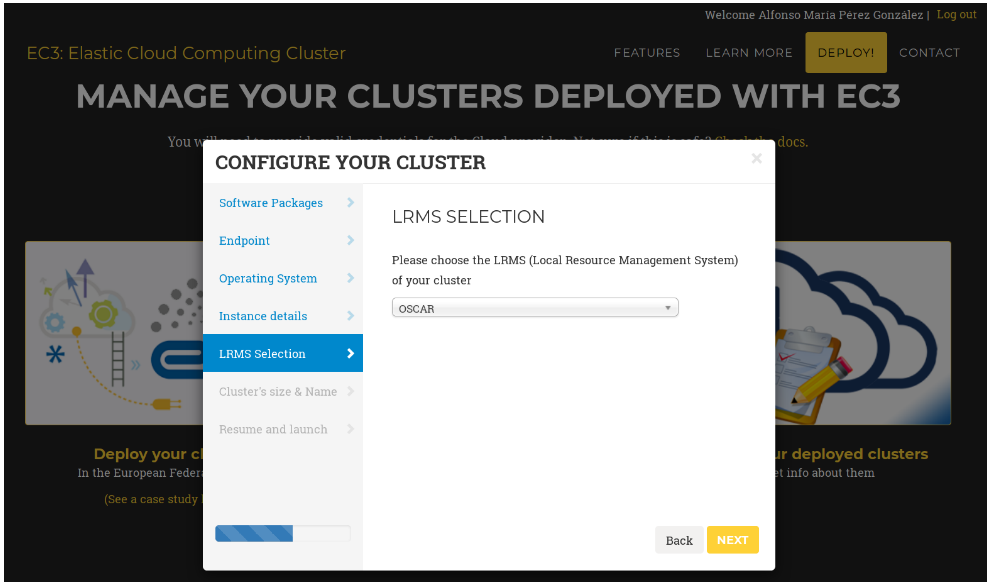

Figure 6.5: Deploying an OSCAR cluster in the EGI federated cloud through the EC3 tool.

we create the plant classification function that is automatically linked to the Onedata space. As it is shown in Figure 6.6, through the Oneprovider portal the users can upload their data to be classified.

Once the images have been classified they are being automatically copied to the output folder and can be retrieved by the platform users.

Further information about this use case is available in the corresponding YouTube video[10]. As a summary, we demonstrated how to deploy an image classification service making use of public European infrastructures.

6.5 Multi-cloud workflow for video processing

This use case was presented originally in the Ibergrid 2019 Congress [90] and it was part of a hands-on tutorial. We just saw how to process a video file using AWS services, and in section 5.3 we showed how to do it in OpenNebula, an on-premises cluster provider. The goal of this use case is to create a workflow

[10] https://www.youtube.com/watch?v=ZtAlVc1uLwc

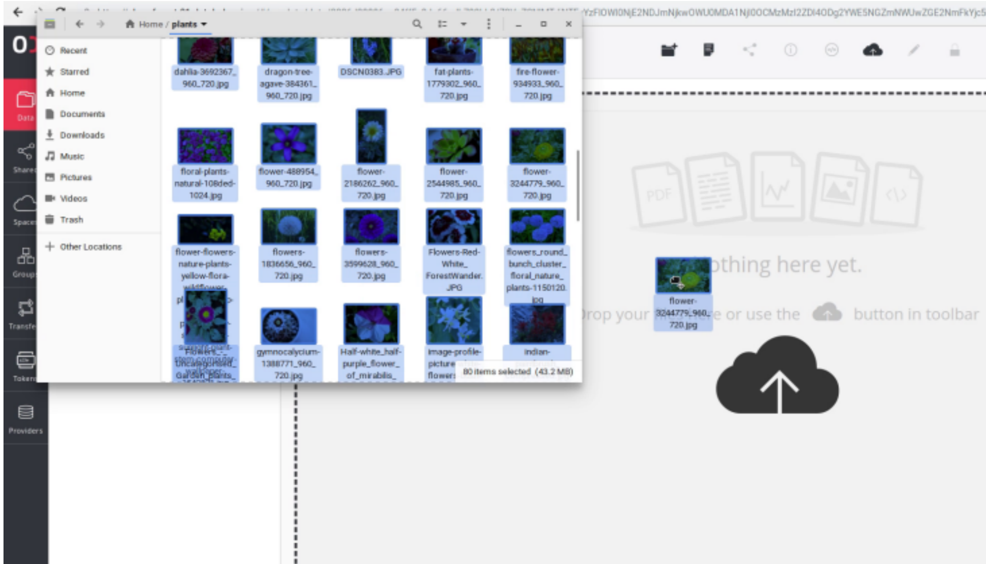

Figure 6.6: Onedata portal from where the users can upload and download their data. The folders inside the space are linked with the OSCAR functions.

Predicted labels:

1. Abies alba | 68 %
2. Picea abies | 20 %
3. Abies pinsapo | 3 %
4. Abies grandis | 1 %
5. Woodwardia radicans | 1 %

Figure 6.7: Plant species recognized by the plant classification service.

that is able to execute functions and share data among different clouds. As a shared storage provider we are going to use the Onedata service through EGI DataHub (as in the previous use case). Inside EGI DataHub, the users have

storage quotas called spaces in where their files can be stored. In this use case, we are using one of this spaces to communicate clouds from different providers. Figure 6.8 shows the high level architecture of the deployed workflow.

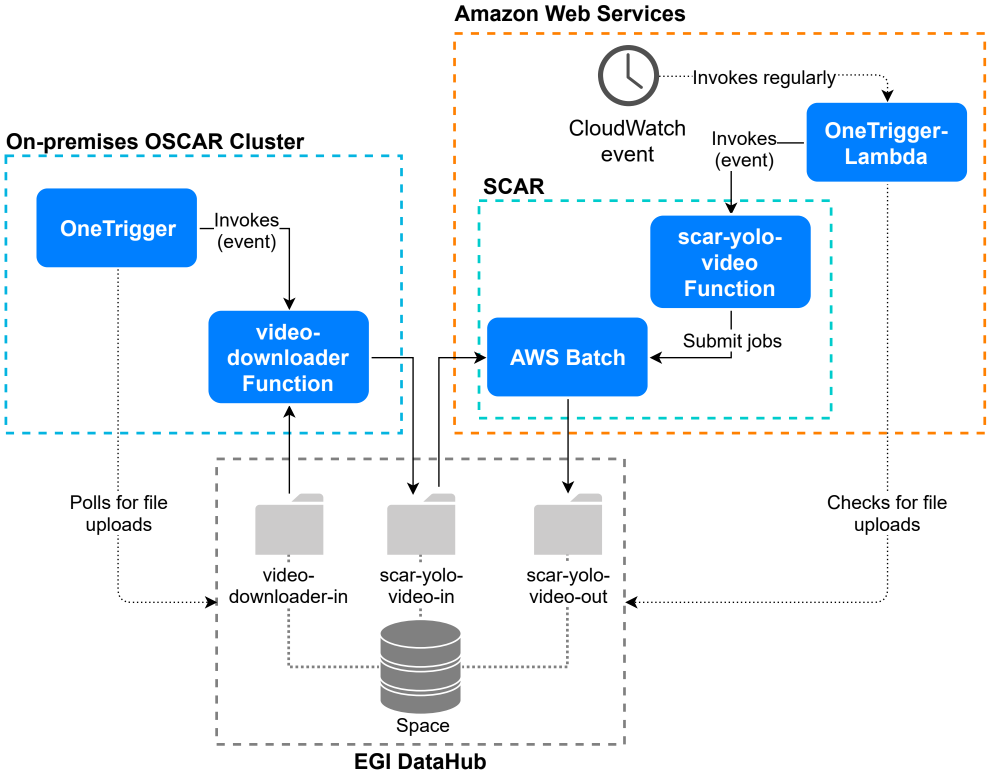

Figure 6.8: Workflow of the multi-cloud data workflow. The *scar-yolo-video-in* folder is used as a link to communicate the different cloud infrastructures.

The workflow is composed by two functions, a *video-downloader* function to download videos with the aria2 [28] tool (deployed in an on-premises cluster) and a function to submit AWS Batch jobs that will process the downloaded video (deployed in a public AWS cluster). The OSCAR function will receive a file with a list of URLs, download all the videos, and store them in the specified output folder. The AWS function will apply pattern recognition to the uploaded video and will store the result in the specified output folder.

Unfortunately, right now Onedata lacks the functionality of triggering an event when a new file is created in an storage space, so in addition to the two

main functions, another one has to be deployed in each cloud to manage automatically the file discovery and eventing in Onedata. More information about the inner working of the OneTrigger function can be found in the master's thesis [164].

In OSCAR the function called *OneTrigger* polls periodically the *video-downloader-in* folder if the Onedata space. If a new file is detected, an event is generated and the OSCAR function is launched to download the required videos. In AWS, a Lambda function called *OneTrigger-Lambda* is created to check for file uploads in the storage space. This function is triggered periodically by a defined CloudWatch event and when it discovers a new file in the linked folder (*scar-yolo-video-in* in this case) it creates an event that is passed to the AWS Lambda service to launch the video processing function. Once the video is processed, it is stored in the *scar-yolo-video-out* and the user can download the result from the Onedata interface.

Summarizing, the use case presented has demonstrated that a data-hybrid cluster infrastructure can be created using the proposed high-throughput programming model in combination with the SCAR and OSCAR tools. This use case tackles the research challenge of using a public open data storage service to communicate different FaaS workflows in different infrastructures. It also paves the way for using on-premises functions for preprocess and anonymize sensitive data and then take advantage of the high scalability and high-end resources of public providers to do the heavy computation at the same time that all the data (even at the intermediate steps) is kept secure in a controlled storage service.

6.6 Air pollution information service

This use case appeared originally in Sebastián Risco's master thesis [164], a more detailed description of the implementation and the services can be found there. The use case was developed to prove that the event-driven file-processing programming model proposed in chapter 4 can be also used to process data from open-data repositories. Figure 6.9 provides a high level overview of the use case architecture.

Three functions have been developed to process the data available at the waqi.info portal, the web page belonging to the World Air Quality Project [178]. This web portal offers an Application Programming Interface (API) to access the data gathered from more than 10,000 stations around the world.

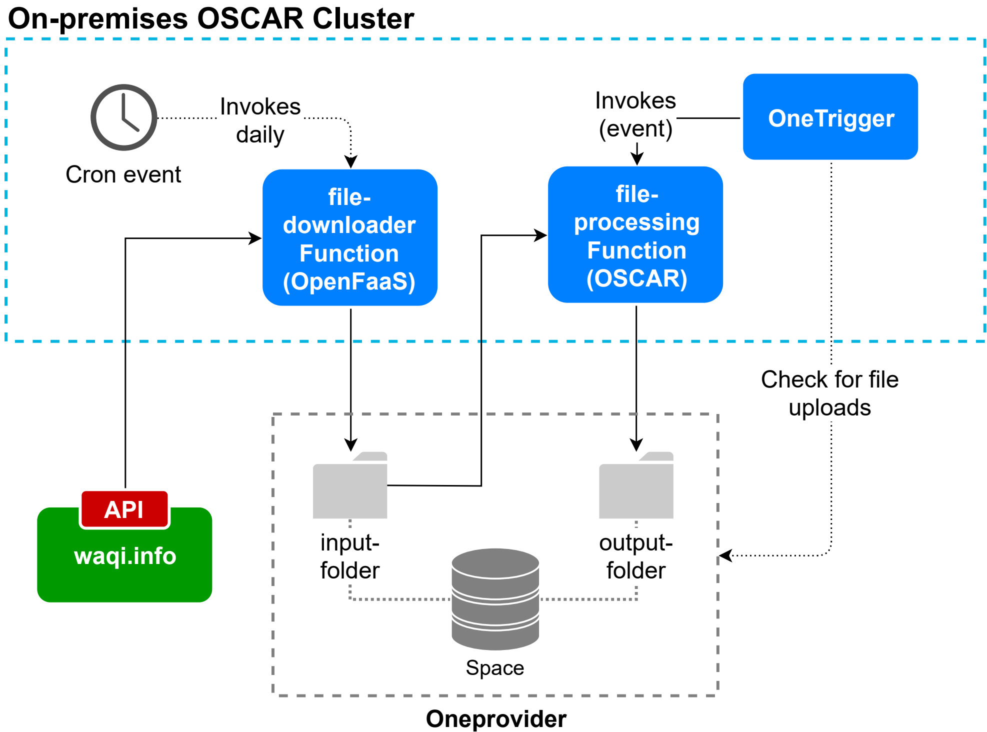

Figure 6.9: Architecture of the air pollution service created using open-data from waqi.info.

Thus, the job of the first function, that is invoked daily, consist on retrieving the data from this portal each day and store it in the *input-folder* specified. The OneTrigger function also checks periodically if new files have been created inside the Onedata space, if a new file is found, the file processing function is launched, the data is analyzed and finally, a snapshot with a summary of the air quality of the region specified is stored in the output folder of the Onedata service, as it can be seen in Figure 6.10.

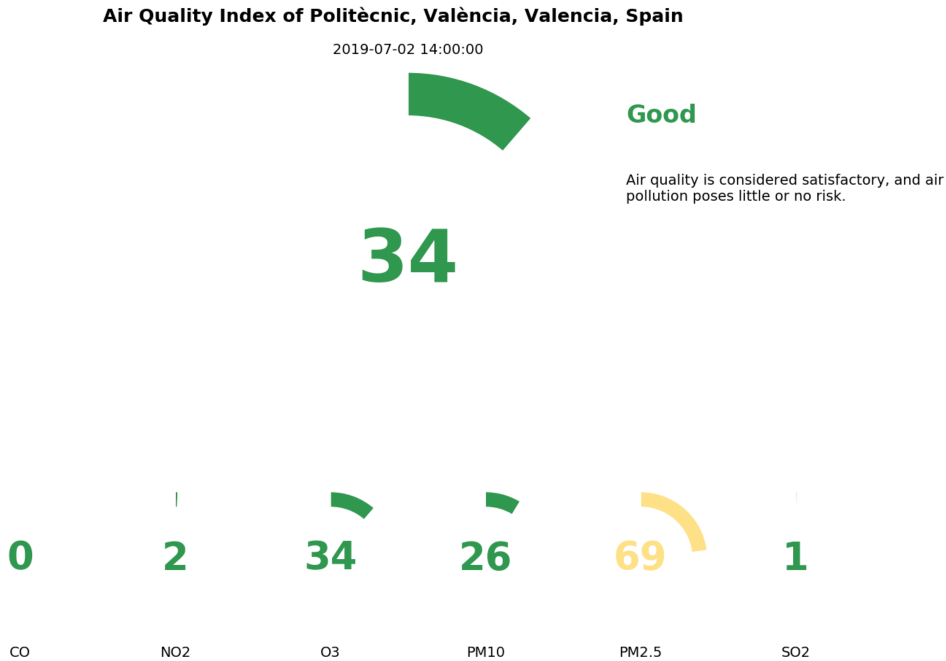

Figure 6.10: Snapshot generated by the air pollution service developed.

6.7 Monetizing Private Algorithm Workflow Executions

This use case appeared originally in Adolfo Mendez's master thesis [120], a more detailed description of the implementation and the services can be found there. The proposed use case presents a diagnose system to detect cardiac issues through the usage of advanced diagnose algorithms. The system will use different diagnose algorithms based on the sensor used to collect the data. To be able to process input data from different sources, the system internally uses two different workflows. The first workflow detects cardiac valve problems, and the second one detects cardiac arrhythmia. Both workflows receive input data from external sensors connected to the internet, and the workflows start when data is collected during a patient recognition test. The workflow to detect cardiac valve problems receives input from an acoustic sensor, and the complete workflow is composed by two filter steps, and one diagnose step that is based on the acoustic signal previously filtered. The workflow to detect cardiac arrhythmia can receive input from two

different sensors (optical or electrical), the input signals are also filtered and then processed by an arrhythmia diagnose algorithm. The generated diagnosis are automatically stored in a storage system and finally an email with the diagnosis is sent to the practitioner that initiated the workflow.

The goal of this use case is to study how to deploy and monetize private algorithms using the AWS services and the SCAR tool. To be able to monetize the API usage, the API Gateway developer portal [11] has been used. Through this portal, the users can subscribe to an API defined in API Gateway. This subscription will have a cost defined by the API creator and can be charged per use (e.g. 0.001$ per invocation) or per time (e.g. 1$ per day). Once the user is subscribed, an API Key is created an associated to the user account. Using this API Key, the user can invoke the registered APIs and execute the defined workflows. The complete case study is described in Figure 6.11.

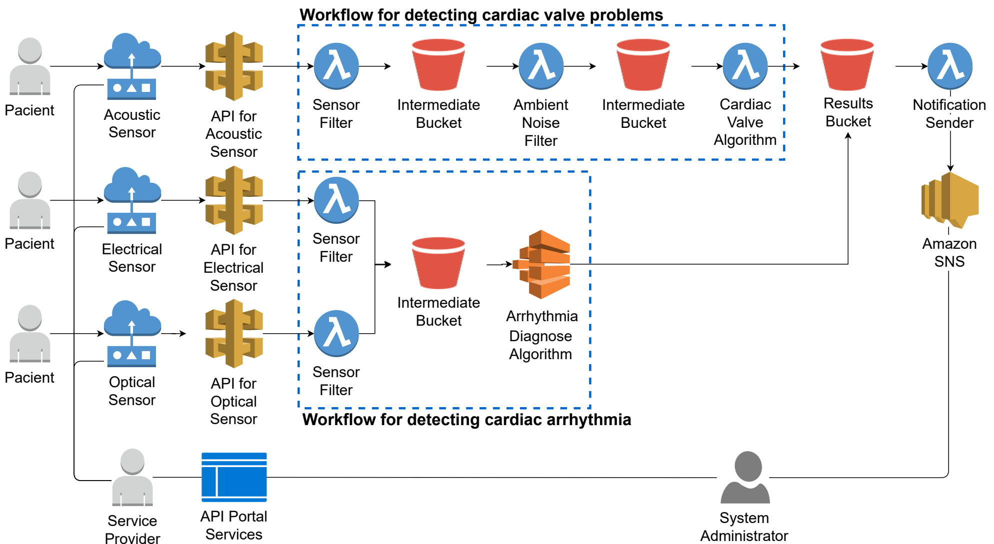

Figure 6.11: High level architecture of the monetized workflow.

As shown in Figure 6.11, the sensors are in charge of triggering the workflows. Once a sensor registers new data, it invokes the defined workflow through the linked API. In both workflows, the AWS Lambda functions were defined following the single responsibility principle, that states that the code of each function will only have responsibility for a single part of the functionality. Furthermore, instead of connecting directly the functions, several intermediate buckets were defined. This buckets allow us to check the

	Serverless ($)	EC2 ($)	On-premises ($)	Cost per request ($)
Scenario 1: 10,000 requests	113.84	142.20	2,957.75	32.02
Scenario 2: 150,000 requests	1,718.08	2,133.00	3,491.11	0.215
Scenario 2: 4,000,000 requests	45,795.38	56,880.00	82,291.97	0.075

Table 6.2: Monthly costs for the three different scenarios.

intermediate generated data easing the application tracing and debugging. Due to Lambda time restrictions, the last part of the cardiac arrhythmia algorithm was defined in AWS Batch. To finish, the last Lambda function publish a message with the workflow results in the AWS Simple Notification Service (SNS) service. Among other features, AWS SNS allow us to send SMS or push notifications to mobile phones, emails, and it encrypts all the messages for increased security. All the Lambda functions, the API endpoints, and the S3 buckets were defined with SCAR. Unfortunately SCAR doesn't allow yet to define and link AWS SNS topics so the SNS topic and the monetized API endpoint were created manually.

To test the defined infrastructure three workloads were created: 1) 10,000 requests/month; 2) 150,000 requests/month; 3) 4,000,000 requests/month. These workloads were defined to simulate the different stages of success of a medical application, so we can analyze how the cost varies depending on the number of users per month.

Table 6.2 summarizes the costs for the defined workloads. We can see that just by using the presented serverless architecture instead of a pure EC2 approach we are saving $\sim$ 20% of the cost. To calculate the cost of the on-premises infrastructure, the calculator provided by AWS[11] has been used. The last column shows the cost that each API invocation should have so we can have a 60% of benefit. As it happens with all this services, the prices decreases drastically when the number of requests per month increase. Finally, a fine-

[11] `https://awstcocalculator.com/`

grained analysis of the costs and a more complete analysis of the workloads can be found in the master's thesis were this use case is presented.

6.8 GROMACS in AWS Batch

The last use case presents a scenario that demonstrates that not all the scientific workloads can directly benefit from moving to the serverless environment. A more detailed explanation of this cost study can be found in the Ibergrid 2019 contribution [181].

The GROningen MAchine for Chemical Simulations (GROMACS) software is a molecular dynamics package mainly designed for simulations of proteins, lipids, and nucleic acids. The goal of this use case was to make a cost comparison between AWS Lambda and a farm server executing the GROMACS package, but due to Lambda restrictions, the final tests were carried out in the AWS Batch infrastructure instead of AWS Lambda. The two different architectures analyzed in this use case are shown in Figure 6.12.

For this use case SCAR was used to create the complete test environment in AWS. As conclusions of this cost study, all the executed tests were executed faster in AWS than in the on-premises infrastructure ($\sim$ 19% less execution time and $\sim$ 23% more calculations per day) but the cost of the AWS services (2,330 €) were almost six times more than the on-premises (394 €) infrastructure ($\sim$ 590%) (a complete trace of the cost calculations can be found in the Ibergrid contribution [181]).

Regarding this use case, one could claim that it is not a pure serverless architecture, because we are using the underlying AWS ECS infrastructure that supports AWS Batch. Additionally, one could argue that if the container executing GROMACS would fit in AWS Lambda, the price could have come down substantially, but the truth about this use case is that is not suitable for the serverless infrastructure as is. It can be concluded that serverless technology should not be seen as a one-size-fits-all technology, and each problem should be studied before migrating it to a serverless architecture.

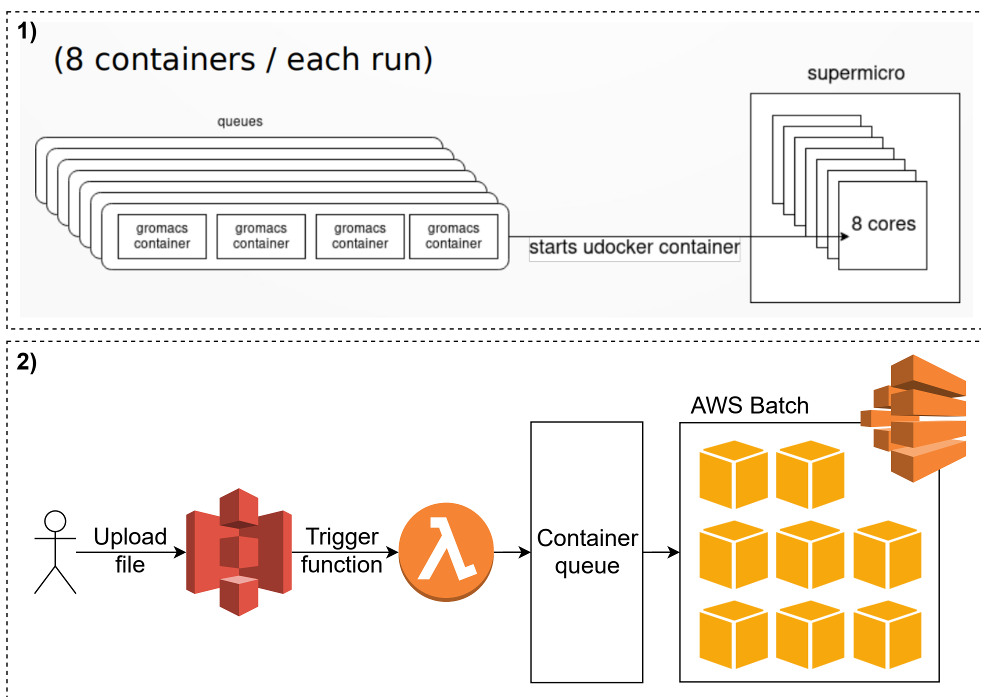

Figure 6.12: Different execution queues for executing GROMACS. 1) The workflow is executed in a bare metal server using a LRMS. 2) The workflow is executed in AWS Batch.

6.9 Scientific diffusion

During the realization of the this thesis the following papers were presented in indexed journals and congresses:

- JCR Q1 Journal: Pérez, A., Moltó, G., Caballer, M., & Calatrava, A. (2018). Serverless computing for container-based architectures. Future Generation Computer Systems, 83, 50–59. https://doi.org/10.1016/j.future.2018.01.022. This paper tackles the research goal of executing generic applications in public serverless infrastructures. The SCAR tool with its Docker in Lambda capabilities and a prototype of the high throughput programming model are presented. It is the basis for chapter 3.

- GGS Class 2 Congress[12]: Pérez, A., Caballer, M., Moltó, G., & Calatrava, A. (2019). A programming model and middleware for high

throughput serverless computing applications. In Proceedings of the ACM Symposium on Applied Computing (Vol. Part F147772, pp. 106–113). Association for Computing Machinery. https://doi.org/10.1145/3297280.3297292 This paper deals with the research challenge of providing the scientific users with a generic highly-scalable data-driven programming model. It presents a complete version of the high throughput programming model in combination with two use cases, one of them demonstrating the execution of function workflows. It is the basis for chapter 4.

- GGS Class 2 Congress[12]: Perez, A., Risco, S., Naranjo, D. M., Caballer, M., & Molto, G. (2019). On-Premises Serverless Computing for Event-Driven Data Processing Applications (pp. 414–421). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/cloud.2019.00073 This paper engages in the research challenge of providing an on-premises infrastructure focused on event-driven executions of compute-intensive applications. The OSCAR framework is presented, introduces the scalability capabilities at node level of a Kubernetes cluster and asses the infrastructure with two different use cases. This paper is the basis for chapter 5.

The SCAR tool, the high throughput programming model, and the OSCAR platform were developed under the “BigCLOE” project, funded by the Spanish “Ministerio de Economía, Industria y Competitividad" with the reference number TIN2016-79951-R. In addition, the development of the OSCAR platform was partially funded by the EGI Strategic and Innovation Fund and by the Primeros Proyectos de Investigación (PAID-06-18), Vicerrectorado de Investigacion, Innovación y Transferencia de la Universitat Politècnica de València (UPV), València, Spain. To finish, the SCAR [61] and the OSCAR [60] tools have been registered in the Explora i+D+i portal of the UPV.

Regarding the SCAR tool, we could say that it has been widely accepted by the users' community. In its GitHub repository[13] it has been starred more than 480 times and it counts with 31 forks. Furthermore, the SCAR tool is featured in the Cloud Native Computing Foundation (CNCF) landscape of serverless tools, as it can be seen in their web portal [85]. The CNCF is part of the nonprofit Linux Foundation and serves as the vendor-neutral home for many of the fastest-growing open source projects, including among others the Kubernetes project. Moreover, the CNCF brings together the world's top

[12]Based on the GII-GRIN-SCIE (GGS) Conference Rating: http://gii-grin-scie-rating.scie.es
[13]https://github.com/grycap/scar

developers, end users, and vendors, and runs the largest open source developer conferences, which has brought a great impact in the SCAR's project visibility.

Additionally, an OSCAR prototype was presented at the Ibergrid 2018 congress [98], and an OSCAR version featuring hybrid-data clouds and GPU support was presented in the Ibergrid 2019 congress [99]. Moreover, OSCAR has been integrated in the European Grid Infrastructure (EGI) Federated Cloud [63]. Through the EC3 portal[14], the users can deploy, in the different federated clouds, an elastic Kubernetes cluster with the OSCAR platform configured.

[14] https://servproject.i3m.upv.es/ec3-ltos/index.php

Chapter 7

Conclusions

This chapter presents a summary of the work done in this thesis and it also states different research paths that can be followed upon the completion of this work.

7.1 Summary and Contributions

This thesis has presented a set of tools and platforms together with a programming model to allow the users to execute highly-parallel event-driven file-processing applications both in public and in private cloud infrastructures. By adopting the presented tools, the users can define workflows of functions and benefit from transparent data management from the beginning to the end of the such workflows.

To this aim, first we introduced SCAR, a framework to execute container-based applications using serverless computing, exemplified using Docker as the technology for containers and AWS Lambda as the underlying serverless platform. SCAR represents a step forward contribution to the state of the art, implemented in an open-source framework, that opens new avenues for adopting serverless computing for a myriad of scientific applications distributed as Docker images. Using SCAR, customized execution environments can now be employed instead of being locked-in to programming functions in the programming languages supported by the serverless platform. This has easily introduced the ability to run generic applications on specific runtime environments defined by Docker Images

stored in Docker Hub, a functionality that is actually missing from most of the current serverless computing platforms. But SCAR not only provides means to deploy Docker containers in AWS Lambda, it also manages the Lambda functions' life cycle and eases the execution of the serverless workflow by applying optimizations without the need of user intervention, such as caching the container's underlying file system to minimize the execution time.

Second, we introduced a high throughput computing programming model to create highly-parallel event-driven serverless applications. The execution environment for this applications was provided by containers created out of customized Docker images. The ability to run code in response to events and the large-scale elasticity provided by the underlying serverless platform opens new avenues for efficient High Throughput Computing tasks. This was demonstrated by the case studies where the programming model abstracted away many implementation details typically required on computing frameworks. Moreover a cost analysis was done comparing the serverless programming model presented and the usual cloud computing architectures. Although the cost analysis revealed that running a serverless architecture could be costlier than deploying virtual machines, the savings in configuration and execution time in combination with the pay-per-use model offered by AWS make the public serverless architectures a good option to deploy applications that have to deal with bursty workloads of short stateless jobs.

Third, to offer an alternative to public serverless providers and avoid the limitations imposed by them, the OSCAR platform was developed. OSCAR is composed of different open-source tools that allow us to provide similar features than the ones present in the public providers. Moreover, while developing OSCAR, a Kubernetes plugin for CLUES was developed, providing the underlying Kubernetes cluster with elastic node capabilities. This feature allow us to offer elasticity at different levels of the infrastructure, first at hardware level, provisioning and terminating nodes based on the CPU and memory consumption of the deployed systems, and second at function level, launching more functions when the workload is increased and scaling down to zero when there is no workload available. To provide the users with an easy access point to the OSCAR platform, a web interface was also developed, where the users can control the complete life cycle of the application, creating, launching and deleting the functions needed for their applications.

Finally to evaluate all the tools developed, several use cases (created by ourselves and by the community) were tested. In these use cases we saw how

to add support for new languages in AWS, we successfully deployed image an video analysis services, we showed the integration of the programming model with public scientific infrastructures (e.g. EGI), we demonstrated how to create data-hybrid workflows, how to take advantage of open-data portals, how to monetize private algorithm workflows and, to finish, we presented a use case to demonstrate that maybe is not cost-wise to migrate all the applications to the proposed serverless architectures without doing first an study of the application deployment process and the costs implied.

7.2 Future work

Future work in the SCAR tool includes adapting the development to other serverless providers. Our dependence on udocker, which is developed in Python, suggests using a provider supporting that language. Fortunately, the main serverless providers support Python in their FaaS services. Furthermore, IBM Cloud Functions supports directly the definition of Docker container as functions, making the portability of the high throughput programming model even easier. In addition, SCAR users could benefit from a mechanism that maintains the deployed Lambda functions 'hot', based on the knowledge extracted from the freeze/thaw cycle study by means of periodic invocations of the Lambda functions.

Regarding the high throughput programming model, we plan to simplify the definition of data driven workflows, so the user can define complete FaaS applications by using a simple infrastructure definition file in YAML. Moreover, we have plans to add support for more storage providers and more event services to ease the deployment of data-hybrid serverless applications that encompass the high scalability capabilities of the cloud providers and the less restricted environments of the on-premises deployments.

Finally, further work in the OSCAR platform by refining the requirements and the behavior of the required pods could lead to a better usage of the cluster resources and thus to a higher throughput when processing functions. Moreover, we presented a GPU integration in the public serverless services through AWS Batch, but for OSCAR we have not provided yet a solution to allow users execute workflows that can take advantage of GPU resources (although this feature is being initially explored, further research is required). In addition, we are planning to integrate SCAR with OSCAR by means of a shared CLI to provide the users with a unique tool from where to deploy event-based data-hybrid serverless applications.

Bibliography

[1] Adobe. *Adobe Creative Cloud*. URL: https://www.adobe.com/es/creativecloud.html (visited on 09/20/2019) (cit. on p. 9).

[2] Carlos de Alfonso et al. "Multi-elastic Datacenters: Auto-scaled Virtual Clusters on Energy-Aware Physical Infrastructures". In: *Journal of Grid Computing* 17.1 (July 2019), pp. 191–204. ISSN: 15729184. DOI: 10.1007/s10723-018-9449-z (cit. on p. 78).

[3] Alibaba. *Alibaba Cloud Function Compute*. URL: https://www.alibabacloud.com/products/function-compute (visited on 10/11/2019) (cit. on p. 15).

[4] Alpine. *Alpine Linux*. URL: https://www.alpinelinux.org/ (cit. on p. 46).

[5] Amazon. *Amazon API Gateway*. URL: https://aws.amazon.com/apigateway (visited on 06/19/2019) (cit. on p. 56).

[6] Amazon. *Amazon CloudWatch*. URL: https://aws.amazon.com/cloudwatch (visited on 06/20/2019) (cit. on p. 37).

[7] Amazon. *Amazon EC2 Container Service (ECS)*. URL: https://aws.amazon.com/es/ecs/ (visited on 06/19/2019) (cit. on pp. 3, 11).

[8] Amazon. *Amazon EKS*. URL: https://aws.amazon.com/eks (visited on 09/23/2019) (cit. on pp. 3, 10).

[9] Amazon. *Amazon EventBridge*. URL: https://aws.amazon.com/eventbridge/ (cit. on p. 20).

[10] Amazon. *Amazon Web Services (AWS)*. URL: https://aws.amazon.com/ (visited on 06/19/2019) (cit. on p. 2).

[11] Amazon. *API Gateway Developer Portal*. URL: https://github.com/awslabs/aws-api-gateway-developer-portal (visited on 10/07/2019) (cit. on p. 115).

[12] Amazon. *AWS Amplify*. URL: https://aws.amazon.com/amplify/ (visited on 09/23/2019) (cit. on p. 11).

[13] Amazon. *AWS Auto Scaling*. URL: https://aws.amazon.com/autoscaling/ (cit. on p. 72).

[14] Amazon. *AWS EC2*. URL: https://aws.amazon.com/ec2/ (visited on 09/20/2019) (cit. on p. 8).

[15] Amazon. *AWS Elastic Beanstalk*. URL: https://aws.amazon.com/elasticbeanstalk (visited on 06/20/2019) (cit. on pp. 9, 28).

[16] Amazon. *AWS Fargate*. URL: https://aws.amazon.com/fargate/ (cit. on p. 15).

[17] Amazon. *AWS Lambda*. URL: https://aws.amazon.com/lambda (visited on 06/19/2019) (cit. on pp. 3, 12, 15).

[18] Amazon. *AWS Lambda : 1792MB 1vCPU*. URL: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html (cit. on p. 16).

[19] Amazon. *AWS Lambda FAQ*. URL: https://aws.amazon.com/lambda/faqs (cit. on p. 44).

[20] Amazon. *AWS Serverless Application Model*. URL: https://github.com/awslabs/serverless-application-model (visited on 06/19/2019) (cit. on p. 28).

[21] Amazon. *Boto 3 Documentation*. URL: https://boto3.readthedocs.io/en/latest/ (visited on 06/20/2019) (cit. on p. 37).

[22] Amazon. *Invoke*. URL: http://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html (visited on 06/20/2019) (cit. on p. 47).

[23] Amazon. *Lambda asycnhronous invocations*. URL: https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html (cit. on pp. 60, 104).

[24] Apache. *Apache Camel*. URL: https://camel.apache.org/ (cit. on p. 27).

[25] Apache. *Apache Camel Components*. URL: https://camel.apache.org/components/latest/ (cit. on p. 27).

[26] Apache. *Apache MXNet*. URL: https://mxnet.apache.org/ (cit. on p. 29).

[27] Apex Software. *Apex: Up, deploy serverless apps in seconds*. URL: http://apex.run/ (visited on 06/19/2019) (cit. on p. 28).

[28] Aria2. *Aria2*. URL: https://aria2.github.io/ (visited on 10/03/2019) (cit. on p. 111).

[29] *AWS Batch*. 2019. URL: https://aws.amazon.com/batch/ (visited on 12/09/2019) (cit. on p. 57).

[30] *AWS EC2 ECU*. 2019. URL: http://aws.amazon.com/ec2/faqs/ (visited on 12/10/2019) (cit. on p. 70).

[31] *AWS EC2 pricing on demand*. 2019. URL: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 12/10/2019) (cit. on p. 70).

[32] *AWS Lambda Layers.* 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html (visited on 12/10/2019) (cit. on p. 40).

[33] 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: 9789811050268. DOI: 10.1007/978-981-10-5026-8_1. arXiv: 1706.03178 (cit. on p. 30).

[34] 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*. New York, New York, USA: ACM Press, 2017, pp. 89–103. ISBN: 9781450355308. DOI: 10.1145/3133850.3133855. arXiv: 1611.02756 (cit. on p. 30).

[35] Vinit Baliyan et al. “Diffusion weighted imaging: Technique and applications”. In: *world journal of radiology* 8 (Sept. 2016), pp. 785–798. DOI: 10.4329/wjr.v8.i9.785 (cit. on p. 68).

[36] Bitnami. *Docker Hub: bitnami/minideb*. URL: https://hub.docker.com/r/bitnami/minideb/ (visited on 06/20/2019) (cit. on p. 49).

[37] Bitnami. *Kubeless*. URL: https://kubeless.io/ (visited on 06/19/2019) (cit. on pp. 13, 24).

[38] Miguel Caballer et al. “Dynamic Management of Virtual Infrastructures”. In: *Journal of Grid Computing* 13.1 (Mar. 2015), pp. 53–70. ISSN: 15729184. DOI: 10.1007/s10723-014-9296-5 (cit. on pp. 8, 78).

[39] Miguel Caballer et al. “EC3: Elastic cloud computing cluster”. In: *Journal of Computer and System Sciences* 79.8 (2013), pp. 1341–1351. ISSN: 00220000. DOI: 10.1016/j.jcss.2013.06.005 (cit. on p. 78).

[40] 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 (cit. on p. 78).

[41] Canonical Ltd. *LXD*. URL: https://linuxcontainers.org/ (visited on 09/23/2019) (cit. on p. 9).

[42] Paul Castro et al. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54. ISSN: 00010782. DOI: 10.1145/3368454. URL: http://dl.acm.org/citation.cfm?doid=3372896.3368454 (cit. on p. 30).

[43] CentOS. *Docker Hub: centos:7*. URL: https://hub.docker.com/_/centos/ (visited on 06/20/2019) (cit. on pp. 47, 49).

[44] CloudFoundry. *CloudFoundry*. URL: https://www.cloudfoundry.org/ (visited on 09/20/2019) (cit. on p. 9).

[45] Cloudify. *Cloudify*. URL: https://cloudify.co/ (visited on 09/20/2019) (cit. on p. 8).

[46] CNCF. *CNCF*. URL: https://www.cncf.io/ (visited on 09/24/2019) (cit. on p. 12).

[47] CNCF. "CNCF WG-Serverless Whitepaper v1.0". URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (cit. on p. 12).

[48] *Comparison of C/POSIX standard library implementations for Linux*. 2020. URL: http://www.etalabs.net/compare_libcs.html (visited on 03/02/2020) (cit. on p. 46).

[49] Adaptive Computing. *TORQUE Resource Manager*. URL: http://www.adaptivecomputing.com/products/torque/ (cit. on p. 72).

[50] D2SI Open Source Platform. *Ooso, serverless mapreduce*. URL: https://kubeless.io/ (visited on 06/19/2019) (cit. on p. 28).

[51] Deep. *Deep Hybrid Datacloud*. URL: https://deep-hybrid-datacloud.eu/ (visited on 10/04/2019) (cit. on p. 108).

[52] Docker. *Docker*. URL: https://www.docker.com/ (visited on 06/19/2019) (cit. on pp. 2, 9).

[53] Docker. *Docker Hub*. URL: https://hub.docker.com/ (visited on 06/20/2019) (cit. on p. 38).

[54] Docker. *Docker Swarm*. URL: https://docs.docker.com/engine/swarm/ (visited on 09/22/2019) (cit. on pp. 2, 10).

[55] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.0 (June 2016). arXiv: 1606.04036 (cit. on p. 2).

[56] *EC2 Compute Optimized*. 2019. URL: https://aws.amazon.com/ec2/instance-types/#Compute_Optimized (visited on 12/05/2019) (cit. on p. 90).

[57] EGI Foundation. *EGI Federated Cloud*. URL: https://www.egi.eu/federation/egi-federated-cloud/ (visited on 06/19/2019) (cit. on p. 94).

[58] EGI Foundation. *EGI Federated Data*. URL: https://wiki.egi.eu/wiki/EGI_Federated_Data (visited on 10/03/2019) (cit. on p. 108).

[59] Adam Eivy. "Be Wary of the Economics of 'Serverless' Cloud Computing". In: *IEEE Cloud Computing* 4.2 (Mar. 2017), pp. 6–12. ISSN: 23256095. DOI: 10.1109/MCC.2017.32 (cit. on p. 30).

[60] *Explora OSCAR*. 2019. URL: https://aplicat.upv.es/exploraupv/ficha-tecnologia/patente_software/27824 (visited on 03/10/2012) (cit. on p. 119).

[61] *Explora SCAR*. 2019. URL: https://aplicat.upv.es/exploraupv/ficha-tecnologia/patente_software/24666 (visited on 10/07/2019) (cit. on p. 119).

[62] Erwin van Eyk et al. "The SPEC cloud group's research vision on FaaS and serverless architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*. New York, New York,

USA: ACM Press, Nov. 2017, pp. 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848 (cit. on pp. 3, 30).

[63] Enol Fernández-Del-Castillo, Diego Scardaci, and Álvaro López García. "The EGI Federated Cloud e-Infrastructure". In: *Procedia Computer Science*. Vol. 68. Elsevier, Jan. 2015, pp. 196–205. DOI: 10.1016/j.procs.2015.09.235 (cit. on pp. 94, 120).

[64] Flexera. *State of the Cloud Report 2019*. URL: https://www.flexera.com/blog/cloud/2019/02/cloud-computing-trends-2019-state-of-the-cloud-survey/ (visited on 10/22/2019) (cit. on p. 1).

[65] Fn. *FN Project*. URL: https://fnproject.io/ (cit. on pp. 13, 26).

[66] Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *USENIX NSDI*. Boston, MA: USENIX Association, 2017, pp. 363–376. ISBN: 9781931971379 (cit. on p. 29).

[67] Geoffrey C Fox et al. *Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research*. Tech. rep. Aug. 2017. DOI: 10.13140/RG.2.2.15007.87206. arXiv: 1708.08028 (cit. on p. 13).

[68] Funktion. *Funktion*. URL: https://funktion.fabric8.io/ (cit. on pp. 13, 27).

[69] Dennis Gannon. *Observations about Serverless Computing With a few examples from AWS Lambda, Azure Functions and Open Whisk*. Tech. rep. 2017. DOI: 10.13140/RG.2.2.30281.03685 (cit. on p. 13).

[70] 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* 97 (Mar. 2019), pp. 259–274. ISSN: 0167739X. DOI: 10.1016/j.future.2019.02.057 (cit. on pp. 29, 53).

[71] Alex Glikson. *TRANSIT: Flexible pipeline for IoT data with Bluemix and OpenWhisk*. URL: https://medium.com/openwhisk/transit-

flexible-pipeline-for-iot-data-with-bluemix-and-openwhisk-4824cf20f1e0 (cit. on p. 29).

[72] Alex Glikson, Stefan Nastic, and Schahram Dustdar. "Deviceless edge computing". In: *Proceedings of the 10th ACM International Systems and Storage Conference on - SYSTOR '17*. SYSTOR '17. New York, New York, USA: ACM Press, 2017, pp. 1–1. ISBN: 9781450350358. DOI: 10.1145/3078468.3078497 (cit. on p. 29).

[73] Jorge Gomes. *uDocker Documentation*. URL: https://github.com/indigo-dc/udocker/blob/master/doc/user_manual.md (visited on 06/20/2019) (cit. on p. 38).

[74] Jorge Gomes et al. "Enabling rootless Linux Containers in multi-user environments: The udocker tool". In: *Computer Physics Communications* 232 (Nov. 2018), pp. 84–97. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.05.021 (cit. on p. 38).

[75] Google. *Firebase*. URL: https://firebase.google.com/ (visited on 09/23/2019) (cit. on p. 11).

[76] Google. *G Suite*. URL: https://gsuite.google.com/ (visited on 09/20/2019) (cit. on p. 9).

[77] Google. *Google App Engine*. URL: https://cloud.google.com/appengine (visited on 09/20/2019) (cit. on p. 9).

[78] Google. *Google Cloud Functions*. URL: https://cloud.google.com/functions/ (visited on 06/19/2019) (cit. on pp. 12, 15).

[79] Google. *Google Cloud Platform*. URL: https://cloud.google.com (visited on 06/19/2019) (cit. on p. 2).

[80] Google. *Google Cloud Run*. URL: https://cloud.google.com/run/ (cit. on p. 15).

[81] Google. *Google Cloud Run: Container Contract.* URL: https://cloud.google.com/run/docs/reference/container-contract (cit. on p. 15).

[82] Google. *Google Compute Engine.* URL: https://cloud.google.com/compute/ (visited on 10/11/2019) (cit. on p. 8).

[83] Google. *Google Kubernetes Engine.* URL: https://cloud.google.com/kubernetes-engine/ (visited on 09/23/2019) (cit. on pp. 3, 10).

[84] Google. *Knative.* URL: https://cloud.google.com/knative/ (visited on 09/27/2019) (cit. on pp. 13, 15, 24).

[85] GRyCAP. *CNCF: Scar.* 2019. URL: https://landscape.cncf.io/format=serverless{\&}selected=scar (visited on 10/04/2019) (cit. on p. 119).

[86] GRyCAP. *Docker Hub: grycap/jenkins:ubuntu14.04-python.* URL: https://hub.docker.com/r/grycap/jenkins (visited on 06/20/2019) (cit. on p. 49).

[87] GRycAP. *faas-supervisor.* URL: https://github.com/grycap/faas-supervisor (cit. on p. 84).

[88] GRyCAP. *Minicon.* URL: https://github.com/grycap/minicon (cit. on p. 46).

[89] GRyCAP. *Resource and Application Description Language (RADL).* URL: https://imdocs.readthedocs.io/en/latest/radl.html (visited on 06/19/2019) (cit. on p. 78).

[90] GRyCAP. *SERVERLESS COMPUTING FOR DATA-PROCESSING APPLICATIONS ON MULTI-CLOUDS (HANDS-ON TUTORIAL).* URL: https://indico.lip.pt/event/575/contributions/2002/ (visited on 10/03/2019) (cit. on p. 109).

[91] Hashicorp. *Consul.* URL: https://www.consul.io/ (visited on 09/23/2019) (cit. on p. 11).

[92] Hashicorp. *Nomad*. URL: https://www.nomadproject.io/ (visited on 09/22/2019) (cit. on pp. 3, 10).

[93] Hashicorp. *Vault*. URL: https://www.vaultproject.io/ (visited on 09/23/2019) (cit. on p. 11).

[94] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: (2018). arXiv: 1812.03651. URL: http://arxiv.org/abs/1812.03651 (cit. on p. 30).

[95] Scott Hendrickson et al. "Serverless Computing with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing*. HotCloud '16. Denver, CO: USENIX Association, 2016 (cit. on p. 28).

[96] Heroku. *Heroku*. URL: https://www.heroku.com/ (visited on 09/20/2019) (cit. on p. 9).

[97] *High Throughput Computing*. 2019. URL: https://wiki.egi.eu/wiki/Glossary_V1#High_Throughput_Computing (visited on 12/05/2019) (cit. on p. 5).

[98] Ibergrid. *On-premises Serverless Container-aware ARchitectures*. 2018. URL: https://indico.lip.pt/event/437/contributions/1408/ (visited on 10/04/2019) (cit. on p. 120).

[99] Ibergrid. *Serverless Computing for Data-Processing Across Public and Federated Clouds*. 2019. URL: https://indico.lip.pt/event/575/contributions/1850/ (cit. on p. 120).

[100] IBM. *IBM Cloud Functions*. URL: https://www.ibm.com/cloud/functions (visited on 09/19/2019) (cit. on pp. 12, 15).

[101] IBM. *OpenWhisk Composer*. URL: https://github.com/ibm-functions/composer (visited on 09/27/2019) (cit. on p. 27).

[102] INDIGO-DataCloud Collaboration. "INDIGO-DataCloud:A data and computing platform to facilitate seamless access to e-infrastructures". In: (Nov. 2017). arXiv: 1711.01981 (cit. on p. 38).

[103] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "Serving deep learning models in a serverless platform". In: (2017). arXiv: 1710.08460. URL: http://arxiv.org/abs/1710.08460 (cit. on p. 29).

[104] *Istio*. URL: https://istio.io/ (visited on 09/27/2019) (cit. on p. 24).

[105] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: (2019). arXiv: 1902.03383. URL: http://arxiv.org/abs/1902.03383 (cit. on p. 30).

[106] Eric Jonas et al. "Occupy the Cloud: Distributed Computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing - SoCC '17* (Feb. 2017), pp. 445–451. DOI: 10.1145/3127479.3128601. arXiv: 1702.04024 (cit. on p. 29).

[107] JS Foundation. *Node-RED*. URL: https://nodered.org/ (visited on 06/20/2019) (cit. on p. 29).

[108] kaniko. *kaniko*. URL: https://github.com/GoogleContainerTools/kaniko (cit. on p. 79).

[109] Kubernetes. *Kubernetes Horizontal Pod Autoscaler*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (cit. on pp. 24, 77).

[110] Kubernetes. *Kubernetes Vertical Pod Autoscaler*. URL: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler (cit. on p. 77).

[111] Kubernetes. *KubernetesCluster Autoscaler*. URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler (cit. on p. 77).

[112] *Kubernetes: Case studies*. URL: https://kubernetes.io/case-studies/ (cit. on p. 10).

[113] Grafana Labs. *Grafana*. URL: https://grafana.com/ (cit. on p. 26).

[114] Linux Fundation. *Kubernetes*. URL: https://kubernetes.io/ (visited on 09/22/2019) (cit. on pp. 3, 10).

[115] Wes Lloyd et al. “Serverless Computing: An Investigation of Factors Influencing Microservice Performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169. ISBN: 978-1-5386-5008-0. DOI: 10.1109/IC2E.2018.00039. URL: https://ieeexplore.ieee.org/document/8360324/ (cit. on p. 29).

[116] Theo Lynn et al. “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Vol. 2017-Decem. IEEE, Dec. 2017, pp. 162–169. ISBN: 978-1-5386-0692-6. DOI: 10.1109/CloudCom.2017.15 (cit. on p. 36).

[117] Maciej Malawski. “Towards Serverless Execution of Scientific Workflows - HyperFlow Case Study”. In: *undefined* (2016). URL: https://www.semanticscholar.org/paper/Towards-Serverless-Execution-of-Scientific-Case-Malawski/1117a065a291a06b066ab6c6af17aa620bf2b589 (cit. on p. 29).

[118] Garrett McGrath et al. “Cloud Event Programming Paradigms: Applications and Analysis”. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, June 2016, pp. 400–406. ISBN: 978-1-5090-2619-7. DOI: 10.1109/CLOUD.2016.0060 (cit. on p. 13).

[119] P M Mell and Tim Grance. *The NIST definition of cloud computing*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, 2011. DOI: 10.6028/NIST.SP.800-145 (cit. on p. 7).

[120] Adolfo Méndez Madrigal. *Ejecución Monetizada de Workflows de Algoritmos Privados en Plataformas Serverless Públicas*. 2019. URL: http://hdl.handle.net/10251/129147 (cit. on p. 114).

[121] Ivan Merelli et al. “Exploiting Docker containers over Grid computing for a comprehensive study of chromatin conformation in different cell types”. In: *Journal of Parallel and Distributed Computing* 134 (2019), pp. 116 –127. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.

jpdc.2019.08.002. URL: http://www.sciencedirect.com/science/article/pii/S0743731519305593 (cit. on p. 38).

[122] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment”. In: *Linux Journal* 2014.239 (2014), pp. 76–90. ISSN: 1075-3583 (cit. on p. 9).

[123] Microsoft. *ACI: azure file share.* URL: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-volume-azure-files (cit. on p. 15).

[124] Microsoft. *ACI: GPU access.* URL: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-gpu (cit. on p. 15).

[125] Microsoft. *Azure Container Instances.* URL: https://azure.microsoft.com/services/container-instances/ (cit. on p. 15).

[126] Microsoft. *Azure Container Service.* URL: https://azure.microsoft.com/services/container-service/ (visited on 06/19/2019) (cit. on p. 3).

[127] Microsoft. *Azure Event Hubs.* URL: https://azure.microsoft.com/services/event-hubs/ (cit. on p. 20).

[128] Microsoft. *Azure Files.* URL: https://azure.microsoft.com/services/storage/files/ (visited on 10/11/2019) (cit. on p. 19).

[129] Microsoft. *Azure IaaS.* URL: https://azure.microsoft.com/en-us/overview/what-is-azure/iaas/ (visited on 10/11/2019) (cit. on p. 8).

[130] Microsoft. *Azure Mobile.* URL: https://azure.microsoft.com/en-us/solutions/mobile/ (visited on 09/23/2019) (cit. on p. 11).

[131] Microsoft. *Azure Service Fabric.* URL: https://azure.microsoft.com/en-us/services/service-fabric/ (visited on 11/27/2019) (cit. on p. 11).

[132] Microsoft. *Azure Service Fabric Programming Model.* URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework (visited on 11/27/2019) (cit. on p. 11).

[133] Microsoft. *Microsoft AKS.* URL: https://azure.microsoft.com/en-us/services/kubernetes-service/ (visited on 09/23/2019) (cit. on p. 10).

[134] Microsoft. *Microsoft Azure.* URL: https://azure.microsoft.com (visited on 06/19/2019) (cit. on p. 2).

[135] Microsoft. *Microsoft Azure Functions.* URL: https://azure.microsoft.com/en-in/services/functions/ (visited on 06/19/2019) (cit. on pp. 12, 15).

[136] Microsoft. *Office 365.* URL: https://www.office.com/ (visited on 09/20/2019) (cit. on p. 9).

[137] MinIO. *MinIO.* URL: https://min.io/ (cit. on p. 79).

[138] Nuclio. *Nuclio.* URL: https://nuclio.io/ (visited on 09/27/2019) (cit. on pp. 13, 25).

[139] Edward Oakes et al. "Pipsqueak: Lean Lambdas with Large Libraries". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW).* IEEE, June 2017, pp. 395–400. ISBN: 978-1-5386-3292-5. DOI: 10.1109/ICDCSW.2017.32 (cit. on p. 28).

[140] OASIS. *TOSCA Simple Profile in YAML Version 1.1.* URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html (visited on 06/19/2019) (cit. on p. 78).

[141] Occopus. *Occopus.* URL: https://occopus.lpds.sztaki.hu/home (cit. on p. 8).

[142] Onedata. *Onedata*. URL: https://onedata.org (visited on 10/01/2019) (cit. on pp. 84, 108).

[143] OpenFaaS. *OpenFaaS*. URL: https://www.openfaas.com/ (visited on 09/28/2019) (cit. on pp. 13, 23, 79).

[144] OpenFaaS. *OpenFaaS connector SDK*. URL: https://github.com/openfaas-incubator/connector-sdk (visited on 09/28/2019) (cit. on p. 23).

[145] OpenNebula. *OpenNebula*. URL: https://opennebula.org (visited on 06/19/2019) (cit. on pp. 2, 8).

[146] OpenStack. *Aodh*. URL: https://github.com/openstack/aodh (visited on 09/28/2019) (cit. on p. 28).

[147] OpenStack. *Keystone*. URL: https://github.com/openstack/keystone (visited on 09/28/2019) (cit. on p. 27).

[148] OpenStack. *OpenStack*. URL: http://openstack.org (visited on 06/19/2019) (cit. on pp. 2, 8).

[149] OpenStack. *Qinling*. URL: https://github.com/openstack/qinling (visited on 09/28/2019) (cit. on pp. 13, 27).

[150] OpenStack. *Swift*. URL: https://github.com/openstack/swift (visited on 09/28/2019) (cit. on p. 27).

[151] OpenStack. *Zaqar*. URL: https://github.com/openstack/zaqar (visited on 09/28/2019) (cit. on p. 28).

[152] OpenStack. *Zun*. URL: https://github.com/openstack/zun (visited on 09/28/2019) (cit. on p. 27).

[153] Greg Owen et al. *Databricks Serverless: Next Generation Resource Management for Apache Spark*. 2017. URL: https://databricks.com/blog/2017/06/07/databricks-serverless-

next-generation-resource-management-for-apache-spark.html (visited on 06/19/2019) (cit. on p. 28).

[154] 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 (cit. on pp. 34, 68).

[155] Pivotal. *Riff*. URL: https://projectriff.io/ (cit. on pp. 13, 24, 26).

[156] Platform9. *Fission*. URL: https://fission.io/ (visited on 06/19/2019) (cit. on pp. 13, 25).

[157] Platform9. *Platform9*. URL: https://platform9.com/ (visited on 09/27/2019) (cit. on p. 25).

[158] *Prometheus*. URL: https://prometheus.io (visited on 09/27/2019) (cit. on p. 24).

[159] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. 2013. URL: http://pjreddie.com/darknet/ (visited on 06/20/2019) (cit. on p. 99).

[160] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *CoRR* abs/1612.0 (Dec. 2016). arXiv: 1612.08242 (cit. on p. 99).

[161] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018). arXiv: 1804.02767 (cit. on pp. 86, 104).

[162] Chris Richards. *Microservices Patterns WITH EXAMPLES IN JAVA*. 1st editio. Vol. 2018. March. Manning Publications, 2018, pp. 1–3. ISBN: 9781617294549. URL: https://b-ok.cc/book/3620439/ea5ed9 (cit. on pp. 9, 10).

[163] Chris Richardson. *Microservices Architetures*. URL: https://microservices.io/patterns/microservices.html (visited on 11/26/2019) (cit. on p. 9).

[164] Sebastián Risco Gallardo. "Plataforma Serverless Híbrida de Procesado de Datos". PhD thesis. Sept. 2019 (cit. on p. 112).

[165] JPiotr Roszatycki. *Fakechroot*. URL: `https://github.com/dex4er/fakechroot` (visited on 06/20/2019) (cit. on p. 38).

[166] Keith Rozario. *Multiprocessing in Lambda functions*. URL: `https://www.keithrozario.com/2019/10/multiprocessing-in-lambda-functions.html` (cit. on p. 16).

[167] SchedMD. *SLURM Workload Manager*. URL: `https://slurm.schedmd.com/` (cit. on p. 72).

[168] Serverless. *Serverless*. URL: `https://serverless.com/` (visited on 09/28/2019) (cit. on p. 24).

[169] Vaishaal Shankar et al. "numpywren: serverless linear algebra". In: (Oct. 2018). arXiv: `1810.09679` (cit. on p. 29).

[170] Josef Spillner. "Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation". In: *CoRR* abs/1703.0 (2017). arXiv: `1703.07562` (cit. on p. 28).

[171] Josef Spillner and Serhii Dorodko. "Java Code Analysis and Transformation into AWS Lambda Functions". In: *CoRR* abs/1702.0 (2017). arXiv: `1702.05510` (cit. on p. 28).

[172] 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. Springer, Cham, 2018, pp. 154–168. ISBN: 9783319733524. DOI: `10.1007/978-3-319-73353-1_11` (cit. on p. 29).

[173] Tekton. *Tekton Pipelines*. URL: `https://github.com/tektoncd/pipeline` (visited on 09/28/2019) (cit. on p. 24).

[174] The Apache Software Foundation. *Apache Kafka*. URL: `https://kafka.apache.org/documentation/streams/` (cit. on p. 26).

[175] The Apache Software Foundation. *Apache Marathon*. URL: `https://mesosphere.github.io/marathon/` (cit. on p. 10).

[176] The Apache Software Foundation. *Apache Mesos*. URL: `https://mesos.apache.org` (visited on 09/22/2019) (cit. on p. 10).

[177] The Apache Software Foundation. *Apache OpenWhisk*. URL: `http://openwhisk.incubator.apache.org/` (visited on 06/19/2019) (cit. on pp. 12, 13, 27).

[178] The World Air Quality Project. *The World Air Quality Index*. URL: `http://waqi.info` (visited on 10/03/2019) (cit. on p. 112).

[179] Ferran Borreguero Torro et al. "Accelerating the Diffusion-Weighted Imaging Biomarker in the clinical practice: Comparative study". In: *Procedia Computer Science*. Vol. 108. Supplement C. 2017, pp. 1185–1194. DOI: `10.1016/j.procs.2017.05.108` (cit. on p. 67).

[180] UDocker. *uDocker*. URL: `https://github.com/indigo-dc/udocker` (visited on 06/19/2019) (cit. on p. 38).

[181] André Vieira. *GROMACS AWS Batch*. 2019. URL: `https://indico.lip.pt/event/575/contributions/1843/` (visited on 10/07/2019) (cit. on p. 117).

[182] Matthew Viljoen et al. "Towards European Open Science Commons: The EGI Open Data Platform and the EGI DataHub". In: *Procedia Computer Science* 97 (2016). 2nd International Conference on Cloud Forward: From Distributed to Complete Computing, pp. 148 –152. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2016.08.294`. URL: `http://www.sciencedirect.com/science/article/pii/S187705091632110X` (cit. on p. 108).

[183] Mario Villamizar et al. "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11.2 (2017), pp. 233–247. ISSN: 1863-2394. DOI: `10.1007/s11761-017-0208-y` (cit. on p. 28).

[184] Virtuozzo. *OpenVZ*. URL: https://openvz.org (visited on 09/23/2019) (cit. on p. 9).

[185] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 133–146. ISBN: 978-1-931971-44-7 (cit. on pp. 44, 53, 68).

[186] Sebastian Werner et al. “Serverless Big Data Processing using Matrix Multiplication as Example”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 358–365. ISBN: 978-1-5386-5035-6. DOI: 10.1109/BigData.2018.8622362. URL: https://ieeexplore.ieee.org/document/8622362/ (cit. on p. 29).

[187] University of Wisconsin-Madison. *HTCondor*. URL: https://research.cs.wisc.edu/htcondor/ (cit. on p. 72).

[188] Mengting Yan et al. “Building a Chatbot with Serverless Computing”. In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs - MOTA ’16*. MOTA ’16. New York, New York, USA: ACM Press, Dec. 2016, pp. 1–4. ISBN: 9781450346696. DOI: 10.1145/3007203.3007217 (cit. on p. 29).

Appendix A

SCAR client commands

In the following pages we present the SCAR command lines as described in the CLI help. First, a general help describing all the available commands is presented.

```
usage: scar [-h] [--version] {init,invoke,run,update,rm,ls,log,put,get}
     ...

Deploy containers in serverless architectures

optional arguments:
  -h, --help            show this help message and exit
  --version             Show SCAR version.

Commands:
  {init,invoke,run,update,rm,ls,log,put,get}
    init                Create lambda function
    invoke              Call a lambda function using an HTTP request
    run                 Deploy function
    update              Update function properties
    rm                  Delete function
    ls                  List lambda functions
    log                 Show the logs for the lambda function
    put                 Upload file(s) to bucket
    get                 Download file(s) from bucket

Run 'scar COMMAND --help' for more information on a command.
```

To control the complete lifecycle of the functions we can use the following commands: 'init', 'invoke', 'run', 'rm', and 'update'. The 'init' command is the most important one, because it automatizes all the function deployment process (e.g. package the required libraries, creation of the required layers, etc) and the linking with other services (e.g. creation of the API Gateway endpoint, creation of the required AWS S3 buckets and folders and linking them with the Lambda function, creation of the CloudWatch Log groups, etc).

```
usage: scar init [-h] [-d DESCRIPTION] [-e ENVIRONMENT]
                 [-le LAMBDA_ENVIRONMENT] [-m MEMORY] [-t TIME]
                 [-tt TIMEOUT_THRESHOLD] [-ll LOG_LEVEL] [-l LAYERS]
                 [-ib INPUT_BUCKET] [-ob OUTPUT_BUCKET] [-em
                     EXECUTION_MODE]
                 [-r IAM_ROLE] [-sv SUPERVISOR_VERSION] [-bm
                     BATCH_MEMORY]
                 [-bc BATCH_VCPUS] [-g] [-j] [-v] [-pf PROFILE]
                 (-i IMAGE | -if IMAGE_FILE | -f CONF_FILE) [-n NAME]
                 [-s INIT_SCRIPT] [-ph] [-ep EXTRA_PAYLOAD]
                 [-db DEPLOYMENT_BUCKET] [-api API_GATEWAY_NAME]

optional arguments:
  -h, --help            show this help message and exit
  -d DESCRIPTION, --description DESCRIPTION
                        Lambda function description.
  -e ENVIRONMENT, --environment ENVIRONMENT
                        Pass environment variable to the container (VAR=
                            val).
                        Can be defined multiple times.
  -le LAMBDA_ENVIRONMENT, --lambda-environment LAMBDA_ENVIRONMENT
                        Pass environment variable to the lambda function
                        (VAR=val). Can be defined multiple times.
  -m MEMORY, --memory MEMORY
                        Lambda function memory in megabytes. Range from
                            128 to
                        3008 in increments of 64
  -t TIME, --time TIME  Lambda function maximum execution time in
      seconds. Max
                        900.
  -tt TIMEOUT_THRESHOLD, --timeout-threshold TIMEOUT_THRESHOLD
                        Extra time used to postprocess the data. This
                            time is
                        extracted from the total time of the lambda
                            function.
  -ll LOG_LEVEL, --log-level LOG_LEVEL
                        Set the log level of the lambda function.
                            Accepted
                        values are:
                        'CRITICAL', 'ERROR', 'WARNING', 'INFO', 'DEBUG'
  -l LAYERS, --layers LAYERS
                        Pass layers ARNs to the lambda function. Can be
                        defined multiple times.
  -ib INPUT_BUCKET, --input-bucket INPUT_BUCKET
```

```
                        Bucket name where the input files will be stored
                            .
  -ob OUTPUT_BUCKET, --output-bucket OUTPUT_BUCKET
                        Bucket name where the output files are saved.
  -em EXECUTION_MODE, --execution-mode EXECUTION_MODE
                        Specifies the execution mode of the job. It can
                            be
                        'lambda', 'lambda-batch' or 'batch'
  -r IAM_ROLE, --iam-role IAM_ROLE
                        IAM role used in the management of the functions
  -sv SUPERVISOR_VERSION, --supervisor-version SUPERVISOR_VERSION
                        FaaS Supervisor version. Can be a tag or 'latest
                            '.
  -bm BATCH_MEMORY, --batch-memory BATCH_MEMORY
                        Batch job memory in megabytes
  -bc BATCH_VCPUS, --batch-vcpus BATCH_VCPUS
                        Number of vCPUs reserved for the Batch container
  -g, --enable-gpu      Reserve one physical GPU for the Batch container
      (if
                        it's available in the compute environment)
  -j, --json            Return data in JSON format
  -v, --verbose         Show the complete aws output in json format
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -i IMAGE, --image IMAGE
                        Container image id (i.e. centos:7)
  -if IMAGE_FILE, --image-file IMAGE_FILE
                        Container image file created with 'docker save'
                            (i.e.
                        centos.tar.gz)
  -f CONF_FILE, --conf-file CONF_FILE
                        Yaml file with the function configuration
  -n NAME, --name NAME  Lambda function name
  -s INIT_SCRIPT, --init-script INIT_SCRIPT
                        Path to the input file passed to the function
  -ph, --preheat        Preheats the function running it once and
      downloading
                        the necessary container
  -ep EXTRA_PAYLOAD, --extra-payload EXTRA_PAYLOAD
                        Folder containing files that are going to be
                            added to
                        the lambda function
  -db DEPLOYMENT_BUCKET, --deployment-bucket DEPLOYMENT_BUCKET
                        Bucket where the deployment package is going to
                            be
                        uploaded.
  -api API_GATEWAY_NAME, --api-gateway-name API_GATEWAY_NAME
                        API Gateway name created to launch the lambda
                            function
```

After the function is created, we can launch it in different ways. Directly through the ‘invoke’ or ‘run’ commands or indirectly by uploading a file to

an S3 bucket with the ‘put’ command. The difference between ‘invoke’ and ‘run’ lies in the invocation method used, that is, ‘invoke’ allow us to launch a Lambda function with an API endpoint defined, and ‘run’ will call directly the function using a python library. You cannot ‘invoke’ a function without an API endpoint, but you can ‘run’ a function defined with an endpoint.

```
usage: scar invoke [-h] [-pf PROFILE] [-a] [-o OUTPUT_FILE]
                   (-n NAME | -f CONF_FILE) [-db DATA_BINARY] [-jd
                       JSON_DATA]
                   [-p PARAMETERS]

optional arguments:
  -h, --help            show this help message and exit
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -a, --asynchronous    Launch an asynchronous function.
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        Save output as a file
  -n NAME, --name NAME  Lambda function name
  -f CONF_FILE, --conf-file CONF_FILE
                        Yaml file with the function configuration
  -db DATA_BINARY, --data-binary DATA_BINARY
                        File path of the HTTP data to POST.
  -jd JSON_DATA, --json-data JSON_DATA
                        JSON Body to Post
  -p PARAMETERS, --parameters PARAMETERS
                        In addition to passing the parameters in the URL
                            , you
                        can pass the parameters here (i.e. '{"key1": "
                            value1",
                        "key2": ["value2", "value3"]}').
```

```
usage: scar run [-h] [-j] [-v] [-pf PROFILE] [-a] [-o OUTPUT_FILE]
                (-n NAME | -f CONF_FILE) [-s RUN_SCRIPT]
                ...

positional arguments:
  c_args                Arguments passed to the container.

optional arguments:
  -h, --help            show this help message and exit
  -j, --json            Return data in JSON format
  -v, --verbose         Show the complete aws output in json format
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -a, --asynchronous    Launch an asynchronous function.
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        Save output as a file
  -n NAME, --name NAME  Lambda function name
  -f CONF_FILE, --conf-file CONF_FILE
                        Yaml file with the function configuration
  -s RUN_SCRIPT, --run-script RUN_SCRIPT
```

```
                        Path to the input file passed to the function
```

The 'rm' command is in charge of deleting the Lambda function and all the links created (if it has any). The AWS S3 buckets and folders created with the 'init' command are not deleted with the 'rm'. This is done to guarantee the data persistence and to avoid deleting it by mistake.

```
usage: scar rm [-h] [-j] [-v] [-pf PROFILE] (-n NAME | -a | -f CONF_FILE
    )

optional arguments:
  -h, --help            show this help message and exit
  -j, --json            Return data in JSON format
  -v, --verbose         Show the complete aws output in json format
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -n NAME, --name NAME  Lambda function name
  -a, --all             Delete all lambda functions
  -f CONF_FILE, --conf-file CONF_FILE
                        Yaml file with the function configuration
```

The 'update' command allow us to update the AWS Lambda properties without having to deploy it again. This command cannot create or delete new AWS S3 buckets or folders and AWS API endpoints. It has only been designed to allow the users modify the intrinsic properties of the Lambda function.

```
usage: scar update [-h] [-d DESCRIPTION] [-e ENVIRONMENT]
                   [-le LAMBDA_ENVIRONMENT] [-m MEMORY] [-t TIME]
                   [-tt TIMEOUT_THRESHOLD] [-ll LOG_LEVEL] [-l LAYERS]
                   [-ib INPUT_BUCKET] [-ob OUTPUT_BUCKET] [-em
                       EXECUTION_MODE]
                   [-r IAM_ROLE] [-sv SUPERVISOR_VERSION] [-bm
                       BATCH_MEMORY]
                   [-bc BATCH_VCPUS] [-g] [-j] [-v] [-pf PROFILE]
                   (-n NAME | -a | -f CONF_FILE)

optional arguments:
  -h, --help            show this help message and exit
  -d DESCRIPTION, --description DESCRIPTION
                        Lambda function description.
  -e ENVIRONMENT, --environment ENVIRONMENT
                        Pass environment variable to the container (VAR=
                            val).
                        Can be defined multiple times.
  -le LAMBDA_ENVIRONMENT, --lambda-environment LAMBDA_ENVIRONMENT
                        Pass environment variable to the lambda function
                        (VAR=val). Can be defined multiple times.
  -m MEMORY, --memory MEMORY
```

```
                         Lambda function memory in megabytes. Range from
                             128 to
                         3008 in increments of 64
  -t TIME, --time TIME   Lambda function maximum execution time in
      seconds. Max
                         900.
  -tt TIMEOUT_THRESHOLD, --timeout-threshold TIMEOUT_THRESHOLD
                         Extra time used to postprocess the data. This
                             time is
                         extracted from the total time of the lambda
                             function.
  -ll LOG_LEVEL, --log-level LOG_LEVEL
                         Set the log level of the lambda function.
                             Accepted
                         values are:
                         'CRITICAL','ERROR','WARNING','INFO','DEBUG'
  -l LAYERS, --layers LAYERS
                         Pass layers ARNs to the lambda function. Can be
                         defined multiple times.
  -ib INPUT_BUCKET, --input-bucket INPUT_BUCKET
                         Bucket name where the input files will be stored
                             .
  -ob OUTPUT_BUCKET, --output-bucket OUTPUT_BUCKET
                         Bucket name where the output files are saved.
  -em EXECUTION_MODE, --execution-mode EXECUTION_MODE
                         Specifies the execution mode of the job. It can
                             be
                         'lambda', 'lambda-batch' or 'batch'
  -r IAM_ROLE, --iam-role IAM_ROLE
                         IAM role used in the management of the functions
  -sv SUPERVISOR_VERSION, --supervisor-version SUPERVISOR_VERSION
                         FaaS Supervisor version. Can be a tag or 'latest
                             '.
  -bm BATCH_MEMORY, --batch-memory BATCH_MEMORY
                         Batch job memory in megabytes
  -bc BATCH_VCPUS, --batch-vcpus BATCH_VCPUS
                         Number of vCPUs reserved for the Batch container
  -g, --enable-gpu       Reserve one physical GPU for the Batch container
       (if
                         it's available in the compute environment)
  -j, --json             Return data in JSON format
  -v, --verbose          Show the complete aws output in json format
  -pf PROFILE, --profile PROFILE
                         AWS profile to use
  -n NAME, --name NAME   Lambda function name
  -a, --all              Update all lambda functions
  -f CONF_FILE, --conf-file CONF_FILE
                         Yaml file with the function configuration
```

The ‘ls’ command allow us to list the defined functions created with SCAR. In addition, it allow us to list the contents of an AWS S3 bucket or a folder in a bucket.

```
usage: scar ls [-h] [-j] [-v] [-pf PROFILE] [-b BUCKET] [-l]

optional arguments:
  -h, --help            show this help message and exit
  -j, --json            Return data in JSON format
  -v, --verbose         Show the complete aws output in json format
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -b BUCKET, --bucket BUCKET
                        Show bucket files
  -l, --list-layers     Show lambda layers information
```

With the 'log' command we can check the execution's traces of the functions. This command also allo us to filter based on the *log stream name* and the *request id*.

```
usage: scar log [-h] [-pf PROFILE] (-n NAME | -f CONF_FILE)
                [-ls LOG_STREAM_NAME] [-ri REQUEST_ID]

optional arguments:
  -h, --help            show this help message and exit
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
  -n NAME, --name NAME  Lambda function name
  -f CONF_FILE, --conf-file CONF_FILE
                        Yaml file with the function configuration
  -ls LOG_STREAM_NAME, --log-stream-name LOG_STREAM_NAME
                        Return the output for the log stream specified.
  -ri REQUEST_ID, --request-id REQUEST_ID
                        Return the output for the request id specified.
```

Both 'put' and 'get' commands are used to manage the upload and download files from AWS S3 buckets. By uploading files to an specific folder in nucket we can trigger AWS Lambda invocations.

```
usage: scar put [-h] -b BUCKET -p PATH [-pf PROFILE]

optional arguments:
  -h, --help            show this help message and exit
  -b BUCKET, --bucket BUCKET
                        Bucket to use as storage
  -p PATH, --path PATH  Path of the file or folder
  -pf PROFILE, --profile PROFILE
                        AWS profile to use
```

```
usage: scar get [-h] -b BUCKET -p PATH [-pf PROFILE]

optional arguments:
  -h, --help            show this help message and exit
```

```
-b BUCKET, --bucket BUCKET
                        Bucket to use as storage
-p PATH, --path PATH  Path of the file or folder
-pf PROFILE, --profile PROFILE
                        AWS profile to use
```

Appendix B

OSCAR Template

The following pages show the template used to deploy the OSCAR infrastructure presented in chapter 5. The latest version of this template can be found in the official GitHub repository[1]

```
description kubernetes (
    kind = 'main' and
    short = 'Install and configure a cluster using the grycap.kubernetes
         ansible role and install all needed services to run OSCAR.' and
    content = 'The template installs the grycap.kubernetes ansible role.
         Initially the template creates as many working node hostnames
        as the sum of the values of feature "ec3_max_instances_max" in
        every system.

Webpage: https://kubernetes.io/'
)

network public (
  # kubernetes ports
  outbound = 'yes' and
  outports contains '443/tcp,22/tcp,6443/tcp,31112/tcp,32112/tcp,31852/
      tcp,8800/tcp'
)

network private ()
```

[1] https://github.com/grycap/oscar/blob/master/templates/oscar-latest.radl

```
system front (
  cpu.count>=2 and
  memory.size>=4096m and
  net_interface.0.connection = 'private' and
  net_interface.0.dns_name = 'kubeserver' and
  net_interface.1.connection = 'public' and
  net_interface.1.dns_name = 'kubeserverpublic' and
  queue_system = 'kubernetes' and
  ec3_templates contains 'kubernetes_oscar' and
  disk.0.applications contains (name = 'ansible.modules.grycap.
      kubernetes') and
  disk.0.applications contains (name = 'ansible.modules.grycap.nfs') and
  disk.0.applications contains (name = 'ansible.modules.grycap.kubefaas
      ') and
  disk.0.applications contains (name = 'ansible.modules.grycap.kubeminio
      ') and
  disk.0.applications contains (name = 'ansible.modules.grycap.
      kuberegistry') and
  disk.0.applications contains (name = 'ansible.modules.grycap.kubeoscar
      ') and
  disk.0.applications contains (name = 'ansible.modules.grycap.clues')
      and
  disk.0.applications contains (name = 'ansible.modules.grycap.im') and
  disk.1.type='standard' and
  disk.1.size=20GB and
  disk.1.device='vdf' and
  disk.1.fstype='ext4' and
  disk.1.mount_path='/pv/minio' and
  disk.2.type='standard' and
  disk.2.size=20GB and
  disk.2.device='vdg' and
  disk.2.fstype='ext4' and
  disk.2.mount_path='/pv/registry'
)

configure front (
@begin
---
  - vars:
      AUTH:
        ec3_xpath: /system/front/auth
      SYSTEMS:
        ec3_jpath: /system/*
      NNODES: '{{ SYSTEMS | selectattr("ec3_max_instances_max", "defined
          ") | sum(attribute="ec3_max_instances_max") }}'

    pre_tasks:
    - name: Create dir for kaniko builds
      file: path=/pv/kaniko-builds state=directory mode=755
    - name: Create auth file dir
      file: path=/etc/kubernetes/pki state=directory mode=755 recurse=
          yes
    - name: Create auth data file with an admin user
```

```
    copy: content='{{ lookup('password', '/var/tmp/dashboard_token
        chars=ascii_lowercase,digits length=16') }},kubeuser,100,"
        users,system:masters"' dest=/etc/kubernetes/pki/auth mode=600
  - name: Generate minio secret key
    set_fact:
      minio_secret: "{{ lookup('password', '/var/tmp/minio_secret_key
          chars=ascii_letters,digits') }}"

  roles:
  - role: 'grycap.nfs'
    nfs_mode: 'front'
    nfs_exports:
    - {path: "/pv/minio", export: "*.localdomain(rw,async,
        no_root_squash,no_subtree_check,insecure)"}
    - {path: "/pv/registry", export: "*.localdomain(rw,async,
        no_root_squash,no_subtree_check,insecure)"}
    - {path: "/pv/kaniko-builds", export: "*.localdomain(rw,async,
        no_root_squash,no_subtree_check,insecure)"}

  - role: 'grycap.kubernetes'
    kube_server: 'kubeserver'
    kube_apiserver_options:
    - {option: "--insecure-port", value: "8080"}
    - {option: "--token-auth-file", value: "/etc/kubernetes/pki/auth"}
    - {option: "--service-node-port-range", value: "80-32767"}
    kube_deploy_dashboard: true
    kube_install_metrics: true
    kube_persistent_volumes:
    - {namespace : "minio", name : "pvnfsminio", label : "minio",
        capacity_storage : "20Gi", nfs_path : "/pv/minio"}
    - {namespace : "docker-registry", name : "pvnfsregistry", label :
        "registry", capacity_storage : "20Gi", nfs_path : "/pv/
        registry"}
    - {namespace : "oscar", name : "pvnfskanikobuilds", label : "oscar
        -manager", capacity_storage : "2Gi", nfs_path : "/pv/kaniko-
        builds"}
    kube_version: 'latest'

  - role: 'grycap.kubefaas'
    faas_framework: 'openfaas'
    master_deploy: true

  - role: 'grycap.kubeminio'
    enable_notifications: true
    webhook_endpoints: [{ id: "1", endpoint: "http://oscar-manager.
        oscar:8080/events"}]
    minio_secretkey: '{{ minio_secret }}'
    master_deploy: true

  - role: 'grycap.kuberegistry'
    public_access: false
    type_of_node: "front"
    svc_name: "registry.docker-registry"
```

```
      delete_enabled: true
      master_deploy: true

    - role: 'grycap.kubeoscar'
      minio_pass: '{{ minio_secret }}'
      vue_app_backend_host: '{{ hostvars[groups["front"][0]]["
          IM_NODE_PUBLIC_IP"] }}:{{ nginx_https_nodeport }}'
      master_deploy: true

    - role: 'grycap.im'

    - role: 'grycap.clues'
      auth: '{{AUTH}}'
      clues_queue_system: kubernetes
      max_number_of_nodes: '{{ NNODES }}'
      vnode_prefix: 'wn'
      clues_config_options:
          - { section: 'scheduling', option: 'IDLE_TIME', value: '300' }
          - { section: 'scheduling', option: 'RECONSIDER_JOB_TIME',
              value: '60' }
          - { section: 'monitoring', option: 'MAX_WAIT_POWERON', value:
              '3000' }
          - { section: 'monitoring', option: 'MAX_WAIT_POWEROFF', value:
              '600' }
          - { section: 'monitoring', option: 'PERIOD_LIFECYCLE', value:
              '10' }
          - { section: 'monitoring', option: 'PERIOD_MONITORING_NODES',
              value: '2' }
          - { section: 'client', option: 'CLUES_REQUEST_WAIT_TIMEOUT',
              value: '3000' }
          # These options enable to have always one slot free
          - { section: 'scheduling', option: 'SCHEDULER_CLASSES', value:
              'clueslib.schedulers.CLUES_Scheduler_PowOn_Requests,
              clueslib.schedulers.CLUES_Scheduler_Reconsider_Jobs,
              clueslib.schedulers.CLUES_Scheduler_PowOff_IDLE, clueslib.
              schedulers.CLUES_Scheduler_PowOn_Free' }
          - { section: 'scheduling', option: 'EXTRA_SLOTS_FREE', value:
              '1' }

@end
)

system wn (
  cpu.count>=2 and
  memory.size>=4096m and
  ec3_node_type = 'wn' and
  net_interface.0.connection='private'
)

configure wn (
@begin
---
  - roles:
```

```
    - role: 'grycap.nfs'
      nfs_mode: 'wn'
      nfs_client_imports:
      - {local: "/pv/minio", remote: "/pv/minio", server_host: "
          kubeserver.localdomain"}
      - {local: "/pv/registry", remote: "/pv/registry", server_host: "
          kubeserver.localdomain"}
      - {local: "/pv/kaniko-builds", remote: "/pv/kaniko-builds",
          server_host: "kubeserver.localdomain"}

    - role: 'grycap.kubernetes'
      kube_type_of_node: 'wn'
      kube_server: 'kubeserver'
      kube_version: 'latest'

    - role: 'grycap.kuberegistry'
      public_access: false
      type_of_node: "wn"
      svc_name: "registry.docker-registry"
@end
)

include kube_misc (
  template = 'openports'
)

deploy front 1
```

Acronyms

AKS Azure Kubernetes Service.

API Application Programming Interface.

AWS Amazon Web Services.

BaaS Backend as a Service.

CaaS Container as a Service.

CD Continuous Delivery.

CI Continuous Integration.

CLI Command-Line Interface.

CLUES Cluster Elasticity System.

CNCF Cloud Native Computing Foundation.

CPU Central processing unit.

CRD Custom Resource Definition.

DNS Domain Name System.

EC2 Elastic Compute Cloud.

EC3 Elastic Cloud Computing Cluster.

ECS Elastic Container Service.

ECU EC2 Compute Unit.

EGI European Grid Infrastructure.

EKS Elastic Kubernetes Service.

FaaS Functions as a Service.

FAQ Frequently Asked Questions.

GCE Google Compute Engine.

GKE Google Kubernetes Engine.

GPU Graphics processing unit.

GUI Graphical User Interface.

HPC High Performance Computing.

HTC High Throughput Computing.

IaaS Infrastructure as a Service.

IM Infrastructure Manager.

LRMS Local Resource Management System.

NFS Network File System.

NIST National Institute of Standards and Technology.

OSCAR Open-source Serverless Computing for Data-Processing Applications.

PaaS Platform as a Service.

RADL Resource Application Description Language.

S3 Amazon Simple Storage Service.

SaaS Software as a Service.

SCAR Serverless Container-aware ARchitectures.

SDK Software Development Kit.

SNS Simple Notification Service.

TLS Transport Layer Security.

TOSCA Topology and Orchestration Specification for Cloud Applications.

URL Uniform Resource Locator.

VM Virtual Machine.

YAML YAML Ain't Markup Language.

YOLO You Only Look Once.